Understanding and Evolving the Rust Programming Language

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

vorgelegt von
Ralf Jung

Saarbrücken, August 2020

Tag des Kolloquiums
2020-08-21

Dekan der Fakultät für Mathematik und Informatik
Prof. Dr. Thomas Schuster

Prüfungsausschuss

Vorsitzender:	Prof. Dr. Sebastian Hack
Gutachter:	Prof. Dr. Derek Dreyer
	Dr. Viktor Vafeiadis
	Dr. François Pottier
Akademischer Mitarbeiter:	Dr. Rodolphe Lepigre

Abstract

Rust is a young systems programming language that aims to fill the gap between *high-level languages*—which provide strong static guarantees like memory and thread safety—and *low-level languages*—which give the programmer fine-grained control over data layout and memory management. This dissertation presents two projects establishing the first formal foundations for Rust, enabling us to better *understand* and *evolve* this important language: *RustBelt* and *Stacked Borrows*.

RustBelt is a formal model of Rust's type system, together with a soundness proof establishing memory and thread safety. The model is designed to verify the safety of a number of intricate APIs from the Rust standard library, despite the fact that the implementations of these APIs use *unsafe* language features.

Stacked Borrows is a proposed extension of the Rust specification, which enables the compiler to use the strong aliasing information in Rust's types to better analyze and optimize the code it is compiling. The adequacy of this specification is evaluated not only formally, but also by running real Rust code in an instrumented version of Rust's *Miri* interpreter that implements the Stacked Borrows semantics.

RustBelt is built on top of *Iris*, a language-agnostic framework, implemented in the Coq proof assistant, for building higher-order concurrent separation logics. This dissertation begins by giving an introduction to Iris, and explaining how Iris enables the derivation of complex high-level reasoning principles from a few simple ingredients. In RustBelt, this technique is exploited crucially to introduce the *lifetime logic*, which provides a novel separation-logic account of *borrowing*, a key distinguishing feature of the Rust type system.

Zusammenfassung

Rust ist eine junge systemnahe Programmiersprache, die es sich zum Ziel gesetzt hat, die Lücke zu schließen zwischen Sprachen mit hohem Abstraktionsniveau, die vor Speicher- und Nebenläufigkeitsfehlern schützen, und Sprachen mit niedrigem Abstraktionsniveau, welche dem Programmierer detaillierte Kontrolle über die Repräsentation von Daten und die Verwaltung des Speichers ermöglichen. Diese Dissertation stellt zwei Projekte vor, welche die ersten formalen Grundlagen für Rust zum Zwecke des besseren *Verständnisses* und der weiteren *Entwicklung* dieser wichtigen Sprache legen: *RustBelt* und *Stacked Borrows*.

RustBelt ist ein formales Modell des Typsystems von Rust einschließlich eines Korrektheitsbeweises, welcher die Sicherheit von Speicherzugriffen und Nebenläufigkeit zeigt. Das Modell ist darauf ausgerichtet, einige komplexe Komponenten der Standardbibliothek von Rust zu verifizieren, obwohl die Implementierung dieser Komponenten *unsichere* Sprachkonstrukte verwendet.

Stacked Borrows ist eine Erweiterung der Spezifikation von Rust, die es dem Compiler ermöglicht, den Quelltext mit Hilfe der im Typsystem kodierten Alias-Informationen besser zu analysieren und zu optimieren. Die Tauglichkeit dieser Spezifikation wird nicht nur formal belegt, sondern auch an echten Programmen getestet, und zwar mit Hilfe einer um Stacked Borrows erweiterten Version des Interpreters *Miri*.

RustBelt basiert auf *Iris*, welches die Konstruktion von Separationslogiken für beliebige Programmiersprachen im Beweisassistenten Coq ermöglicht. Diese Dissertation beginnt mit einer Einführung in Iris und erklärt, wie komplexe Beweismethoden mit Hilfe weniger einfacher Bausteine hergeleitet werden können. In RustBelt wird diese Technik für die Umsetzung der „Lebenszeitlogik“ verwendet, einer Erweiterung der Separationslogik mit dem Konzept von „Leihgaben“ (*borrows*), welche eine wichtige Rolle im Typsystem von Rust spielen.

Acknowledgments

For the past six years, while working towards this dissertation, I had the time of my life. I was able to follow my curiosity at my heart's desire, to travel the world, and—most importantly—to learn from and work with the most amazing people.

First and foremost, of course, I am indebted beyond words to my advisor, Prof. Derek Dreyer. In German, we call our PhD advisor "Doktorvater", doctoral father, and Derek really is my academic father in the best possible meaning of the word. He knew exactly how to convince me to work with him (by providing a never-ending series of interesting problems to solve), he knew when to throw me in at the deep end (by registering me for an overseas conference when I had no passport and never navigated an airport on my own), and above all he always supported me in every way I could imagine (like when I was in the USA without a credit card, or when my job plans changed abruptly due to a global pandemic). His eerie ability to ask exactly the right question after a long and deeply technical presentation of my latest ideas always baffled me. The German virtue of punctuality is not his strength, but when it counted, he was always there and always on time. I could not have done any of this without you, Derek.

Furthermore I would like to thank Viktor Vafeiadis and François Pottier for reviewing my dissertation and providing plenty of valuable feedback. I thank Prof. Gert Smolka and Prof. Sebastian Hack for providing advice throughout my undergraduate time and beyond. I also thank all the office and IT staff at MPI-SWS that helped with everything from travel bureaucracy to organizing a PhD defense during a pandemic to configuring our servers to just the right specification. Special thanks to Rose Hoberman for proof-reading large parts of my dissertation and to Rodolphe Lepigre for helping to squash plenty of typos.

All of the work described in this dissertation could not be done by one person alone, so I also thank all my academic collaborators. In particular I would like to thank Robbert Krebbers, with whom I wrote half of my papers. Robbert's dedication to detail and consistency is astonishing, as is his ability to do solid engineering in a patchwork of languages that grew organically over decades. Iris would not be what it is without him. He is the ideal collaborator, both fun to work with and to hang out with over a Belgian beer. A big thank you also to Jacques-Henri Jourdan, whose creativity for circumnavigating technical challenges was crucial for RustBelt.

I would like to thank Mozilla for two internships during which I developed what eventually became Stacked Borrows. In particular thank you to my internship advisors, Aaron Turon and Niko Matsakis. Aaron in fact helped initiate each of the three projects described in this dissertation: his post-doc with Derek briefly overlapped with my PhD, and when he left I inherited an early version of Iris from him. He left to Mozilla to work on Rust, which is when the idea for RustBelt started to shape. And later he invited me to do an internship with Mozilla Research, which eventually turned into Stacked Borrows. I also shamelessly copied the typesetting style of his dissertation. Niko gave me the chance to come back for a second internship, and provided me with the unique opportunity of influencing how a real language deals with unsafe code and undefined behavior.

Over the past decade, I had the fortune to become close friends with a group of nerdy computer scientists, filling my non-research time with delight as we play board games, hunt werewolves, search geocaches, or discuss the meaning of life over lunch. I love you all! You are exactly the people I need to have around me. In particular, I thank Elias, Tim, Jana, and Andrea for proof-reading part of my dissertation, and Janno for being my travel companion countless times and sticking with me when I got stuck on the wrong continent.

And finally, last but certainly not least, I thank my family. My father, knowingly or not, started this entire journey when he gifted me a book that teaches programming to children. He also used his editing superpower on almost my entire dissertation. My mother was always around when I needed her, but at the same time made sure I would learn to stand on my own feet. Whenever I started to be too sure of myself, my brother was ready to show me my limits. And lastly, I would like to mention Norbert Lukas, my late grandfather, an architect, who would have been so happy to see his grandson become a doctor of engineering. I dedicate this dissertation to him.

Saarbrücken, August 2020

Ralf Jung

Contents

Chapter 1

Introduction

Systems programming has, for the longest time, resisted the adoption of languages that provide any form of safety guarantee. The majority of low-level software is still written in C or C++, which—while having undergone serious modernization in recent years—still put most of the burden of ensuring basic memory safety onto the programmer. Even the most diligent programmer will make a mistake occasionally, and the resulting memory safety issues can cause real problems: both Microsoft and the Chrome team at Google report that around 70% of the security vulnerabilities in their products are caused by memory safety violations.[1]

Memory safety has been successfully achieved in many other languages by giving up some amount of control over how the program interacts with memory: the typical approach is to use garbage collection, restricting programmer control over both data layout and deallocation. However, in systems programming, where minimizing resource consumption (including CPU time and memory usage) is a primary concern, giving up that control is often not an option. Thus, a language is needed that can offer the same amount of control as C and C++ while also providing high-level safety guarantees.

Rust[2] is a young programming language that claims to fill this gap. Sponsored by Mozilla and developed in the open by a large and diverse community of contributors, it is already seeing industrial application in Mozilla's own Firefox browser, in small start-ups, and in big established companies.[3] Even Microsoft's Security Response Center Team recently announced that it is actively exploring the use of Rust to stem the tide of security vulnerabilities.[4]

Like C++, Rust gives the programmer control over memory management (letting the programmer determine how data is laid out in memory and when memory gets deallocated), and it puts a heavy emphasis on *zero-cost abstractions*:[5]

> What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better.

At the same time, and unlike C++, Rust promises to guarantee type safety and memory safety (no use-after-free, no double-free). More than that, Rust rules out common mistakes that plague many safe languages, such as *iterator invalidation*[6] where an iterator gets invalidated because the data structure it iterates over is mutated during the iteration. Rust is even able to guarantee that the program is free from *data races*, which

[1] Thomas, "A proactive approach to more secure code", 2019 [Tho19]; The Chromium project, "Chromium security: Memory safety", 2020 [The20].

[2] Rust teams, "Rust Programming Language", 2020 [Rus20].

[3] See `https://www.rust-lang.org/production` for some examples.

[4] Levick, "Why Rust for safe systems programming", 2019 [Lev19]; Burch, "Using Rust in Windows", 2019 [Bur19].

[5] Stroustrup, "The design and evolution of C++", 1994 [Str94]; Stroustrup, "Foundations of C++", 2012 [Str12].

[6] Bierhoff, "Iterator specification with typestates", 2006 [Bie06].

means there is no *unintended* communication between threads through shared memory. This goes beyond the safety guarantees of mainstream “safe” languages,[7] making Rust both a more powerful and a safer language.

That is the claim, anyway. This dissertation presents the first logical framework capable of formally proving these claims correct. Before laying out the structure of that framework, we[8] will provide some more background on Rust and explain what makes verifying Rust’s safety claims particularly challenging.

1.1 *Making systems programming safer with unsafe code?*

To demonstrate the kind of memory safety problems that arise commonly in systems programming languages, let us consider the following C++ code:

```
1  std::vector<int> v { 10, 11 };
2  int *vptr = &v[1]; // Points *into* ‘v‘.
3  v.push_back(12);
4  std::cout << *vptr; // Bug (use-after-free)
```

In the first line, this program creates a `std::vector` (a growable array) of integers. The initial contents of `v`, the two elements `10` and `11`, are stored in a buffer in memory. In the second line, we create a pointer `vptr` that points *into* this buffer; specifically it points to the place where the second element (with current value `11`) is stored. Now both `v` and `vptr` point to (overlapping parts of) the same buffer; we say that the two pointers are *aliasing*.

In line 3, we push a new element to the end of `v`. The element `12` is added after `11` in the buffer backing `v`. If there is no more space for an additional element, a new buffer is allocated and all the existing elements are moved over. Let us assume this is what happens here. Why is this case interesting? Because `vptr` still points to the old buffer! In other words, adding a new element to `v` has turned `vptr` into a dangling pointer. This is possible because both pointers were aliasing: an action through a pointer (`v`) will in general also affect all its aliases (`vptr`). We can visualize the situation after line 3 as follows:

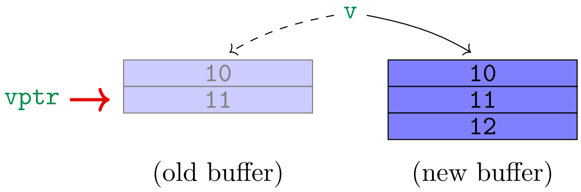

Figure 1.1: The shape of memory in the C++ pointer invalidation program after line 3.

The fact that `vptr` is now a dangling pointer becomes a problem in the last line. Here we load from `vptr`, and since it is a dangling pointer, this is a use-after-free bug.

In fact, the problem is common enough that one instance of it has its own name: *iterator invalidation*,[9] which we already mentioned earlier, basically reduces to the same pattern (where the pointer `vptr` is maintained internally by the iterator). It most commonly arises when one iterates over

[7] In contrast, Java gives very weakly defined semantics to data races. In Go and Swift, data races can even violate memory safety.

[8] Following common scientific practice, we use the first person plural to refer to work done by the author, whether or not it was done in collaboration with others. For further details, see §1.6.

[9] Gregor and Schupp, “Making the usage of STL safe”, 2002 [GS02].

some container data structure in a loop, and indirectly, but accidentally, calls an operation that mutates the data structure. Notice that in practice the call to the operation that mutates the data structure (`push_back` in line 3 of our example) might be deeply nested behind several layers of abstraction. In particular when code gets refactored or new features get added, it is often near impossible to determine if pushing to a certain vector will invalidate pointers elsewhere in the program that are going to be used again later.

In Rust, issues like iterator invalidation and null pointer misuse are detected statically, by the compiler—they lead to a compile-time error instead of a run-time exception or a security vulnerability. Consider the following Rust translation of our C++ example:

```
1  let mut v = vec![10, 11];
2  let vptr = &mut v[1]; // Points *into* ‘v‘.
3  v.push(12); // May reallocate the backing store of v.
4  println!("{}", *vptr); // Compiler error
```

Like in the C++ version, there is a buffer in memory, and `vptr` points into the middle of that buffer (causing aliasing); `push` might reallocate the buffer, which leads to `vptr` becoming a dangling pointer, and that leads to a use-after-free in line 4.

But none of this happens; instead the compiler shows an error: “cannot borrow `v` as mutable more than once at a time”. In §8, we will explain in detail how the Rust typechecker is able to detect this problem. For now, it suffices to say that Rust’s type system is more complex than the “average”, incorporating ideas such as *ownership* (a form of linear types) and a flow-sensitive analysis for tracking how pointers are *borrowed* for certain *lifetimes* (inspired by prior work on *regions*).[10]

However, there is a catch: the inner workings of `Vec` make significant use of direct pointer manipulation to achieve, *e.g.*, amortized constant-time pushing of elements to the vector (very much like `std::vector` in C++). One of the key choices the Rust designers made is *not* to build a type system that can verify safety of these kinds of low-level patterns that systems programmers regularly employ to make their code more efficient. Instead, Rust introduces the `unsafe` keyword to clearly mark syntactic scopes in which such potentially dangerous operations can be performed. This avoids accidentally leaving the realm of safe Rust by using such features.

At this point one might wonder, how does `unsafe` code not entirely subvert the safety guarantees Rust claims to provide? The answer lies in the power of *safe abstractions*. Rather than having an extremely complex type system that can verify the safety of a data structure that manually deals with uninitialized memory, concurrency or any other subtle aspect of low-level programming, Rust focuses on being able to help all the *clients* of such a data structure by making sure that they follow the contract that was intended by the implementer.

For example, `std::vector` in C++ comes with a lot of good documentation that explains how to properly use this type, pointing out issues such as the one in our example above. However, these are all just

[10] See §14 for citations and related work.

comments written for humans—there is no way for the *typechecker* to help the programmer follow those rules. In contrast, Rust's `Vec` comes with sufficiently detailed *types* so that the typechecker can check if `Vec` is used correctly. The authors of `Vec` claim that the `unsafe` operations they used have been properly "encapsulated" behind these types. This means that programmers who rely just on the public API and do not use `unsafe` themselves are shielded from any unsafe behaviors. The Rust type system is not strong enough to ensure correct encapsulation (it cannot prove this code to be safe), but it *is* strong enough to ensure that the public API of `Vec` is *used correctly*.

Shielding the users of `Vec` from unsafety relies on both ownership and pointer lifetimes being enforced by the typechecker. One can view the Rust type system as some kind of "language" suited to express the allowed interactions at an API surface. While this language does have some non-trivial vocabulary, it is arguably simpler than the type systems of prior work such as Cyclone[11] or Mezzo[12]—Rust's type system only has to be sufficient to express the *surface* of types like `Vec`; thanks to its reliance on `unsafe` code, it does not have to be able to express what is going on in their implementation.

Still, there could be a mismatch between what the author of `Vec` expects the type system to enforce, and what the type system really does. That risk is rather low for `Vec` as it only makes rather conventional use of the type system, but other standard library types like `Mutex` use the type system in more interesting ways. `Mutex` is a primitive for mutable state shared between concurrent threads, with run-time enforcement of mutual exclusion ensuring safety—in other words, it is a standard lock. The Rust type system is generally built around the idea that shared state is read-only, but types like `Mutex` use a clever combination of run-time and type-system enforcement to circumvent this restriction, enabling safe code to work with shared mutable state. This pattern is called *interior mutability*. To ensure that Rust's promise of safe systems programming holds true, it is important to be able to verify that abstractions such as `Vec` or `Mutex` safely encapsulate their unsafe implementation details.

And indeed, several bugs have been found in the Rust standard library where the public API of a type was insufficiently constrained to guarantee that it could not be used incorrectly by safe code.[13] Of these, "Leakpocalypse"[14] is particularly noteworthy because it only arises as a *combination* of multiple libraries, each of which is (believed to be) sound in isolation. A few years later, a similar case was found[15] where some standard library types as well as a new language feature turned out to be incompatible with a user-defined unsafely implemented library. This shows a clear need for some formal way of establishing not just safety of a single library, but also safe composition of a collection of libraries.

Unfortunately, the typical *syntactic* approach to verification of type soundness—due originally to Wright and Felleisen[16] and later refined and popularized as "progress and preservation"[17]—is unsuited for verifying Rust's "extensible" notion of safety. Progress and preservation is a *closed-world* method, *i.e.*, it assumes a fixed set of safe language primitives with their typing rules. The resulting theorem in Rust would only apply to

[11] Swamy *et al.*, "Safe manual memory management in Cyclone", 2006 [Swa+06].

[12] Balabonski, Pottier, and Protzenko, "The design and formalization of Mezzo, a permission-based programming language", 2016 [BPP16].

[13] Ben-Yehuda, "std::thread::JoinGuard (and scoped) are unsound because of reference cycles", 2015 [Ben15]; Biocca, "std::vec::IntoIter::as_mut_slice borrows &self, returns &mut of contents", 2017 [Bio17]; Jung, "MutexGuard<Cell<i32>> must not be Sync", 2017 [Jun17]; Jourdan, "Insufficient synchronization in Arc::get_mut", 2018 [Jou18].

[14] Ben-Yehuda, "std::thread::JoinGuard (and scoped) are unsound because of reference cycles", 2015 [Ben15].

[15] Jeffrey, "Rust 1.20 caused pinning to become incorrect", 2018 [Jef18].

[16] Wright and Felleisen, "A syntactic approach to type soundness", 1994 [WF94].

[17] Harper, "Practical foundations for programming languages (second edition)", 2016 [Har16].

programs that are not using any unsafely implemented library. Of course, the same is true for other languages with unsafe "escape hatches", such as `Obj.magic` in OCaml, `unsafePerformIO` in Haskell, or foreign-function calls to C code in pretty much any language. However, those escape hatches are typically considered niche features and thus declared out of scope of formalization efforts. This is not an option in Rust: `unsafe` code is used pervasively at the foundation of the Rust ecosystem, and hence must be taken into account by a realistic formalization of the language.

1.2 Understanding Rust: RustBelt

For this dissertation, we have developed RustBelt,[18] the first formal (and machine-checked) model of Rust which can account for Rust's extensible approach to safe systems programming and verify its soundness. Our soundness proof is *extensible* in the sense that, given a new unsafely implemented library and its type, RustBelt defines a proof obligation that is sufficient to show that any safe Rust program using this library is memory safe and thread safe. It is also *modular* in the sense that libraries can be verified independently and composed arbitrarily.

At the center of RustBelt is $\lambda_{\textsf{Rust}}$, a Rust core language, with its type system. For reasons of scale, $\lambda_{\textsf{Rust}}$ is not a full model of Rust, for which no formal description exists anyway. But crucially, $\lambda_{\textsf{Rust}}$ incorporates the core features that make up Rust's ownership discipline: *borrowing*, *lifetimes*, and *lifetime inclusion*. $\lambda_{\textsf{Rust}}$ is loosely inspired by MIR, the Mid-level Intermediate Representation in the Rust compiler on which correct usage of borrows is checked. For simplicity, we omit some orthogonal features of Rust such as traits (an adaptation of Haskell type classes), we avoid the subtleties of integer-pointer casts[19] by not supporting such an operation, and we circumvent the complications associated with weak memory[20] by adopting a simplified memory model supporting only non-atomic and sequentially-consistent atomic operations.[21] Nevertheless, $\lambda_{\textsf{Rust}}$ is a realistic enough model that we were able to uncover a previously unknown soundness bug in the Rust standard library.[22]

The key idea that makes the extensible soundness proof of RustBelt work is to define a *semantic model* of $\lambda_{\textsf{Rust}}$'s type system. This is hardly a new idea; in fact Milner's original type soundness proof for an ML-style polymorphic λ-calculus[23] followed the semantic approach, based on the idea of a *logical relation*.[24] A logical relation defines what a type "means"—what terms it is inhabited by—through the *observable behavior* of a term rather than "what the syntactic typing rules allow". In particular, for a function to be well-typed, only its input-output behavior is relevant; the body of the function can use unsafe features as long as it does not violate the type system guarantees.

Scaling that approach up to more powerful languages involving higher-order state turned out to be quite challenging, so the simpler—but less powerful—syntactic approach of progress and preservation mostly took over. Over the last two decades, however, work on logical relations has made a lot of progress,[25] and at this point there are established techniques for building "step-indexed" logical relations that can handle all type

[18] RustBelt is also the name of the larger research project of which this work is the cornerstone; for further information, see `https://plv.mpi-sws.org/rustbelt/`.

[19] Lee *et al.*, "Reconciling high-level optimizations and low-level code in LLVM", 2018 [Lee+18].

[20] Batty, "The C11 and C++11 concurrency model", 2015 [Bat15].

[21] Marrying $\lambda_{\textsf{Rust}}$ with a relaxed memory model is the topic of some exciting follow-on work (Dang *et al.*, "RustBelt meets relaxed memory", 2020 [Dan+20]).

[22] Jung, "MutexGuard<Cell<i32>> must not be Sync", 2017 [Jun17].

[23] Milner, "A theory of type polymorphism in programming", 1978 [Mil78].

[24] Tait, "Intensional interpretations of functionals of finite type I", 1967 [Tai67]; Plotkin, "Lambda-definability and logical relations", 1973 [Plo73].

[25] Appel, "Foundational proof-carrying code", 2001 [App01]; Ahmed *et al.*, "Semantic foundations for typed assembly languages", 2010 [Ahm+10]; Appel and McAllester, "An indexed model of recursive types for foundational proof-carrying code", 2001 [AM01]; Ahmed, "Semantics of types for mutable state", 2004 [Ahm04].

system features of a modern language like Rust, including recursive types and higher-order state.

In RustBelt, we follow the style of "logical" accounts of step-indexed logical relations.[26] This means we interpret the types of λ_{Rust} as predicates on values expressed in a suitably powerful logic, and we interpret the typing judgments of λ_{Rust} as logical entailments between these predicates.

With our semantic model in hand, the safety proof of λ_{Rust} divides into three parts:

1. Verify that the typing rules of λ_{Rust} are sound when interpreted semantically. For each typing rule we show a lemma establishing that the semantic interpretations of the premises imply the semantic interpretation of the conclusion. This is called *the fundamental theorem of logical relations*.
2. Verify that, if a closed program is *semantically* well-typed according to the model, it is safe to execute—there will be no safety violations. This is called *adequacy*.
3. For any library that employs `unsafe` code internally, verify that its implementation satisfies the "safety contract" of its public interface. This safety contract is determined by the semantic interpretations of the types of the library's public API. It establishes that all `unsafe` code has been properly "encapsulated" behind that API. In essence, the semantic interpretation of the interface yields a *library-specific verification condition*.

Together, these proofs establish that, as long as the only `unsafe` code in a well-typed λ_{Rust} program is confined to libraries that satisfy their verification conditions, the program is safe to execute.

Using the Coq proof assistant,[27] we have formally proven the fundamental theorem and adequacy (Theorem 4 and Theorem 5 in §12), and we have also proven the verification conditions for (λ_{Rust} models of) several standard Rust libraries that use `unsafe` code, including `Arc`, `Rc`, `Cell` (§13.1), `RefCell`, `Mutex` (§13.2), `RwLock`, `mem::swap`, `thread::spawn`, `rayon::join`, and `take_mut`.

While the high-level structure of our soundness proof is standard,[28] developing such a proof for a language as subtle and sophisticated as Rust has required us to tackle a variety of technical challenges. The two key challenges we would like to highlight here are the *choice of the right logic* and finding an appropriate *model of lifetimes and borrowing*.

Choosing the right logic for modeling Rust. "Which logic are you using" might sound like an odd question to ask, but for our semantic model of λ_{Rust} this was the most fundamental design choice. In Rust, many types encode not just information about a value, but *ownership* of some form, like exclusive ownership of a region of memory. While ownership can be encoded explicitly as part of building the semantic model (*e.g.*, by working with predicates over some suitable form of "resources"), doing so is tedious and error-prone, akin to writing a computer program in assembly language. Thus we choose to work at a higher level of abstraction by adopting a *separation logic*[29] as the framework in which we construct our semantic

[26] 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, Dreyer, and Birkedal, "Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency", 2013 [TDB13]; Krogh-Jespersen, Svendsen, and Birkedal, "A relational model of types-and-effects in higher-order concurrent separation logic", 2017 [KSB17].

[27] Coq Team, "The Coq proof assistant", 2020 [Coq20].

[28] Milner, "A theory of type polymorphism in programming", 1978 [Mil78]; Ahmed, "Semantics of types for mutable state", 2004 [Ahm04].

[29] Reynolds, "Separation logic: A logic for shared mutable data structures", 2002 [Rey02]; O'Hearn, "Resources, concurrency, and local reasoning", 2007 [OHe07].

model. Using a separation logic means we can use its connectives such as points-to ($\mapsto$) and the separating conjunction ($*$), which lend themselves very naturally to encode the basic ownership involved in types like `Vec`.

However, basic separation logic is not sufficient: it is very limited in the kinds of ownership it can express. To achieve maximal expressiveness when defining the semantic interpretation of Rust's types, we need a more flexible logic. This is where *Iris*[30] enters the picture. Iris is a language-agnostic framework for building higher-order concurrent separation logics. On top of that, Iris comes with built-in support for step-indexing, avoiding the tedious manual bookkeeping that usually comes with step-indexed logical relations. Moreover, Iris lets the user introduce their own notion of "ownership", based on the idea of *fictional separation*,[31] where a single piece of shared state can be *logically* divided between multiple threads. And finally, Iris comes with excellent support for interactive machine-checked proofs in Coq.[32] Iris is the first major project described in this dissertation, and forms the foundation for the later chapters on RustBelt.

Modeling lifetimes and borrowing. Iris provides built-in support for reasoning about many forms of ownership. But for a complete model of λ_{Rust} types, we also need to express logically what it means to *borrow* something for a certain *lifetime*.[33] This is where we exploit that Iris was designed from the start to *derive* new reasoning principles *inside the logic*.

Using all key features of Iris, including *impredicative invariants*[34] and *higher-order ghost state*,[35] we construct the novel *lifetime logic* and verify its correctness in Iris. The primary feature of the lifetime logic is a logical proposition which mirrors the "borrowing" mechanism of the Rust type system. This lifetime logic makes it possible for us to give fairly direct interpretations of a number of Rust's most semantically complex types, and to verify their soundness at a high level of abstraction. A detailed description of the lifetime logic (§11) sits at the heart of the second main project in this dissertation, RustBelt.

1.3 Evolving Rust: Stacked Borrows

Type systems like that of Rust are useful not only for making programs more secure and reliable, but also for helping compilers generate more efficient code. For example, a language with a strong type system does not have to waste time and space maintaining dynamic type information on all run-time data. In Rust, the type system imposes a strict discipline on pointer aliasing. It is thus a natural question to ask whether that static discipline can be exploited by the compiler.

In particular, *mutable references* `&mut T` in Rust are unique pointers that cannot alias with anything else in scope. This knowledge should enable us to optimize the following function:

```
fn example1(x: &mut i32, y: &mut i32) -> i32 {
  *x = 42;
  *y = 13;
  return *x; // Has to read 42, because x and y cannot alias!
}
```

[30] 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].

[31] Dinsdale-Young, Gardner, and Wheelhouse, "Abstraction and refinement for local reasoning", 2010 [DGW10]; Dinsdale-Young *et al.*, "Concurrent abstract predicates", 2010 [Din+10].

[32] 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].

[33] The Rust concepts of borrowing and lifetimes will be explained in §8.

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

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

Since mutable references are unique, `x` and `y` cannot alias. As a consequence, the compiler should be allowed to assume that the read in line 4 will yield `42`, so it can remove the memory access and replace it by a constant. Typically, the way such an optimization would be justified is through *reordering* of instructions. If we know `x` and `y` do not alias, then we can reorder lines 2 and 3, and it becomes clear that the value read from `x` in line 4 must be the value just written to it, namely `42`.[36]

Having access to this kind of alias information is a compiler writer's dream, in part because it is essential in justifying reorderings and other program transformations which are key to improving code generation,[37] and in part because such alias information is typically hard to come by.[38] In particular, the alias analysis that is possible in most programming languages is fundamentally limited by the weakness of conventional type systems. For example, in cases like the one above, where `x` and `y` are passed to a function from its environment, a C/C++ compiler cannot make any *local* assumption about whether these pointers alias with each other or with any other pointer the program might be using—it would need to perform a more global, *interprocedural* analysis, which is often prohibitively expensive (or impossible if the whole program is not available). In contrast, because it enforces a strict discipline of pointer usage, the Rust type system provides a rich source of alias information that can be used to justify transformations like the one above *intraprocedurally*.

Unfortunately, things are not quite so easy: as mentioned above, Rust supports direct pointer manipulation using "raw pointers" in `unsafe` code. It is easy to write `unsafe` code that, when compiled with the current compiler,[39] makes the above function return `13`:

```
1  fn main() {
2    let mut local = 5;
3    let raw_pointer = &mut local as *mut i32;
4    let result = unsafe {
5      example1(&mut *raw_pointer, &mut *raw_pointer)
6    };
7    println!("{}", result); // Prints "13".
8  }
```

In line 3, this code uses the `as` operator to cast a *reference* to `local` (that would have type `&mut i32`) into a *raw pointer* with type `*mut i32`. Raw pointers are not tracked by the type system: similar to pointers in C, they can be freely interconverted with integers in safe code, and arbitrary aliasing and address arithmetic are possible. To maintain soundness of the type system, *dereferencing* a raw pointer is only permitted inside `unsafe` blocks, ensuring that the programmer explicitly opted-in to potentially unsafe behavior. Raw pointers exist to allow implementing low-level pointer-heavy data structures in Rust that internally do not follow Rust's aliasing discipline, and they are also often necessary when interfacing with C code through a foreign-function interface (FFI).

In the example above, however, we are using raw pointers for a more sinister purpose. In line 5, we convert the raw pointer back to a reference. `&mut *raw_pointer` dereferences the raw pointer and immediately takes the address again, so this is effectively a cast from `*mut i32` back to

[36] Concurrent mutation is ruled out because that would introduce a data race, which is considered undefined behavior.

[37] Ghiya, Lavery, and Sehr, "On the importance of points-to analysis and other memory disambiguation methods for C programs", 2001 [GLS01]; Wilson and Lam, "Efficient context-sensitive pointer analysis for C programs", 1995 [WL95].

[38] Horwitz, "Precise flow-insensitive may-alias analysis is NP-hard", 1997 [Hor97].

[39] All tests were done with the Rust stable release 1.35.0 in release mode.

`&mut i32`. The sinister part about this cast is that we do it twice! The type system does not stop us, as it does not even attempt to track what happens with raw pointers. As a result of all of this, we call `example1` with two aliasing references that both point to `local`, and the function returns `13`. This means our desired optimization of always returning `42` observably changes program behavior.

It is tempting to ignore this problem because it "just" concerns `unsafe` code. However, that would be neglecting the important role `unsafe` code plays in the Rust ecosystem. Not all Rust code is safe—in fact pretty much all programs depend on *some* `unsafe` code—but that does not mean that Rust's safety guarantees are useless. As we have discussed, the important part is that `unsafe` code is *encapsulated within a safe abstraction*. This approach lets Rust programmers use data structures such as `Vec` without worrying about memory safety issues, while at the same time not compromising efficiency when compared with unsafe languages such as C or C++. The Rust ecosystem rests on the interplay of safe code where possible and `unsafe` code where needed, and hence any realistic consideration of compilation for Rust programs requires proper treatment of `unsafe` code.

So, let us look at our example program again.[40] We want to argue that it is correct to compile this program such that it prints `42`, but the only way we can make such an argument is by tweaking the operational semantics! The fact that the program "circumvents" the type system is irrelevant, because the type system is not involved in the definition of what it means for a Rust compiler to be correct. A compiler correctness statement that only applies to well-typed programs (in the safe fragment of Rust) would be useless for all practical purposes, because—as argued above—most programs being compiled contain `unsafe` code. So, with the type system being ruled out, the only knob we have left is the *dynamic semantics* of the source language: we have to define the behavior of our source program in a way that, actually, it *is* allowed for the program to output `42`.

In the third main part of this dissertation, we describe *Stacked Borrows*, an operational semantics for pointer aliasing in Rust. Stacked Borrows says that our example program has *undefined behavior*,[41] which means the compiler is allowed to compile it in any way it chooses.

Defining a semantics with that property is not a simple task. A naive semantics, such as the one used in λ_{Rust}, will give the example program a defined meaning and thus force the compiler to print `13`. Compared to such a naive semantics, we have to "add" some undefined behavior to obtain the desired optimizations. But of course we should not add "too much" undefined behavior! We have to be careful that "desired" programs are still well-defined. This includes all safe programs, but should also include enough `unsafe` programs to still make `unsafe` Rust a useful language for implementing data structures such as `Vec`, and to minimize the chances of programmers accidentally running into undefined behavior.

Stacked Borrows enforces Rust's aliasing discipline by introducing clearly defined conditions under which a Rust program exhibits undefined

[40] Together, `main` and `example1` form a closed program.

[41] Wang *et al.*, "Undefined behavior: What happened to my code?", 2012 [Wan+12].

behavior due to an aliasing violation. The key idea is to define a dynamic version of the static analysis—called the *borrow checker*—which Rust already uses to check that references are accessed in accordance with its aliasing discipline. The borrow checker enforces in particular that references that might be aliasing are used in a "well-nested" manner, in a sense that we will define in §15. We model this discipline in our dynamic analysis with the use of a per-location *stack*:[42] our dynamic analysis detects when references get used in a non-stack-like way, and flags such programs as having undefined behavior. Based on that core structure, we then extend Stacked Borrows with rules for raw pointers (which the static borrow checker ignores) with the goal of being maximally liberal while not interfering with the key properties of the "safe fragment" of our dynamic analysis.

We have validated Stacked Borrows in two ways:

- To ensure that Stacked Borrows does not introduce "too much" undefined behavior, we have equipped Miri,[43] an existing interpreter for Rust programs, with an implementation of Stacked Borrows. We have run the OS-independent part of the Rust standard library test suite[44] in this interpreter. The majority of tests passed without any adjustments. We also uncovered a few violations of the model, almost all of which have been accepted by the Rust maintainers as bugs and have since been fixed in the standard library. For further discussion, see §18.
- To ensure that Stacked Borrows has "enough" undefined behavior, we give proof sketches demonstrating that a few representative compiler transformations, such as the one in `example1`, are legal under this semantics. These proof sketches are all backed by fully mechanized proofs in Coq,[45] based on a formalization of Stacked Borrows and a framework for open simulations.[46]

1.4 Overview and contributions

The remainder of this dissertation is structured into the three main parts described above: Iris, RustBelt, and Stacked Borrows. Each part begins with a brief overview outlining the material covered by its individual chapters. The conclusion in §20 summarizes this dissertation's work and the impact it has had so far, and suggests some avenues for future research.

Contributions. Part I is about Iris. Iris is a multi-institution collaborative project,[47] and there are many things to say about its high-level design, its soundness proof, and its implementation in Coq. To focus the presentation, this part of the thesis explains the motivation behind the design of Iris and introduces the pieces of the logic that are required to understand RustBelt. For more details on the lower-level aspects of how Iris is implemented, we refer the reader to "Iris from the ground up".[48]

In Part II, we describe RustBelt. The contribution here is a formal model of core Rust (focusing on ownership, borrowing, and lifetimes) together with a syntactic type system, and a semantic model of said type system. This semantic model is based on the novel *lifetime logic*, which

[42] A "stack of borrows", so to say.

[43] Miri is available at `https://github.com/rust-lang/miri/`.

[44] Concretely, the tests for `libcore`, `liballoc`, and `HashMap`.

[45] Coq Team, "The Coq proof assistant", 2020 [Coq20].

[46] Hur *et al.*, "The marriage of bisimulations and Kripke logical relations", 2012 [Hur+12].

[47] `https://iris-project.org/`

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

equips separation logic with a notion of *borrowing*. On top of all that, we show how to formally verify semantic safety of some tricky Rust libraries that use `unsafe` code.

The final Part III is about Stacked Borrows. We propose a set of precise rules controlling when aliasing is allowed in Rust and when it is not. We give sketches of proofs (which have been formalized in Coq) demonstrating that these rules indeed are sufficient to ensure correctness of some simple but powerful optimizations. We also evaluate an implementation of these rules in Miri, a Rust interpreter, to validate that real `unsafe` code complies with the rules of the model.

1.5 Publications

This dissertation contains text and material from several publications:

- "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning" [Jun+15], which appeared in POPL 2015, is the first paper on Iris, later called "Iris 1.0".
- "Iris from the ground up: A modular foundation for higher-order concurrent separation logic" [Jun+18b], which appeared in the Journal of Functional Programming in 2018, is a significantly revised and expanded synthesis of the "Iris 2.0"[49] and "Iris 3.0"[50] papers, which appeared in ICFP 2016 and ESOP 2017, respectively. The Iris introduction (§3) is an expanded version of the one given in this paper, as is the discussion of the Iris ghost state combinators (§4.1), the global ghost state (§4.3), and the higher-order ghost state paradox (§6.1).
- "RustBelt: Securing the foundations of the Rust programming language" [Jun+18a], which appeared in POPL 2018, is the basis of Part II about RustBelt. However, Part II far exceeds the conference paper in depth and breadth of the presented material.
- "Safe systems programming in Rust: The promise and the challenge" [Jun+20b], a general-audience article which has been accepted to CACM, is the source of the initial motivating example for Rust in §1.1.
- "Stacked Borrows: An aliasing model for Rust" [Jun+20a], which appeared in POPL 2020, has been adapted into Part III (as well as parts of the introduction and the Rust tutorial in §8) with only a few adjustments.

The following list details the origin of the text of the individual chapters:

- §1 (Introduction) takes some material from the RustBelt and Stacked Borrows papers and the CACM article, combined with a lot of new text.
- §2 (Why Iris?) is mostly new text, combined with material from the Iris journal paper ("Iris from the ground up").
- §3 (An introduction to Iris) is based on the Iris journal paper, but the example and explanation are expanded to showcase more features of Iris.

[49] Jung *et al.*, "Higher-order ghost state", 2016 [Jun+16]. Note that this paper is the author's master's thesis and its contributions are thus not contributions of this dissertation. See §1.6 for more details.

[50] Krebbers *et al.*, "The essence of higher-order concurrent separation logic", 2017 [Kre+17].

- §4 (Ghost state constructions) takes some of the description of general resource algebra (RA) combinators (§4.1) and the global RA construction (§4.3) from the Iris journal paper. The description of how to encode a heap using the Iris RA combinators as well as §4.2 and §4.4 are new text.
- §5 (Invariants and modalities) is new text.
- §6 (Paradoxes) contains a description of the higher-order ghost state paradox from the Iris journal paper (§6.1). §6.2 is new.
- §7 (Key differences to prior work) is mostly new, but parts of §7.2 are taken from the Iris journal paper.
- §8 (Rust 101) is a combination of the Rust tutorials of the RustBelt and Stacked Borrows papers.
- §9 (The λ_{Rust} language and type system) is based on the RustBelt paper, but I expanded the discussion of the operational semantics and added another example for typechecking a piece of λ_{Rust} code. The text in the "appendix" in §9.4 is new.
- §10 (A semantic model of λ_{Rust} types in Iris) is taken from the RustBelt paper, but §10.2 and the "case of the missing $\triangleright$" are new.
- §11 (Lifetime logic) is a significantly expanded version of the corresponding section of the RustBelt paper. §11.4–§11.7 are entirely new.
- §12 (Semantic type system soundness) is almost entirely new, with just the initial exposition of the theorems being based on the RustBelt paper.
- §13 (Modeling types with interior mutability) is almost entirely new; the RustBelt paper contains just a brief description of the semantic interpretations of `Cell` and `Mutex`.
- §14 (Related work) is an expanded version of the related work section of the RustBelt paper.
- §15–§19 (Stacked Borrows) are taken from the Stacked Borrows paper with only minor adjustments.
- §20 (Conclusion) is mostly new; some of the description of future work is based on the Iris journal paper, the RustBelt paper, and the Stacked Borrows paper.

1.6 Collaborations

All of the work presented in this dissertation is the result of collaborative projects which I spearheaded.[51] Here, I explain the nature and extent of my specific contributions to each part.

As mentioned above, Iris (Part I) is a major project with involvement from multiple research institutions. The design and implementation of the logic have enjoyed many iterative refinements and improvements to the system from a variety of contributors over time. As such, it is difficult to tease apart precisely who is responsible for what. Rather than attempting to carve out a story about Iris that emphasizes my own individual contributions, I have instead aimed to construct a narrative that best prepares the reader to understand RustBelt, since the latter is the heart of this dissertation.

[51] In just this section, I will use the first person singular to refer specifically to my own contributions in contrast to those of my collaborators.

That said, I have certainly played a leading role in Iris's development from the very beginning of the project. I was first author of the original "Iris 1.0" paper,[52] and in addition to leading the writing of the paper, I was the one primarily responsible for a number of its main technical innovations, including: the encoding of general *state-transition systems* with tokens (in the spirit of Turon *et al.*[53]) in terms of a partial commutative monoid (§4.2), the pattern of *authoritative* resources guarded by an invariant (§4.4, inspired by Krishnaswami *et al.*[54]), the factoring of the typical partial commutative monoid for heaps into a combination of several independent constructions (§4.1), and the formalization of *logically atomic triples* (not covered in this dissertation but central to the story of the original Iris paper and developed further in recent follow-on work[55]). I also turned the view shift binary connective into a modality[56] (§5.5), and generalized that modality to carry two masks, thereby introducing the idea of *mask-changing view shifts* (§5.3).

In the subsequent papers on the evolving design of Iris,[57] I continued to play a leading role in the writing and overall development, as well as to be personally responsible for a number of key technical contributions. In particular, I worked out the *accessor* specification pattern based on mask-changing view shifts (§5.6), a pattern which is not actually described in any of the Iris publications so far but has been used in the "Iris 3.0" paper, in the lifetime logic, and in simplifying the encoding of logically atomic triples.[58] I also showed that we can encode cancellable invariants using the persistent invariants of Iris and some extra ghost state (§5.2). Finally, I uncovered some boundaries in the design space of higher-order separation logics by proving two paradoxes (§6): naive higher-order ghost state is unsound, and impredicative invariants can be used to violate linearity.

Note that the "Iris 2.0" paper is my master's thesis. Its contributions—in particular, the foundations of higher-order ghost state and the introduction of a "core" to Iris resource algebras—are thus *not* to be considered contributions of this dissertation. However, some of this material is essential to how Iris works, and is thus reviewed in the Iris tutorial (§3).

One key reason for the success of Iris has been its effective implementation in Coq, in particular the Iris Proof Mode[59] and its successor, MoSeL.[60] These contributions are due primarily to Robbert Krebbers, not me. However, the Coq implementation of Iris is under constant development by a growing team of contributors,[61] and I serve as one of its three core maintainers (together with Robbert Krebbers and Jacques-Henri Jourdan). Moreover, my collaborators insist that I mention that the Iris project would not have "taken off" to the extent that it did, were it not for my efforts to set up and maintain the infrastructure for an open-source project, including public version control, bug tracking, code review, and continuous integration.

Regarding RustBelt (Part II), I developed the initial version of the type system (§9), its semantic model (§10, §12), and the lifetime logic rules (§11) myself. Jacques-Henri Jourdan had the key idea of introducing *lifetime intersection* into the lifetime logic. I fully developed the model of the lifetime logic in Iris on paper, except for some details of the final

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

[53] Turon *et al.*, "Logical relations for fine-grained concurrency", 2013 [Tur+13].

[54] Krishnaswami *et al.*, "Superficially substructural types", 2012 [Kri+12].

[55] Jung *et al.*, "The future is ours: Prophecy variables in separation logic", 2020 [Jun+20c]; Jung, "Logical atomicity in Iris: The good, the bad, and the ugly", 2019 [Jun19d].

[56] This "primitive view shift" modality was not mentioned in the Iris publications, but appeared as `vs` in the technical appendix of the first Iris paper [Jun+15]. In follow-on work by Robbert Krebbers, Aleš Bizjak, and me [Kre+17], we managed to define that modality in terms of lower-level logical primitives, at which point we renamed it to *fancy update* modality.

[57] 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].

[58] Jung, "Logical atomicity in Iris: The good, the bad, and the ugly", 2019 [Jun19d].

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

[60] Krebbers *et al.*, "MoSeL: A general, extensible modal framework for interactive proofs in separation logic", 2018 [Kre+18].

[61] `https://gitlab.mpi-sws.org/iris/iris/-/blob/master/CHANGELOG.md` provides a full list of who contributed to which Iris release.

induction in §11.7 which Jacques-Henri Jourdan, Robbert Krebbers, and I worked out together while formalizing my arguments in Coq. Jacques-Henri Jourdan and I worked together on verifying the various `unsafe` libraries. The Coq formalization was led by Jacques-Henri Jourdan, but also received major contributions from Robbert Krebbers and me. Finally, I led the writing of the RustBelt paper,[62] although it also received significant contributions from all co-authors.

For Stacked Borrows (Part III), I developed the rules and implementation of Stacked Borrows in Miri as part of an internship with Mozilla. The later formal definition (§17) and correctness proof in Coq were led by Hoang-Hai Dang, with some assistance from Jeehoon Kang, but I helped to determine the key invariant of the proof and to complete its mechanization in Coq. I wrote the text of the Stacked Borrows paper[63] almost entirely myself, with just minor input from my coauthors.

All results presented in this dissertation have been formally verified in Coq. As declared above, I did not lead these proof mechanization efforts, but I did contribute significantly to them. The respective Coq sources are available under the following URLs:

- Part I, Iris:
  `https://gitlab.mpi-sws.org/iris/iris`
- Part II, RustBelt:
  `https://gitlab.mpi-sws.org/iris/lambda-rust`
- Part III, Stacked Borrows:
  `https://gitlab.mpi-sws.org/FP/stacked-borrows`

[62] Jung *et al.*, “RustBelt: Securing the foundations of the Rust programming language”, 2018 [Jun+18a].

[63] Jung *et al.*, “Stacked Borrows: An aliasing model for Rust”, 2020 [Jun+20a].

This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project “RustBelt”, funded under the European Union’s Horizon 2020 Framework Programme (grant agreement no. 683289).

Part I

Iris

Iris is a separation logic framework for building program logics and logical relations. Within just a few years, it has already been adopted by several research groups: more than twenty publications[1] and four dissertations[2] have made use of Iris. In particular, the second part of this dissertation—RustBelt—builds on top of Iris. The purpose of this first part is two-fold: to explain the motivation and key ideas of Iris and compare it to related work, and furthermore to lay the foundations that will be needed to understand the technical aspects of the second part.

This part begins by explaining how Iris fits into the general line of research on separation logics (§2). We proceed with a tour demonstrating the key features of Iris with a simplified example (§3). Having laid the foundations, we then go into more detail about ghost state (§4) and invariants (§5) in Iris, with a focus on how to define complex abstractions from simpler foundations. We close with a discussion of two paradoxes that establish clear limitations in the design of Iris (§6) and a discussion of the key design decisions that distinguish Iris from its predecessors (§7).

[1] That many citations do not fit in the margin, so we list them at the bottom of this page instead.

[2] Timany, "Contributions in programming languages theory: Logical relations and type theory", 2018 [Tim18]; Krogh-Jespersen, "Towards modular reasoning for stateful and concurrent programs", 2018 [Kro18]; Tassarotti, "Verifying concurrent randomized algorithms", 2019 [Tas19]; Krishna, "Compositional abstractions for verifying concurrent data structures", 2019 [Kri19].

Krogh-Jespersen, Svendsen, and Birkedal, "A relational model of types-and-effects in higher-order concurrent separation logic", 2017 [KSB17]; Tassarotti, Jung, and Harper, "A higher-order logic for concurrent termination-preserving refinement", 2017 [TJH17]; Kaiser *et al.*, "Strong logic for weak memory: Reasoning about release-acquire consistency in Iris", 2017 [Kai+17]; Swasey, Garg, and Dreyer, "Robust and compositional verification of object capability patterns", 2017 [SGD17]; Timany *et al.*, "A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of `runST`", 2018 [Tim+18]; Jung *et al.*, "RustBelt: Securing the foundations of the Rust programming language", 2018 [Jun+18a]; Frumin, Krebbers, and Birkedal, "ReLoC: A mechanised relational logic for fine-grained concurrency", 2018 [FKB18]; Tassarotti and Harper, "A separation logic for concurrent randomized programs", 2019 [TH19]; Bizjak *et al.*, "Iron: Managing obligations in higher-order concurrent separation logic", 2019 [Biz+19]; Mével, Jourdan, and Pottier, "Time credits and time receipts in Iris", 2019 [MJP19]; Frumin, Gondelman, and Krebbers, "Semi-automated reasoning about non-determinism in C expressions", 2019 [FGK19]; Timany and Birkedal, "Mechanized relational verification of concurrent programs with continuations", 2019 [TB19]; Chajed *et al.*, "Verifying concurrent, crash-safe systems with Perennial", 2019 [Cha+19]; Sammler *et al.*, "The high-level benefits of low-level sandboxing", 2020 [Sam+20]; Jung *et al.*, "The future is ours: Prophecy variables in separation logic", 2020 [Jun+20c]; Hinrichsen, Bengtson, and Krebbers, "Actris: Session-type based reasoning in separation logic", 2020 [HBK20]; de Vilhena, Pottier, and Jourdan, "Spy game: Verifying a local generic solver in Iris", 2020 [dPJ20]; Dang *et al.*, "RustBelt meets relaxed memory", 2020 [Dan+20]; Krogh-Jespersen *et al.*, "Aneris: A mechanised logic for modular reasoning about distributed systems", 2020 [Kro+20]; Krishna, Patel, and Shasha, "Verifying concurrent search structure templates", 2020 [KPS20]; Giarrusso *et al.*, "Scala step-by-step: Soundness for dot with step-indexed logical relations in Iris", 2020 [Gia+20]; Frumin, Krebbers, and Birkedal, "Compositional non-interference for fine-grained concurrent programs", 2021 [FKB21].

Chapter 2

Why Iris?

Iris grew out of a long line of work on separation logic, starting with the seminal work of O'Hearn, Reynolds, and Yang[1] and continuing with concurrent separation logic[2] and plenty of follow-on work.[3] To explain the motivation behind Iris, we begin with a brief review of this history.

2.1 Separation logic

Separation logic was introduced 20 years ago[4] as a derivative of Hoare logic geared towards reasoning more modularly and scalably about pointer-manipulating programs.

The poster child of separation logic is the *points-to assertion*: $\ell \mapsto v$ is a logical statement expressing *knowledge* that the current value stored at location ℓ is v, and also expressing *ownership* of said location by the current code. Ownership implies that there cannot be another part of the program that can "interfere" by updating ℓ; in fact, the rest of the program cannot even read ℓ or make any kind of statement about this allocation. The *right* to mutate or even deallocate ℓ is only granted to code that holds ownership.

This is in contrast to most non-separation logics where statements like "ℓ points to v" (suitably formalized) are inherently *unstable*: because there is no notion of "rights" or "ownership", any other code (be it another thread, or a callback we are calling in higher-order code) could write to ℓ any time, which would make "ℓ points to v" no longer true. Separation logic provides a way to make stable statements in an unstable world: $\ell \mapsto v$ is stable under interference from the rest of the program even though it is a statement about mutable state in an imperative language.[5]

Besides the points-to assertion, separation logic also introduced (inspired by earlier work on "bunched implications"[6]) a new way of *composing* assertions: the *separating conjunction* $P * Q$ asserts that not only do P and Q both hold, they actually hold for *disjoint parts* of the heap. The usefulness of the separating conjunction is best demonstrated by considering the *frame rule* of separation logic:

$$\frac{\{P\}\, e\, \{Q\}}{\{P * R\}\, e\, \{Q * R\}}\ \textsc{frame}$$

This rule says that if we have a proof of some specification for e with precondition P and postcondition Q (say, some specification that was verified

[1] O'Hearn, Reynolds, and Yang, "Local reasoning about programs that alter data structures", 2001 [ORY01].

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

[3] Vafeiadis and Parkinson, "A marriage of rely/guarantee and separation logic", 2007 [VP07]; Feng, Ferreira, and Shao, "On the relationship between concurrent separation logic and assume-guarantee reasoning", 2007 [FFS07]; Feng, "Local rely-guarantee reasoning", 2009 [Fen09]; Dodds *et al.*, "Deny-guarantee reasoning", 2009 [Dod+09]; Dinsdale-Young *et al.*, "Concurrent abstract predicates", 2010 [Din+10]; Fu *et al.*, "Reasoning about optimistic concurrency using a program logic for history", 2010 [Fu+10]; Turon, Dreyer, and Birkedal, "Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency", 2013 [TDB13]; Svendsen and Birkedal, "Impredicative concurrent abstract predicates", 2014 [SB14]; Nanevski *et al.*, "Communicating state transition systems for fine-grained concurrent resources", 2014 [Nan+14]; da Rocha Pinto, Dinsdale-Young, and Gardner, "TaDA: A logic for time and data abstraction", 2014 [dDG14].

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

[5] Some descendants of CSL combine "unstable" assertions with separation logic. We will come back to this point in §7.1.

[6] O'Hearn and Pym, "The logic of bunched implications", 1999 [OP99]; Ishtiaq and O'Hearn, "BI as an assertion language for mutable data structures", 2001 [IO01].

for a function), and if before executing e we have some *extra* ownership R on top of the precondition P, then we can *frame* R around e and conclude that we still own R after e returns, on top of the postcondition Q.[7]

FRAME is an extremely powerful principle. The separating conjunction is key to even being able to express this rule. Imagine for a second that we would replace the separating conjunction by a plain conjunction ($\land$). Then clearly, there is no way this frame rule could hold, as we could derive the following nonsense rule (by picking $R := P$):

$$\frac{\{P\}\,e\,\{Q\}}{\{P\}\,e\,\{Q \land P\}}$$

Basically, anything that was true before e would still be true afterwards. In an imperative language where pre- and postconditions can refer to mutable state, this cannot be true—statements such as P are in general *not stable.*

The reason the frame rule works is that it uses *separating conjunction* to make sure that the ownership R that is "framed around" e is *disjoint* from the precondition P. The frame rule says that *any state not explicitly described in the precondition is unaffected by* e.

To summarize, separation logic enables local reasoning about mutable state by (a) giving proofs a local way to talk about mutable state while excluding interference from other code, through ownership expressed by the *points-to assertion*; and (b) letting proofs preserve ownership of unrelated parts of the heap when applying specifications, through *framing.*

2.2 Concurrent separation logic

With the introduction of *concurrent separation logic* (CSL), O'Hearn and Brookes demonstrated that separation logic is a great fit for verification of concurrent programs.[8]

Parallel composition. The key proof rule of CSL is the following rule for *parallel composition*:[9]

$$\overset{\text{PAR}}{\frac{\{P_1\}\,e_1\,\{Q_1\} \qquad \{P_2\}\,e_2\,\{Q_2\}}{\{P_1 * P_2\}\,e_1 \mid\mid e_2\,\{Q_1 * Q_2\}}}$$

This rule says that to verify a parallel composition $e_1 \mid\mid e_2$ executing two threads concurrently, we can verify both threads individually as long as we *separately* own their preconditions. We then separately own their postconditions when both threads are done.

This rule is beautiful! It has to be seen in contrast to the kind of global conditions that are common in the line of work descending from Owicki-Gries's seminal first program logic for concurrent programs, where *every statement* made anywhere in the proof (including all intermediate pre- and postconditions) must be checked for interference with other threads. In CSL, because assertions are stable under interference, this all gets replaced by a single separating conjunction. In some sense, we can even see this rule as a generalization of the frame rule: if e_2 is a thread

[7] The original frame rule had some extra side-conditions concerning free variables. We will come back to this in §7.3 when explaining why they are not needed in Iris.

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

[9] Again, we are omitting side-conditions concerning free variables.

that does nothing (represented below by the unit value, ()), its pre- and postcondition become the same, so let us call it R. We can specialize PAR as follows:

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{R\}\, ()\, \{R\}}{\{P_1 * R\}\, e_1 \mid\mid ()\, \{Q_1 * R\}}$$

As we can see, framing is basically parallel composition with a thread that does nothing.

Resource invariants. The rule for parallel composition is sufficient to verify programs where concurrent threads only act on disjoint state. However, most concurrent programs at some point have to *communicate*, which requires sharing some state (be it via a mutable heap or message-passing channels). This entails some form of interference between the communicating threads. To reason about interference in a local way, CSL introduced a simple form of *resource invariants* tied to "conditional critical regions".[10] Each region is associated with a resource invariant (expressed as a separation logic formula), indicating what it is that this critical region "guards" or "protects". The proof rule for critical region r grants the program access to that resource invariant I_r for the duration of the critical region:[11]

$$\textsc{critical-region}\quad \frac{\{P * I_r\}\, e\, \{Q * I_r\}}{\{P\}\, \texttt{with}\ r\ \texttt{do}\ e\ \texttt{endwith}\, \{Q\}}$$

This rule says that I_r is added to the resources available to this thread for the duration of the critical region r (compare the two preconditions), but when the critical region ends, the process must again return ownership of I_r and said ownership is taken away (compare the two postconditions). This very elegantly expresses the idea that the thread *temporarily* has exclusive access to the resources guarded by the critical region.

Resource invariants are a very simple form of *protocol* describing how multiple threads interact on a piece of shared state. The ingenuity of the approach lies in the fact that once the resource invariant is determined, each thread has to only consider said invariant and its own actions; the actions of the other threads are abstracted away by the invariant. This means that verification can proceed in a modular fashion.

2.3 Extensions of CSL

O'Hearn showed that, using only the standard rules of separation logic plus PAR and CRITICAL-REGION, one could elegantly verify some rather "daring" synchronization patterns in which ownership of shared resources is transferred subtly between threads in a way that is not syntactically evident from the program text.

It is thus unsurprising that the vast majority of the prior work cited above descends more or less directly from CSL. The way in which that work innovates is mostly along two axes: generalizing the heap to other kinds of "resources" that can be owned, and generalizing invariants to other kinds of "protocols" that are expressible in the logic.

[10] *Resource* here is a generalization of "heap", referring to whatever it is that can be *owned* by some code and split disjointly by the separating conjuction.

[11] This rule is simplified to only apply to non-conditional critical regions, and again side-conditions regarding free variables are omitted.

More resources. The next key step in separation logic was the introduction of "fictional separation":[12] even though two threads may be operating on the same piece of the physical state, we can conceptually view them as operating on disjoint *logical* parts of that state.

The simplest example of fictional separation are "fractional permissions":[13] the points-to assertion becomes $\ell \overset{q}{\mapsto} v$, which expresses ownership of the fraction q of location ℓ. Separating conjunction splits the fraction into pieces:

$$\ell \xmapsto{q_1+q_2} v \iff \ell \xmapsto{q_1} v * \ell \xmapsto{q_2} v$$

Full ownership ($q = 1$) is required to *write* to ℓ, but *reading* can be done by any thread owning some fraction of the location. This permits verification of code that performs read-only sharing of some part of the memory: as long as no thread writes to the shared location, no synchronization[14] is necessary. At the same time, keeping track of the exact fraction even for partial ownership allows *re-combining* the permissions later so that the full permission can be recovered when all threads are done.

The culmination of this development are "user-defined resources", where the user of a logic gets to pick an arbitrary instance of some algebra which they can then use as their notion of "resource" for their proofs. Particularly noteworthy for the development of Iris is the *Views Framework*:[15] a general recipe for how to define a separation logic given some "separation algebra" as a notion of resources, and given some relation between these resources and the physical state of the program. Through this relation, resources are viewed as "views" onto the physical state. The key innovation of the Views Framework is its *action judgment* that defines when a view may be "updated" from one resource to another. This idea of "updating" resources makes them behave a lot like a piece of state that can be manipulated in the logic but does not have to exist in the real program. And indeed these user-defined resources can take the role of auxiliary variables[16] in Hoare logic, also sometimes called "ghost state".

The Views Framework requires fixing the notion of resources before defining the logic. *Fictional Separation Logic*[17] demonstrates an approach for "dynamically" adding resources during the proof.[18]

More protocols. However, adding more and more flexibility to resources (extending what can be *owned* in a separation logic) was not sufficient. Plain resource invariants were unable to describe the intricate interactions arising in complex concurrent data structures, so new logics also introduced more powerful "protocol" mechanisms to describe those interactions. Usually, this involved some form of "states" and "transitions" with "guards" on the transitions to encode asymmetry (where a transition can be taken by some but not all threads).[19] Initially, states were just invariants and transitions could move from one invariant to another;[20] later work introduced a notion of an "abstract state" that a protocol can be in,[21] a state-transition system on those abstract states, and a state interpretation defining the invariant that has to hold in each state.[22]

Unlike for the case of resources, no consolidation of these protocol mechanisms was in sight. As a reaction to this situation, Matthew Parkinson wrote in his position paper *The Next 700 Separation Logics*:[23]

[12] Dinsdale-Young, Gardner, and Wheelhouse, "Abstraction and refinement for local reasoning", 2010 [DGW10]; Dinsdale-Young *et al.*, "Concurrent abstract predicates", 2010 [Din+10].

[13] Boyland, "Checking interference with fractional permissions", 2003 [Boy03]; Bornat *et al.*, "Permission accounting in separation logic", 2005 [Bor+05].

[14] *e.g.*, through a critical region

[15] Dinsdale-Young *et al.*, "Views: Compositional reasoning for concurrent programs", 2013 [Din+13].

[16] Jones, "The role of auxiliary variables in the formal development of concurrent programs", 2010 [Jon10]; Kleymann, "Hoare logic and auxiliary variables", 1999 [Kle99].

[17] Jensen and Birkedal, "Fictional separation logic", 2012 [JB12].

[18] We will see in §4.3 how we modularly handle defining the notion of resources in Iris.

[19] These guards form another kind of resource, but are intrinsically linked to the protocol mechanism.

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

[21] 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].

[22] The difference between the newer "abstract state" approach and the prior one can be compared to the difference between a tagged and an untagged union: the current abstract state is maintained as an *explicit* piece of information, instead of permitting any transition that happens to match the resources currently justifying the invariant.

[23] Parkinson, "The next 700 separation logics - (Invited paper)", 2010 [Par10].

> In recent years, separation logic has brought great advances in the world of verification. However, there is a disturbing trend for each new library or concurrency primitive to require a new separation logic.

Parkinson argued that what is needed is a general logic for concurrent reasoning, into which a variety of useful specifications can be encoded via the abstraction facilities of the logic. "By finding the right core logic," he wrote, "we can concentrate on the difficult problems." We are proposing Iris as that core logic.

2.4 Iris

The new key idea that Iris brings to the story of concurrent separation logics is that *sufficiently flexible ownership can encode protocols.* Instead of coming up with yet another novel protocol mechanism that somehow encompasses all of the previous ones, Iris rests on the realization that arbitrary user-defined resources combined with simple resource invariants in the style of original CSL are powerful enough already. Since resources are little more than arbitrary partial commutative monoids, the slogan for the initial version of Iris[24] (later dubbed "Iris 1.0") was: *Monoids and invariants are all you need.*

However, during further development, it turned out that this slogan was inadequate in two ways:

- *Plain monoids are not enough.* There are some useful kinds of ghost state, such as "saved propositions" (see §6), for which the structure of the ghost state and the propositions of the separation logic must be defined in a mutually recursive way. We refer to this as *higher-order ghost state*, and for encoding higher-order ghost state, it seems we need something more sophisticated than monoids.

- *Iris is not just monoids & invariants.* Although monoids and invariants did indeed constitute the two main conceptual elements of Iris 1.0—and they are arguably "canonical" in their simplicity and universality—some of the mechanisms that embed these concepts into the logic were more ad-hoc. For example, Iris 1.0 had built-in support for creating many instances of the same kind of ghost state, and invariants required a notion of "namespaces" to ensure soundness. Iris 1.0 also came with primitives such as *mask-changing view shifts* (for performing logical updates to resources that may involve "opening" or "closing" invariants) and *weakest preconditions* (for encoding Hoare triples). Moreover, the proof rules for these mechanisms were non-standard, and their semantic model quite involved, making the justification of the primitive rules—not to mention the very *meaning* of Iris's Hoare-style program specifications—very difficult to understand or explain. Indeed, the Iris 1.0 paper avoided presenting the formal model of program specifications.

In subsequent work on "Iris 2.0"[25] and "Iris 3.0"[26] we have addressed these two points:

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

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

[26] Krebbers *et al.*, "The essence of higher-order concurrent separation logic", 2017 [Kre+17].

- In Iris 2.0, to support higher-order ghost state, we proposed a generalization of partial commutative monoids (PCMs) to what we call *cameras*.[27] Roughly speaking, a camera is a kind of "step-indexed PCM", *i.e.*, a PCM equipped with a step-indexed notion of equality[28] such that the composition operator of the PCM is sufficiently "compatible" with the step-indexed equality. Step-indexing has always been essential to how Iris models the propositions of higher-order separation logic, and by incorporating step-indexed equality into PCMs, cameras enable us to model ghost state that can embed propositions.

- In Iris 3.0, we simplified the remaining sources of complexity in Iris by taking the Iris story to its "logical" conclusion: applying the reductionist Iris methodology to Iris itself! Specifically, at the core of Iris 3.0 is a small, resourceful *base logic* which distills the essence of Iris to a bare minimum: a higher-order logic extended with the basic connectives of bunched implications[29] (separating conjunction and magic wand), a predicate for resource ownership, and a handful of simple modalities. We do not bake in any propositions about programs or invariants. It is only this base logic whose soundness must be proven directly against an underlying semantic model (employing the aforementioned cameras). Moreover, using the mechanisms provided by the base logic, Iris 1.0's fancier mechanisms of mask-changing view shifts and weakest preconditions, and even invariants themselves—together with their associated proof rules—*can all be derived within the logic.*

In other words, the key to unifying all protocol mechanisms in Iris turned out to be the complete removal of dedicated protocol mechanisms from the logic, relying entirely on ownership of resources. It turns out that *fancy monoids are all you need.*

However, for the purpose of this dissertation, it is not helpful to go all the way down to the Iris base logic. We refer the interested reader to "Iris from the ground up",[30] which explains in great detail how to construct a program logic with support for invariants using the base logic, and how to justify the soundness of the Iris base logic in the first place. In the following, we will instead stay on the level of said program logic, and focus more on constructing higher-level reasoning principles from monoids and invariants.

[27] Cameras were originally called CMRAs (but already pronounced "cameras"), for "Complete Metric space Resource Algebras". That name stopped working when we realized we could drop the completeness requirement.

[28] Appel and McAllester, "An indexed model of recursive types for foundational proof-carrying code", 2001 [AM01]; Birkedal *et al.*, "First steps in synthetic guarded domain theory: Step-indexing in the topos of trees", 2011 [Bir+11].

[29] O'Hearn and Pym, "The logic of bunched implications", 1999 [OP99]; Ishtiaq and O'Hearn, "BI as an assertion language for mutable data structures", 2001 [IO01].

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

Chapter 3
An introduction to Iris

In this chapter, we introduce Iris by verifying a small example program, motivating and demonstrating the use of its most important features.

Iris is a generic higher-order concurrent separation logic.[1] "Generic" refers to the fact that the logic is parameterized by the language of program expressions that one wishes to reason about, so the same logic can be used for a wide variety of languages. Before we start with the example, we briefly introduce our idealized language and the grammar of the logic.

Language. For the purpose of this dissertation we instantiate Iris with HL (HeapLang): an ML-like language with higher-order store, fork, and compare-exchange (**CmpX**), as given below:

$$\begin{aligned}
v, w \in \mathit{Val} &::= () \mid z \mid \mathtt{true} \mid \mathtt{false} \mid \ell \mid \lambda x.e \mid && (\ell, z \in \mathbb{Z}) \\
&\quad\; (v, w) \mid \mathtt{inl}(v) \mid \mathtt{inr}(v) \\
e \in \mathit{Expr} &::= v \mid x \mid e_1(e_2) \mid \mathtt{fork}\ \{e\} \mid \mathtt{assert}(e) \mid \\
&\quad\; \mathtt{ref}(e) \mid {!}\,e \mid e_1 \leftarrow e_2 \mid \mathtt{CmpX}(e, e_1, e_2) \mid \dots \\
K \in \mathit{Ctx} &::= \bullet \mid K(e) \mid v(K) \mid \mathtt{assert}(K) \mid \\
&\quad\; \mathtt{ref}(K) \mid {!}\,K \mid K \leftarrow e \mid v \leftarrow K \mid \\
&\quad\; \mathtt{CmpX}(K, e_1, e_2) \mid \mathtt{CmpX}(v, K, e_2) \mid \mathtt{CmpX}(v, v_1, K) \mid \dots
\end{aligned}$$

We omit the usual projections on pairs and pattern matching on sums as well as primitive operations on values (comparison, addition, ...).

The operational semantics are defined in Figure 3.1. The state of a program execution is given by a *heap* σ mapping locations ℓ to values v, and a *thread pool* storing the expression each thread is executing. *Thread-pool reduction* $(T_1, \sigma_1) \rightarrow_{\mathsf{tp}} (T_2, \sigma_2)$ just picks some thread nondeterministically[2] and takes a step in that thread. *Thread reduction* uses evaluation contexts K to find a reducible *head expression*[3] and reduces it. *Head reduction* $(e_1, \sigma_1) \rightarrow_{\mathsf{h}} (e_2, \sigma_2, \vec{e}_f)$ says that head expression e_1 with current heap σ_1 can step to e_2, change the heap to σ_2 and spawn new threads $\vec{e}_f$ (this is a *list* of expressions) in the process. We use "stuckness" (irreducible expressions) to model bogus executions, like a program that tries to use a Boolean (or an integer) to access memory, or a program that runs into `assert(false)`.

The steps our programs can take are mostly standard, except for $\mathtt{CmpX}(\ell, w_1, w_2)$: this *compare-exchange* instruction *atomically* compares

[1] "Separation logic" here implies that it also is a program logic—though Iris is used not just for program verification, *e.g.*, in §10 we will see how to define a logical relation with Iris.

[2] This is demonic non-determinism, so we abstract over all possible schedulers.

[3] There can be several head expressions, but in our language only one of them is reducible. In other words, evaluation order is deterministic.

Head reduction.

$$
\begin{aligned}
((\lambda x.e)(v), \sigma) &\to_{\mathsf{h}} (e[v/x], \sigma, \epsilon) && \\
(\mathtt{fork}\,\{e\}\,, \sigma) &\to_{\mathsf{h}} ((), \sigma, e) && \\
(\mathtt{assert}(\mathtt{true}), \sigma) &\to_{\mathsf{h}} ((), \sigma, \epsilon) && \\
(\mathtt{ref}(v), \sigma) &\to_{\mathsf{h}} (\ell, \sigma[\ell \leftarrow v]\,, \epsilon) && \text{if } \sigma(\ell) = \bot \\
(!\,\ell, \sigma[\ell \leftarrow v]) &\to_{\mathsf{h}} (v, \sigma[\ell \leftarrow v]\,, \epsilon) && \\
(\ell \leftarrow w, \sigma[\ell \leftarrow v]) &\to_{\mathsf{h}} ((), \sigma[\ell \leftarrow w]\,, \epsilon) && \\
(\mathtt{CmpX}(\ell, w_1, w_2), \sigma[\ell \leftarrow v]) &\to_{\mathsf{h}} ((v, \mathtt{true}), \sigma[\ell \leftarrow w_2]\,, \epsilon) && \text{if } v \cong w_1 \\
(\mathtt{CmpX}(\ell, w_1, w_2), \sigma[\ell \leftarrow v]) &\to_{\mathsf{h}} ((v, \mathtt{false}), \sigma[\ell \leftarrow v]\,, \epsilon) && \text{if } v \ncong w_1
\end{aligned}
$$

Thread-local and threadpool reduction.

$$
\frac{(e, \sigma) \to_{\mathsf{h}} (e', \sigma', \vec{e}_f)}{(K[\,e\,], \sigma) \to_{\mathsf{t}} (K[\,e'\,], \sigma', \vec{e}_f)}
\qquad
\frac{(e, \sigma) \to_{\mathsf{t}} (e', \sigma', \vec{e}_f)}{(T_1; e; T_2, \sigma) \to_{\mathsf{tp}} (T_1; e'; T_2; \vec{e}_f, \sigma')}
$$

Figure 3.1: Operational semantics.

the value stored in location ℓ with w_1, and if the two are equal, stores w_2 in ℓ; if they are unequal, the value stored in ℓ remains unchanged. It returns a pair of the old value stored in ℓ and a Boolean indicating whether the comparison was successful.[4]

Our comparison operator $\cong$ is non-standard in that it is *partial*: not all values can be compared. For example, we do not allow comparing closures with each other, and we do not allow comparing compound types such as pairs, because both of these operations cannot be implemented on real hardware (either atomically, or at all). In general, comparison is only permitted if at least one operand is "unboxed", which means it is a literal ($()$, z, ℓ, **`true`**, or **`false`**) or **`inl`**/**`inr`** of a literal.[5] We use $v \cong w_q$ to say that the two values can be compared and are equal, and $v \ncong w_1$ to say that they can be compared and are unequal. If v and w_1 cannot be compared, **`CmpX`** is stuck. (The same comparison judgments are also used for the binary $=$ operator.)

Logic. The logic of Iris includes the usual connectives and rules of higher-order separation logic, some of which are shown in the grammar below.[6]

$$
\begin{aligned}
P, Q, R ::= {} & \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall x.\, P \mid \exists x.\, P \mid \\
& P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto v \mid t = u \mid \\
& \Box\, P \mid \boxed{P}^{\mathcal{N}} \mid \boxed{a}^{\gamma} \mid \mathcal{V}(a) \mid \{P\}\, e\, \{v.\, Q\}_{\mathcal{E}} \mid P \Rrightarrow_{\mathcal{E}} Q \mid \ldots
\end{aligned}
$$

We will talk in more detail about the non-standard connectives as they come up.

We show some of the proof rules for Hoare triples in Figure 3.2. Since HL is an expression-based language (as opposed to a statement-based language), Hoare triples need to be able to talk about the *return value* of the expression. This is made possible in Iris by making the postcondition

[4] This matches exactly the `AtomicCmpXchgInst` in LLVM. Not that HL is modeled after LLVM, but we wanted to make sure the semantics are implementable and useful.

[5] We call these values "unboxed" because in a hypothetical implementation of HeapLang, we could represent them all in a machine word and represent all larger compound values as pointers to a "boxed" representation on the heap.

[6] Actually, many of the connectives given in this grammar are not primitive to Iris. They are defined as *derived forms*, and this flexibility is an important aspect of the logic. We will see a glimpse of that in §4 and §5; for further details, see "Iris from the ground up: A modular foundation for higher-order concurrent separation logic" [Jun+18b].

HOARE-FRAME

$$\frac{\{P\}\, e\, \{w.\, Q\}_{\mathcal{E}}}{\{P * R\}\, e\, \{w.\, Q * R\}_{\mathcal{E}}}$$

HOARE-VALUE

$$\{\mathsf{True}\}\, v\, \{w.\, w = v\}_{\mathcal{E}}$$

HOARE-BIND

$$\frac{\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}} \qquad \forall v.\, \{Q\}\, K[v]\, \{w.\, R\}_{\mathcal{E}}}{\{P\}\, K[e]\, \{w.\, R\}_{\mathcal{E}}}$$

HOARE-λ

$$\frac{\{P\}\, e[v/x]\, \{w.\, Q\}_{\mathcal{E}}}{\{P\}\, (\lambda x.e)(v)\, \{w.\, Q\}_{\mathcal{E}}}$$

HOARE-FORK

$$\frac{\{P\}\, e\, \{\mathsf{True}\}}{\{P\}\, \mathtt{fork}\ \{e\}\ \{\mathsf{True}\}_{\mathcal{E}}}$$

HOARE-ASSERT

$$\{\mathsf{True}\}\, \mathtt{assert}(\mathtt{true})\, \{\mathsf{True}\}_{\mathcal{E}}$$

HOARE-ALLOC

$$\{\mathsf{True}\}\, \mathtt{ref}(v)\, \{\ell.\, \ell \mapsto v\}_{\mathcal{E}}$$

HOARE-LOAD

$$\{\ell \mapsto v\}\ !\,\ell\, \{w.\, w = v * \ell \mapsto v\}_{\mathcal{E}}$$

HOARE-STORE

$$\{\ell \mapsto v\}\, \ell \leftarrow w\, \{\ell \mapsto w\}_{\mathcal{E}}$$

HOARE-CMPX-SUC

$$\frac{v \cong w_1}{\{\ell \mapsto v\}\, \mathtt{CmpX}(\ell, w_1, w_2)\, \{v'.\, v' = (v, \mathtt{true}) * \ell \mapsto w_2\}_{\mathcal{E}}}$$

HOARE-CMPX-FAIL

$$\frac{v \ncong w_1}{\{\ell \mapsto v\}\, \mathtt{CmpX}(\ell, w_1, w_2)\, \{v'.\, v' = (v, \mathtt{false}) * \ell \mapsto v\}_{\mathcal{E}}}$$

HEAP-EXCLUSIVE

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

Figure 3.2: Basic Iris rules for Hoare triples.

a *predicate* on the return value of the expression e, with an explicit binder (usually v).[7] We sometimes (ab)use that binder as a (refutable) pattern, implying that the return value must match the pattern (*e.g.*, in HOARE-ALLOC, where the return value must be a location). When we omit it, it means the return value is unconstrained (like in HOARE-FORK).

Another consequence of the expression-based nature of HL is that the usual *sequence rule* to chain statements is generalized to the *bind rule* (HOARE-BIND): when some expression e is plugged into an evaluation context K (*i.e.*, it is in evaluation position), we can first verify e and then use its result v to verify the evaluation of the context with that result ($K[v]$).

We also have the usual frame rule that was mentioned in the previous chapter (HOARE-FRAME)—without any side-conditions (see §7.3). HOARE-λ reflects that function application is subject to β-reduction. We have similar rules for each head reduction step that does not interact with the heap. The rules for heap accesses are standard for separation logics, except HOARE-CMPX-SUC and HOARE-CMPX-FAIL, which reflect the operational semantics of **`CmpX`**.

The reader might wonder what happened to the rule for parallel composition, or indeed what happened to the parallel composition operator itself? Instead of parallel composition, which executes two expressions in parallel and waits until both are done, HL has **`fork`**, which spawns of a new thread that just keeps running asynchronously in the background. **`fork`** provides *unstructured* parallelism which is strictly more powerful than the *structured* parallelism afforded by parallel composition.[8] This corresponds to the usual concurrency primitive of languages like Java, C++ or Rust, which also offer unstructured parallelism. The proof rule

[7] The more common approach to talk about the return value in a postcondition is to use a magic identifier like `ret` for this purpose. We use a binder to be more true to the mechanization in Coq, and because it avoids shadowing in case of nested Hoare triples.

[8] Another term sometimes used in this context is “fork-join parallelism”. Confusingly, this typically (but not always) refers to *structured* parallelism with parallel composition, which can be viewed as a single *fork-join* operation. With unstructured parallelism, *fork* and *join* are two independently usable operations. In HL, *join* can be implemented inside the language, so we did not make it a primitive operation.

for **`fork`** (HOARE-FORK) says that any resources P owned by the current thread may be used to verify the forked-off thread. If that seems confusing, consider this rule composed with framing:

$$\frac{\{P\}\, e\, \{\mathsf{True}\}}{\{P * Q\}\, \texttt{fork}\ \{e\}\ \{Q\}_{\mathcal{E}}}$$

This already looks much more like PAR from the introduction: the resources of the current thread can be split into P (handed off to the forked-off thread) and Q (kept by the current thread). Unlike parallel composition, there is no built-in mechanism to get resources back from the new thread, which corresponds to there not being a built-in way to wait for the other thread to finish (to "join" the other thread). Such a mechanism can be implemented inside HL and verified in Iris, so it is not provided as a primitive.[9]

What makes Iris a *higher-order* separation logic is that universal and existential quantifiers can range over any type, including that of propositions themselves and even (higher-order) predicates. Furthermore, notice that Hoare triples $\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}}$ are part of the proposition logic (also often called "assertion logic") instead of being a separate entity (the "program logic" is entirely separated in CSL and many of its descendants). As a consequence, triples can be used in the same way as any logical proposition, and in particular, they can be *nested* to give specifications of higher-order functions. Hoare triples $\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}}$ are moreover annotated with a *mask* $\mathcal{E}$ to keep track of which invariants are currently in force. We will come back to invariants and masks in §3.2, but for the time being we omit them.

A motivating example. We will demonstrate the higher-order aspects of Iris, and some other of its core features, by verifying the safety of the higher-order program given in Figure 3.3. This example is of course rather contrived, but it serves to showcase the core features of Iris. Notice that we are using `typewriter` font for language-level terms, sans-serif font for logic-level terms, and *italic* font for domains (corresponding to types in our Coq formalization).[10]

The function `mk_oneshot`() allocates a oneshot ("write-once") location at x and returns a record c with two closures. (Formally, records are syntactic sugar for pairs, and let-binding and sequencing are defined in terms of immediately applied λ-terms.) The function c.`set`(n) sets the location x to n, asserting that this has not happened before. We use **`CmpX`** to ensure correctness of the check even if two threads concurrently try to set the location. The function c.`check`() records the current state of the location x and then returns a closure which, if the location x has already been initialized, asserts that it does not change.

Intuitively, the assertion in c.`check` can never fail because x is *hidden* from the client. It can only be mutated by calling c.`set`, which will not do anything if x already contains an **`inr`**. In contrast, the assertion in c.`set` *can* fail, the client just has to call the function twice. The specification for the example is thus going to express that c.`set` must only be called once, but c.`check` can be called arbitrarily often.[11]

[9] For the curious, this is implemented in the Iris Coq repository in `theories/heap_lang/lib/spawn.v`. Parallel composition is implemented in `theories/heap_lang/lib/par.v`.

[10] We do not distinguish language-level and logic-level *variables* in their font. This choice is justified in §7.3.

[11] This goes to show that we could in fact replace the **`CmpX`** by a normal store. We use **`CmpX`** with an assertion on the old value to ensure that any safety specification for c.`set` has to enforce this "one-shot" property.

Example code.

```
mk_oneshot := λ_.
          let x = ref(inl(0));
          {  set = λn. let (_, b) = CmpX(x, inl(0), inr(n));
                       assert(b),
           check = λ_. let y = ! x;
                      λ_. let y′ = ! x;
                         match y with
                           inl(_) ⇒ ()
                         | inr(_) ⇒ assert(y = y′)
                         end   }
```

Example specification.

$$
\begin{array}{l}
\{\mathsf{True}\} \\
\quad \texttt{mk\_oneshot}() \\
\left\{\begin{array}{l} c.\, \exists T.\, T * \big(\forall v.\, \{T\}\, c.\texttt{set}(v)\, \{\mathsf{True}\}\big) * \\ \quad \{\mathsf{True}\}\, c.\texttt{check}()\, \{f.\, \{\mathsf{True}\}\, f()\, \{\mathsf{True}\}\} \end{array}\right\}
\end{array}
$$

Figure 3.3: Example code and specification.

Specifying the example. The specification in Figure 3.3 looks a little funny with most pre- and postconditions being True. The reason for this is that all we are aiming to show here is that the code is *safe*, *i.e.*, that the assertions "succeed". Recall that in our semantics[12], **assert**(e) gets stuck if e evaluates to **false**. Correspondingly, we have the rule HOARE-ASSERT for **assert**(**true**), but no corresponding rule for **assert**(**false**). In Iris, Hoare triples imply safety,[13] so we do not need to impose any further conditions: just showing *any* triple will mean that, if the precondition of the triple is satisfied, all assertions will pass.

We use nested Hoare triples to express that mk_oneshot() returns closures: Since Hoare triples are just propositions, we can put them into the postcondition of mk_oneshot() to describe what the client can assume about c. The postcondition also has to express that c.set(n) can only be called once. We do this here by saying there there exists *some* proposition T (for "token") that represents the *right to call* c.set(n). We initially hand out that right as part of the postcondition of mk_oneshot(), and then we consume that right in the precondition of c.set(n). Calling c.set(n) twice would thus require giving up ownership of T twice, but propositions in a separation logic can, in general, not be duplicated ($P \Rightarrow P * P$ does *not* hold)—so T can only be used once. Except for this restriction, since Iris is a *concurrent* program logic, the specification for mk_oneshot() actually permits clients to call c.set(n) and c.check(), as well as the closure f returned by c.check(), concurrently from multiple threads and in any combination.

It is worth pointing out that Iris is an *affine* separation logic, which means that it enjoys the weakening rule $P * Q \Rightarrow P$.[14] Intuitively, this

[12] Figure 3.1 on page 26

[13] This is to say, they imply that the program will not get stuck. However, it is also possible to use Iris with a variant of Hoare triples that permits programs getting stuck.

[14] This is also sometimes called *intuitionistic* separation logic, in contrast to *classical* separation logic which does not support weakening. However, when applied to Iris, this terminology does not make much sense. See the end of §6.2 for further discussion.

rule lets one *throw away* resources; for example, in the postcondition of a Hoare triple one can throw away ownership of unused memory locations. Since Iris is affine, it does not have the Emp connective, which asserts ownership of no resources;[15] rather, Iris's True connective, which describes ownership of *any* resources, is the identity of separating conjunction (*i.e.*, $P * \mathsf{True} \Leftrightarrow P$). We discuss this design choice further in §6.2.

High-level proof structure. To perform this proof, we need to somehow encode the fact that we are only performing a *oneshot* update to x. To this end, we will allocate a ghost location $\boxed{a}^{\gamma}$ with name γ and value a, which essentially mirrors the current state of x. This may at first sound rather pointless; why should we record a value in the ghost state that is exactly the same as the value in a particular physical location?

The point is that using ghost state lets us choose what kind of *sharing* is possible on the location. For a physical location ℓ, the proposition $\ell \mapsto v$ expresses full ownership of ℓ (and hence the absence of any sharing of it). In contrast, Iris permits us to choose whatever kind of structure and ownership we want for our ghost location γ; in particular, we can define it in such a way that, although the contents of γ mirror the contents of x, we can freely share ownership of γ once it has been initialized (by a call to `set`). This in turn will allow the closure returned by `check` to own a piece of γ witnessing its value after initialization. We will then have an *invariant* (see §3.2) tying the value of γ to the value of x, so we *know* which value that closure is going to see when it reads from x, and we know that that value is going to match y.

Another way to describe what is happening is to say that we are applying the idea of *fictional separation*:[16] The separation on γ (after `set` was called) is "fictional" in the sense that multiple threads can own parts of γ and therefore manipulate the same shared variable x, the two being tied together by an invariant.

With this high-level proof structure in mind, we now explain how exactly ownership and sharing of ghost state can be controlled.

3.1 Ghost state in Iris: Resource algebras

Iris allows one to use ghost state via the proposition $\boxed{a}^{\gamma}$, which asserts ownership of a piece a of a ghost location γ. The flexibility of Iris stems from the fact that for each ghost location γ, the user of the logic can pick the type M of its value a, instead of this type being fixed in advance by the logic. However, M cannot just be any type, it must have enough structure to be suited as a notion of "ownership" and to make $\boxed{a}^{\gamma}$ useful, namely:

- It should be possible to compose ownership of different threads. To this end, the type M should have an operator $(\cdot)$ for *composition.* The crucial rule for this operation in the logic is $\boxed{a \cdot b}^{\gamma} \Leftrightarrow \boxed{a}^{\gamma} * \boxed{b}^{\gamma}$ (as we will see with ghost-op in Figure 3.5).
- Combinations of ownership $\boxed{a}^{\gamma} * \boxed{b}^{\gamma}$ that do not make sense should be ruled out by the logic (*i.e.*, they should entail False). This happens,

[15] O'Hearn, Reynolds, and Yang, "Local reasoning about programs that alter data structures", 2001 [ORY01].

[16] Dinsdale-Young, Gardner, and Wheelhouse, "Abstraction and refinement for local reasoning", 2010 [DGW10]; Dinsdale-Young *et al.*, "Concurrent abstract predicates", 2010 [Din+10].

for example, when multiple threads claim to have ownership of an exclusive resource. To make this possible, the operator $(\cdot)$ should be partial.

Composition of ownership should moreover be *associative and commutative*, to reflect the associative and commutative nature of separating conjunction.[17] For that reason, *partial commutative monoids* (PCMs) have become the canonical structure for representing ghost state in separation logics. In Iris, we are deviating slightly from this, using our own notion of a *resource algebra* (RA), whose definition is in Figure 3.4. Every PCM is an RA, but not vice versa—and as we will see in our example, the additional flexibility afforded by RAs results in additional logical expressiveness.

A *resource algebra* (RA) is a tuple $(M, \mathcal{V} : M \to \mathit{Prop}, |{-}| : M \to M^?, (\cdot) : M \times M \to M)$ satisfying:

$$
\begin{aligned}
&\forall a, b, c.\; (a \cdot b) \cdot c = a \cdot (b \cdot c) && (\textsc{ra-assoc})\\
&\forall a, b.\; a \cdot b = b \cdot a && (\textsc{ra-comm})\\
&\forall a.\; |a| \in M \Rightarrow |a| \cdot a = a && (\textsc{ra-core-id})\\
&\forall a.\; |a| \in M \Rightarrow ||a|| = |a| && (\textsc{ra-core-idem})\\
&\forall a, b.\; |a| \in M \land a \preccurlyeq b \Rightarrow |b| \in M \land |a| \preccurlyeq |b| && (\textsc{ra-core-mono})\\
&\forall a, b.\; \mathcal{V}(a \cdot b) \Rightarrow \mathcal{V}(a) && (\textsc{ra-valid-op})\\
\text{where}\quad & M^? := M \uplus \{\bot\} \qquad \text{with} \qquad a^? \cdot \bot := \bot \cdot a^? := a^? &&\\
& a \preccurlyeq b := \exists c \in M.\; b = a \cdot c && (\textsc{ra-incl})\\
& a \rightsquigarrow B := \forall c^? \in M^?.\; \mathcal{V}(a \cdot c^?) \Rightarrow \exists b \in B.\; \mathcal{V}(b \cdot c^?) &&\\
& a \rightsquigarrow b := a \rightsquigarrow \{b\} &&
\end{aligned}
$$

A *unital resource algebra* (uRA) is a resource algebra M with an element ε satisfying:

$$
\mathcal{V}(\varepsilon) \qquad\qquad \forall a \in M.\; \varepsilon \cdot a = a \qquad\qquad |\varepsilon| = \varepsilon
$$

Figure 3.4: Resource algebras.

There are two key differences between RAs and PCMs:

1. Instead of partiality, RAs use *validity* to rule out invalid combinations of ownership. Specifically, there is a predicate $\mathcal{V} : M \to \mathit{Prop}$ identifying *valid* elements. Validity is compatible with the composition operation (ra-valid-op).

   This take on partiality is necessary when defining the structure of *higher-order ghost state*,[18] *i.e.*, ghost state whose structure depends on *iProp*, the type of propositions of the Iris logic.[19] Making composition $(\cdot)$ a total operation is also a good choice for mechanization in Coq, where partial functions have to be handled fully explicitly (working with `option` everywhere), which is rather painful.[20]

2. Instead of having a single unit ε that is an identity to *every* element (*i.e.*, that enjoys $\varepsilon \cdot a = a$ for any a), RAs have a partial function $|{-}|$ that assigns to an element a its *(duplicable) core* $|a|$, as demanded by ra-core-id. We further demand that $|{-}|$ is idempotent (ra-core-idem) and monotone (ra-core-mono) with respect to the *extension order*, defined similarly to that for PCMs (ra-incl).

[17] This also reflects the associativity and commutativity of parallel composition.

[18] Jung *et al.*, "Higher-order ghost state", 2016 [Jun+16]; Jung *et al.*, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b].

[19] Note the difference between *Prop*, which denotes the type of propositions of the meta-logic (*e.g.*, Coq), and *iProp*, which denotes the type of propositions of Iris.

[20] This is also why we did not follow the alternative approach of making composition a 3-place relation instead of a function (see §7.2 for a more in-depth comparison).

An element can have no core, which is indicated by $|a| = \bot$. In order to conveniently deal with partial cores, we use the metavariable $a^?$ to range over elements of $M^? := M \uplus \{\bot\}$ and lift the composition $(\cdot)$ to $M^?$. As we will see in §4.1, partial cores help us to build interesting composite RAs from smaller primitives.

In the special case that an RA *does* have a unit ε such that $\varepsilon \cdot a = a$ and moreover $|\varepsilon| = \varepsilon$, we call it a *unital* RA (uRA). From ra-core-mono, it follows that the core of a uRA is a total function, *i.e.*, $|a| \neq \bot$.

Resource algebras are additionally equipped with an "inclusion relation": $a \preccurlyeq b$ says that b is a "bigger" resource, *i.e.*, that it is obtained by composing a with some other resource c. Note that inclusion is in general *not* reflexive, but it is reflexive for unital RAs.[21] In the context of PCMs, this relation is also often called "extension order".

We also define a notion of *frame-preserving updates* $a \rightsquigarrow B$, which we will explain after introducing our example RA.

A resource algebra for our example. We will now define the RA that can be used to verify the example, which we call the *oneshot* RA.[22] The goal of this RA is to appropriately reflect the state of the physical location x. The carrier is defined using a datatype-like notation as follows:[23]

$$M := \mathsf{pending}(q : \mathbb{Q} \cap (0, 1]) \mid \mathsf{shot}(n : \mathbb{Z}) \mid \lightning$$

The two important states of the ghost location are: $\mathsf{pending}(q)$, to represent the fact that the single update has not yet happened, and $\mathsf{shot}(n)$, saying that the location has been set to n. The fraction q here (restricted to be strictly positive and not exceed 1) provides the ability to *split* ownership of $\mathsf{pending}(\_)$. This works similar to fractional permissions for the points-to assertion (discussed in §2.3).

We need an additional element $\lightning$ to account for partiality; it is the only invalid element:

$$\mathcal{V}(a) := a \neq \lightning$$

The most interesting part of an RA is, of course, its composition: What happens when ownership of two threads is combined?[24]

$$\mathsf{pending}(q_1) \cdot \mathsf{pending}(q_2) := \begin{cases} \mathsf{pending}(q_1 + q_2) & \text{if } q_1 + q_2 \leq 1 \\ \lightning & \text{otherwise} \end{cases}$$

$$\mathsf{shot}(n) \cdot \mathsf{shot}(m) := \begin{cases} \mathsf{shot}(n) & \text{if } n = m \\ \lightning & \text{otherwise} \end{cases}$$

This definition has the following important properties:

$$\mathcal{V}(\mathsf{pending}(1) \cdot a) \Rightarrow \mathsf{False} \qquad \text{(pending-excl)}$$

$$\mathsf{pending}(q_1 + q_2) = \mathsf{pending}(q_1) \cdot \mathsf{pending}(q_2) \qquad \text{(pending-split)}$$

$$\mathcal{V}(\mathsf{shot}(n) \cdot \mathsf{shot}(m)) \Rightarrow n = m \qquad \text{(shot-agree)}$$

$$\mathsf{shot}(n) = \mathsf{shot}(n) \cdot \mathsf{shot}(n) \qquad \text{(shot-idem)}$$

[21] The use of $\preccurlyeq$ for an irreflexive relation is an unfortunate accident of the development of the RA axioms over time. We barely ever use inclusion for non-unital RAs, so in practice this is not much of a problem. We considered changing the definition of inclusion so that it always is reflexive, but then other useful properties fail to hold, such as $(a_1, b_1) \preccurlyeq (a_2, b_2) \iff a_1 \preccurlyeq a_2 \wedge b_1 \preccurlyeq b_2$.

[22] Here, we give an explicit definition of the oneshot RA. This may feel a bit verbose, and indeed, in §4.1 we show that this RA can in fact be defined using a couple of combinators.

Note that the use of fractions is not strictly necessary for the example (we could use the "authoritative" mechanism introduced in §4.4 instead), but makes the RA easier to explain.

[23] Notations for invalid elements have not been entirely consistent among Iris papers. In this dissertation we use $\lightning$ to denote the invalid element of an RA (if the RA has such an element), $\bot$ to denote the absence of a core, and ε to denote the global unit (if it exists).

[24] Compositions not defined by the following equation are mapped to $\lightning$.

The property PENDING-EXCL says that composition of $\mathsf{pending}(1)$ with anything else is invalid. As a result of this, if we *own* $\mathsf{pending}(1)$, we know that no other thread can own another part of this location. With PENDING-SPLIT, we also know we can, for example, turn $\mathsf{pending}(1)$ into separate ownership of $\mathsf{pending}(1/2)$ and $\mathsf{pending}(1/2)$, and recombine them later to get back full ownership. This will be useful in the proof.

Furthermore, SHOT-AGREE says that composition of two $\mathsf{shot}(-)$ elements is valid only if the parameters (*i.e.*, the values picked for the oneshot) are the same. This reflects the idea that once a value has been picked, it becomes the only possible value of the location; every thread agrees on what that value is. Finally, SHOT-IDEM says that we can also duplicate ownership of the location as much as we want, *once it has been set to some n*. This allows us to share ownership of this piece of ghost state among any number of threads.

We also have to define the core $|-|$:

$$|\mathsf{pending}(q)| := \bot \qquad |\mathsf{shot}(n)| := \mathsf{shot}(n) \qquad |\lightning| := \lightning$$

Note that $\mathsf{pending}(q)$ has no suitable unit element ($\mathsf{pending}(0)$ is not an element of the carrier M), so we assign no core.

This completes the definition of the oneshot RA. It is now straightforward to verify that this RA satisfies the RA axioms.

Frame-preserving updates. So far, we have defined which states our ghost location can be in and how the state of the location can be distributed across multiple threads. What is still missing, however, is a way of *changing* the ghost location's state. The key is that we want to do this in a *thread-local* way, but each thread only owns some piece of the ghost location. The overall state is obtained by composing all these pieces. Iris always maintains the invariant that the overall state of all ghost locations is a *valid* RA element. When the ghost state *owned by some thread* changes, it is important that this *global* invariant is maintained. We call such well-behaved local ghost state changes *frame-preserving updates.*[25]

The simplest form of frame-preserving update is *deterministic.* We can do a frame-preserving update from a to b (written $a \rightsquigarrow b$) when the following condition is met:

$$\forall c^? \in M^?.\, \mathcal{V}(a \cdot c^?) \Rightarrow \mathcal{V}(b \cdot c^?)$$

In other words, for any resource (called a *frame*) $c^? \in M^?$ such that a is compatible with $c^?$ (*i.e.*, $\mathcal{V}(a \cdot c^?)$), it has to be the case that b is also compatible with $c^?$.

For example, with our oneshot RA it is possible to pick a value if it is still pending, provided we fully own the pending state (and not just some fraction of it):

$$\mathsf{pending}(1) \rightsquigarrow \mathsf{shot}(n) \qquad\qquad (\text{ONESHOT-SHOOT})$$

The reason for this is PENDING-EXCL: $\mathsf{pending}(1)$ actually is not compatible with *any* element; composition always yields $\lightning$. It is thus the case that from $\mathcal{V}(\mathsf{pending}(1) \cdot c^?)$, we know $c^? = \bot$. This makes the rest of the proof trivial.

[25] Frame-preserving updates are equivalent to *semantic entailment* in the Views framework [Din+13].

If we think of the frame $c^?$ as being the composition of the resources owned by all the *other* threads, then a frame-preserving update is guaranteed not to invalidate (the composition of) the resources of *all* running threads. The frame can be $\bot$ if no other thread has any ownership of this ghost location.[26] By doing only frame-preserving updates, we know we will never "step on anybody else's toes".[27]

In general, we also permit *non-deterministic* frame-preserving updates (written $a \rightsquigarrow B$) where the target element b is not fixed a priori, but instead a set B is fixed and some element $b \in B$ is picked depending on the current frame. This is formally defined in Figure 3.4. We will discuss non-deterministic frame-preserving updates further when we encounter our first example of such an update in §4.1.

Ownership and knowledge. Notice that when talking about shot, we use the term "ownership" in a rather loose sense: we say that any element of an RA can be "owned". For elements like $\mathsf{shot}(n)$ that satisfy the property $a = a \cdot a$, owning a is equivalent to owning multiple copies of a—in this particular case, ownership is no longer exclusive, and it may be more appropriate to call this *knowledge*. We can hence think of $\mathsf{shot}(n)$ as representing the knowledge that the value of the location has been set to n.

At the same time, knowledge of $\mathsf{shot}(n)$ is inherently connected with the *(non-duplicable) ownership* of $\mathsf{pending}(q)$, in the sense that if we own $\mathsf{pending}(1)$, we can use ONESHOT-SHOOT to *turn ownership into knowledge* by picking an n. $\mathsf{pending}(q)$ and $\mathsf{shot}(n)$ are also linked in the sense that they are mutually exclusive: having both at the same time is a contradiction ($\mathsf{pending}(q) \cdot \mathsf{shot}(n) = \lightning$). As a consequence, the transition from $\mathsf{pending}$ to shot is *irreversible*: once it happened, that fact becomes freely shareable knowledge, and we can never go back again.

As we can see, resource algebras use the same mechanism to serve a double purpose: modeling both (1) ownership of resources and (2) sharing of knowledge. This unified mechanism is more powerful than both of these aspects would be separately, because resource algebras can *link* ownership and knowledge in intricate ways. This is, in fact, the explanation for why *fancy monoids* (resource algebras) *are all you need* to model both resources and protocols (as alluded to in §2).

Proof rules for ghost state. Resource algebras are embedded into the logic using the proposition $\boxed{a}^{\gamma}$, which asserts ownership of a piece a of the ghost location γ. Validity is expressed using the proposition $\mathcal{V}(a)$.[28] The main connective for manipulating these ghost assertions is called a *view shift*:[29] $P \Rrightarrow_{\mathcal{E}} Q$ says that, given resources satisfying P, we can change the ghost state and end up with resources satisfying Q. (We will come back to the *mask* annotation $\mathcal{E}$ in §3.2.) Intuitively, view shifts are like Hoare triples, but without any code—there is just a precondition and a postcondition. They do not need any code because they only touch the ghost state, which does not correspond to any operation in the actual program. In the following, we discuss the ghost state proof rules shown in Figure 3.5.

[26] Note that the notion of frame-preserving updates is defined for RAs in general, and not just uRAs. To that end, we cannot rely on the presence of a global unit ε to account for the absence of a frame (or the absence of another thread with ownership of the ghost location).

[27] This also implies that every Iris proposition is *stable* in the sense that it cannot be invalidated by actions of other parties (than the one owning the proposition). This is in contrast to many other variants of CSL; we will come back to this point in §7.1.

[28] Since we ignore higher-order ghost state for this dissertation, we can conflate validity in the logic and on the meta-level. For a fully precise description of Iris [Jun+18b], these two concepts must be properly distinguished.

[29] View shifts are derived from the notion of *repartitioning* in CAP [Din+10]. As far as we know, the term "view shift" was first used in HoCAP [SBP13] referring to the Views framework [Din+13]. In related work, view shifts are sometimes called *ghost moves*.

Ghost state.

GHOST-ALLOC
$$\frac{\mathcal{V}(a)}{\mathsf{True} \Rrightarrow_{\mathcal{E}} \exists\gamma.\, \boxed{a}^{\gamma}}$$

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

GHOST-VALID
$$\boxed{a}^{\gamma} \Rightarrow \mathcal{V}(a)$$

GHOST-UPDATE
$$\frac{a \rightsquigarrow B}{\boxed{a}^{\gamma} \Rrightarrow_{\mathcal{E}} \exists b \in B.\, \boxed{b}^{\gamma}}$$

View shifts.

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

VS-REFL
$$P \Rrightarrow_{\mathcal{E}} P$$

VS-TRANS
$$\frac{P \Rrightarrow_{\mathcal{E}} Q \qquad Q \Rrightarrow_{\mathcal{E}} R}{P \Rrightarrow_{\mathcal{E}} R}$$

VS-FRAME
$$\frac{P \Rrightarrow_{\mathcal{E}} Q}{P * R \Rrightarrow_{\mathcal{E}} Q * R}$$

Invariants.

INV-ALLOC
$$P \Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}$$

VS-INV-TIMELESS
$$\frac{P * Q_1 \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}} P * Q_2 \qquad \mathcal{N} \subseteq \mathcal{E} \qquad \mathsf{timeless}(P)}{\boxed{P}^{\mathcal{N}} * Q_1 \Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}} * Q_2}$$

HOARE-INV-TIMELESS
$$\frac{\{P * Q_1\}\, e\, \{v.\, P * Q_2\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathsf{atomic}(e) \qquad \mathsf{timeless}(P) \qquad \mathcal{N} \subseteq \mathcal{E}}{\left\{\boxed{P}^{\mathcal{N}} * Q_1\right\}\, e\, \left\{v.\, \boxed{P}^{\mathcal{N}} * Q_2\right\}_{\mathcal{E}}}$$

Persistent propositions.

PERSISTENT-DUP
$$\frac{\mathsf{persistent}(P)}{P \Leftrightarrow P * P}$$

HOARE-CTX
$$\frac{\{P * Q\}\, e\, \{v.\, R\}_{\mathcal{E}} \qquad \mathsf{persistent}(Q)}{Q \vdash \{P\}\, e\, \{v.\, R\}_{\mathcal{E}}}$$

VS-CTX
$$\frac{P * Q \Rrightarrow_{\mathcal{E}} R \qquad \mathsf{persistent}(Q)}{Q \vdash P \Rrightarrow_{\mathcal{E}} R}$$

PERSISTENT-INV
$$\mathsf{persistent}(\boxed{P}^{\mathcal{N}})$$

PERSISTENT-GHOST
$$\frac{|a| = a}{\mathsf{persistent}(\boxed{a}^{\gamma})}$$

Structural rules for
$\vee$, $\wedge$, $*$, $\forall$, $\exists$.

Timeless propositions.

TIMELESS-GHOST
$$\mathsf{timeless}(\boxed{a}^{\gamma})$$

TIMELESS-HEAP
$$\mathsf{timeless}(\ell \mapsto v)$$

Structural rules for
$\vee$, $\wedge$, $*$, $\forall$, $\exists$.

Figure 3.5: Example Iris proof rules.

The rule GHOST-ALLOC can be used to allocate a new ghost location, with an arbitrary initial state a so long as a is valid according to the chosen RA. The rule GHOST-UPDATE says that we can perform frame-preserving updates on ghost locations, as described above.

The proof rule GHOST-OP says that ghost state can be separated (in the sense of separation logic) following the composition operation $(\cdot)$ defined for the RA, and GHOST-VALID encodes the fact that only valid RA elements can ever be owned.

All the usual structural rules for Hoare triples also hold for view shifts, like framing (VS-FRAME). The rule HOARE-VS illustrates how view shifts are used in program verification: we can apply view shifts in the pre- and postconditions of Hoare triples. This corresponds to composing the "triples for ghost moves" (*i.e.*, view shifts) with a Hoare triple for e. Doing so does not change the expression in the triple because the ghost state actions performed by the view shifts do not speak about any actual code.

hoare-vs is basically the normal rule of consequence of Hoare logic, except that it uses view shifts instead of implications. Still, there is fundamental difference between view shifts ($\Rrightarrow$) on the one hand and implications ($\Rightarrow$) and magic wands ($-\!\!*$) on the other: the implication $P \Rightarrow Q$ says that whenever P holds, Q necessarily holds too *in the same state*. In contrast, the view shift $P \Rrightarrow Q$ says that whenever P holds, Q holds under the proviso that we *change* the ghost state.[30] Hence, unlike implication and wand, the only way to eliminate a view shift is through the rule hoare-vs.

3.2 Invariants

Now that we have set up the structure of our ghost location γ, we have to connect the state of γ to the actual physical value of x. This is done using an *invariant*.

An invariant[31] is a property that holds at all times: each thread accessing the state may assume the invariant holds before each step of its computation, but it must also ensure that the invariant continues to hold after each step. Since we work in a separation logic, the invariant does not just "hold"; it expresses ownership of some resources, and threads accessing the invariant get access to those resources. The rule hoare-inv-timeless, basically the Iris version of critical-region from §2.2 (page 21), realizes this idea as follows:

$$\frac{\{P * Q_1\}\, e\, \{v.\, P * Q_2\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathsf{atomic}(e) \qquad \mathsf{timeless}(P) \qquad \mathcal{N} \subseteq \mathcal{E}}{\left\{\boxed{P}^{\mathcal{N}} * Q_1\right\}\, e\, \left\{v.\, \boxed{P}^{\mathcal{N}} * Q_2\right\}_{\mathcal{E}}}$$

This rule is quite a mouthful, so we will go over it carefully. First, there is the proposition $\boxed{P}^{\mathcal{N}}$, which states that P (an arbitrary proposition) is maintained as an invariant. The rule says that having this proposition in the precondition permits us to access the invariant, which involves acquiring ownership of P *before* the verification of e and giving back ownership of P *after* said verification. Crucially, we require that e is *atomic*, meaning that this computation is guaranteed to complete in a single step. This is essential for soundness: the rule allows us to temporarily use the ownership held by the invariant and even violate said invariant, but after a single atomic step (*i.e.*, before any other thread could take a turn and notice that the invariant is broken), we have to establish the invariant again.

vs-inv-timeless is very similar to hoare-inv-timeless: invariants may be accessed not just when reasoning about atomic expressions, but also when performing a "ghost update" without any code being executed at all.

Notice that $\boxed{P}^{\mathcal{N}}$ is *just another kind of proposition*, and it can be used anywhere that normal propositions can be used—including the pre- and postconditions of Hoare triples, and invariants themselves, resulting in nested invariants. The latter property is sometimes referred to as *impredicativity*. We will discuss the full proof rules for impredicative invariants in §5.1, which will require the introduction of a *later* modality. Until then, we work with simplified rules that require the invariant proposition P

[30] To give a comparison with functional programming, implications/wands correspond to pure functions while view shifts are monadic (also see §5.5).

[31] Ashcroft, "Proving assertions about parallel programs", 1975 [Ash75].

to be *timeless*, which rules out impredicativity. Intuitively, everything "first-order" is timeless, including assertions like heap ownership or ghost ownership.[32] Timelessness is preserved by most logical connectives, *i.e.*, if P and Q are both timeless then so is $P * Q$ *et cetera.* In contrast, Hoare triples, view shifts and invariant assertions themselves are *not* timeless, but it is rare to encounter those *inside* an invariant.[33]

Finally, we come to the *mask* $\mathcal{E}$ and *namespace* $\mathcal{N}$: they avoid the issue of *reentrancy.* The restriction to atomic expressions e in the rule above makes sure no two threads can access the same invariant at the same time, but we still have to make sure that the same invariant is not accessed twice at the same time *by the same thread*, which would incorrectly duplicate the underlying resource. To this end, each invariant is allocated in a *namespace* $\mathcal{N}$. Furthermore, each Hoare triple is annotated with a *mask* to keep track of which invariant namespaces are still enabled.[34] Accessing an invariant removes its namespace from the mask, ensuring that it cannot be accessed again in a nested fashion.

Invariants are created using the INV-ALLOC rule (Figure 3.5): whenever a proposition P has been established, it can be turned into an invariant. This can be viewed as a transfer of the resources backing P from a single thread to a shared space accessible by all threads. Creating an invariant is a view shift; this hints at the fact that invariants themselves are yet another form of (fancy) ghost state.[35]

3.3 Persistent propositions

We have seen that Iris can express both ephemeral *ownership* of exclusive resources (like $\ell \mapsto v$ or $\boxed{\mathsf{pending}(1)}^{\gamma}$), as well as *knowledge* of properties like $\boxed{P}^{\mathcal{N}}$ and $\boxed{\mathsf{shot}(n)}^{\gamma}$ that, once true, hold true forever. We call the latter class of propositions *persistent.* Further examples of persistent propositions are validity ($\mathcal{V}(a)$), equality ($t = u$), Hoare triples ($\{P\}\, e\, \{v.\, Q\}$) and view shifts ($P \Rrightarrow Q$). Persistence is preserved by most logical connectives, *i.e.*, if P and Q are both persistent then so is $P * Q$ and so on. Persistent propositions can be freely duplicated (PERSISTENT-DUP); the usual restriction of resources being usable only once does not apply to them.

Both ownership and knowledge are forms of ghost state in Iris. This versatility of ghost ownership is also visible in its relation to persistence: while ghost ownership of some elements (like $\mathsf{pending}(q)$) is ephemeral, ownership *of a core* is persistent (PERSISTENT-GHOST). This manifests the idea mentioned in §3.1 that RAs can express both ownership and knowledge in one unified framework, with knowledge merely referring to ownership of a persistent resource. In this view, the core $|a|$ is a function which extracts the *knowledge* out of RA element a. In the proof of `check`, persistent ghost ownership will be crucial.

One important role of persistent propositions is related to nested Hoare triples: as expressed by the rule HOARE-CTX, the nested Hoare triple may only use propositions Q from the "outer" context that are persistent. Persistence guarantees that Q will still hold when the Hoare

[32] The exception here is higher-order ghost state, see §6.1

[33] It may seem odd that we pay the high cost of the later modality when actually nesting invariants is a rare thing to do. However, while rare, putting invariants or view shifts into invariants *is* occasionally needed, and moreover prior work showed that the book-keeping required to make sure that invariants do *not* get nested in a higher-order logic is quite heavy (Svendsen, Birkedal, and Parkinson, "Modular reasoning about separation of concurrent data structures", 2013 [SBP13]).

[34] By convention, the absence of a mask means we are using the full mask ($\top$) containing all the namespaces. This includes the specification in Figure 3.3.

[35] Krebbers *et al.*, "The essence of higher-order concurrent separation logic", 2017 [Kre+17].

triple is "invoked" (*i.e.*, when the code it specifies is executed), even if that happens multiple times.

Closely related to persistence is the notion of *duplicable propositions*, *i.e.*, propositions P for which one has $P \Leftrightarrow P * P$. This is a strictly weaker notion, however: not all duplicable propositions are persistent. For example, considering the fractional points-to connective $\ell \overset{q}{\mapsto} v$ mentioned in §2.3, the proposition $\exists q.\, \ell \overset{q}{\mapsto} v$ is duplicable (which follows from halving the fractional permission q), but it is not persistent.[36] Another very similar example is $\exists q.\, \boxed{\mathsf{pending}(q)}^{\gamma}$. Intuitively, the proof of $P \Rightarrow P * P$ for these propositions works by *splitting* the resource into smaller and smaller pieces using rules like pending-split; but none of the resources satisfying these propositions are themselves duplicable (in the sense of $a = a \cdot a$) as would be required for a persistent proposition: we have seen that $\mathsf{shot}(n) = \mathsf{shot}(n) \cdot \mathsf{shot}(n)$ (shot-idem); no such equation holds for $\mathsf{pending}(q)$.

3.4 Proof of the example

We have now seen enough features of Iris to proceed with the actual verification problem outlined in Figure 3.3. We repeat the desired specification and show a Hoare outline of the proof in Figure 3.6.[37] We made all the masks explicit in the specification. The mask $\top$ indicates that all invariants are enabled. The proof outline itself shows all the resources that are owned at the respective program point, and also indicates the current mask in the subscript.

Proof of `mk_oneshot`. First of all, from the allocation performed by the **ref** construct, we obtain $x \mapsto \texttt{inl}(0)$ via hoare-alloc. Next, we use ghost-alloc to allocate a new ghost location γ with the structure of the oneshot RA defined above, picking the initial state $\mathsf{pending}(1)$.[38] We split $\mathsf{pending}(1)$ into two halves, each represented as $\mathsf{pending}(1/2)$, using ghost-op. Finally, we establish and create the following invariant:

$$I := (x \mapsto \texttt{inl}(0) * \boxed{\mathsf{pending}(1/2)}^{\gamma}) \vee (\exists n.\, x \mapsto \texttt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma})$$

Since x is initialized with $\texttt{inl}(0)$, the invariant I initially holds. We use up most of our resource when establishing the invariant with inv-alloc, but we keep one half of the ghost variable.

What remains to be done is establishing our postcondition, which consists of an existentially quantified T and two Hoare triples. We pick T to be $\boxed{\mathsf{pending}(1/2)}^{\gamma}$, matching our remaining resources (except for the invariant $\boxed{I}^{\mathcal{N}}$). Thanks to hoare-ctx, we can use the freshly allocated invariant for the remaining postcondition, the two triples. In the proof outline, hoare-ctx allows us to keep resources when we "step over a λ", but only when the resources are persistent. This corresponds to the fact that the function can be called several times, so that the resources should not be used up by one call.

Proof of `set`. The function `set` accesses location x, so we start by opening the invariant around the **CmpX**. **CmpX** is atomic, so the rule

[36] If we wanted to change this, we would have to give up one or more proof rules for persistence. Specifically, of the rules in Figure 5.4 on page 69, we would at least lose commutativity of persistence with existential quantification, because $\exists q.\, \ell \overset{q}{\mapsto} v$ would be persistent but $\ell \overset{q}{\mapsto} v$ certainly would not [BB17]. (The other direction still works.) From our experiments so far, it seems likely that we would also lose commutativity with universal quantification (in the direction of moving the quantifier in). See `https://gitlab.mpi-sws.org/iris/iris/-/issues/224` for further discussion.

[37] Between curly braces and in green, we show the current context at each step in the proof. The code that is being verified is in blue.

[38] This is implicitly using hoare-vs to justify applying a view shift while verifying a Hoare triple.

Example specification.

$$\{\mathsf{True}\}$$

$$\quad \texttt{mk\_oneshot}()$$

$$\left\{\begin{array}{l} c.\, \exists T.\, T * \big(\forall v.\, \{T\}\; c.\mathsf{set}(v)\; \{\mathsf{True}\}_{\top}\big) * \\ \quad \{\mathsf{True}\}\; c.\mathsf{check}()\; \{f.\, \{\mathsf{True}\}\; f()\; \{\mathsf{True}\}_{\top}\}_{\top} \end{array}\right\}_{\top}$$

Example proof outline.

```
$\texttt{let}\ x = \texttt{ref}(\texttt{inl}(0));$
$\left\{x \mapsto \texttt{inl}(0) * \boxed{\mathsf{pending}(1)}^{\gamma}\right\}_{\top}$    (hoare-alloc, ghost-alloc)
$\left\{x \mapsto \texttt{inl}(0) * \boxed{\mathsf{pending}(1/2)}^{\gamma} * \boxed{\mathsf{pending}(1/2)}^{\gamma}\right\}_{\top}$    (ghost-op)
$\left\{\boxed{I}^{\mathcal{N}} * \boxed{\mathsf{pending}(1/2)}^{\gamma}\right\}_{\top}$    (inv-alloc)
    where $I := (x \mapsto \texttt{inl}(0) * \boxed{\mathsf{pending}(1/2)}^{\gamma}) \lor (\exists n.\, x \mapsto \texttt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma})$
Pick $T := \boxed{\mathsf{pending}(1/2)}^{\gamma}$. We have to prove $T$ (easy) and two Hoare triples. (hoare-ctx)

$\{\ \ \texttt{set} = \lambda n.$
    $\left\{\boxed{I}^{\mathcal{N}} * T\right\}_{\top}$
        $\{I * T\}_{\top\setminus\mathcal{N}}$    (hoare-inv-timeless)
        $\left\{x \mapsto \texttt{inl}(0) * \boxed{\mathsf{pending}(1/2)}^{\gamma} * \boxed{\mathsf{pending}(1/2)}^{\gamma}\right\}_{\top\setminus\mathcal{N}}$    (ghost-op, ghost-valid)
        $\texttt{let}\ (\_, b) = \texttt{CmpX}(x, \texttt{inl}(0), \texttt{inr}(n));$
        $\left\{x \mapsto \texttt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma} * b = \texttt{true}\right\}_{\top\setminus\mathcal{N}}$    (hoare-cmpx-suc, ghost-op, ghost-update)
        $\{I * b = \texttt{true}\}_{\top\setminus\mathcal{N}}$
    $\left\{\boxed{I}^{\mathcal{N}} * b = \texttt{true}\right\}_{\top}$
    $\texttt{assert}(b)$    (hoare-assert)
    $\{\mathsf{True}\}_{\top}$

  $\texttt{check} = \lambda\_.$
    $\left\{\boxed{I}^{\mathcal{N}}\right\}_{\top}$
        $\{I\}_{\top\setminus\mathcal{N}}\ \texttt{let}\ y = !\,x;\ \{I * P\}_{\top\setminus\mathcal{N}}$    (hoare-inv-timeless, hoare-load, ghost-op)
            where $P := y = \texttt{inl}(0) \lor (\exists n.\, y = \texttt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma})$
    $\left\{\boxed{I}^{\mathcal{N}} * P\right\}_{\top}$
        $\lambda\_.$
        $\left\{\boxed{I}^{\mathcal{N}} * P\right\}_{\top}$
            $\{I * P\}_{\top\setminus\mathcal{N}}$    (hoare-inv-timeless)
            $\texttt{let}\ y' = !\,x;$    (hoare-load)
            $\{I * (y = \texttt{inl}(0) \lor (\exists n.\, y = y' = \texttt{inr}(n)))\}_{\top\setminus\mathcal{N}}$    (ghost-op, ghost-valid)
        $\left\{\boxed{I}^{\mathcal{N}} * (y = \texttt{inl}(0) \lor (\exists n.\, y = y' = \texttt{inr}(n)))\right\}_{\top}$
        $\texttt{match}\ y\ \texttt{with}$
          $\texttt{inl}(\_) \Rightarrow ()$
        $\mid \texttt{inr}(\_) \Rightarrow \texttt{assert}(y = y')$    (hoare-assert)
        $\texttt{end}$
        $\{\mathsf{True}\}_{\top}$
    $\}$
```

Figure 3.6: Example proof outline.

hoare-inv-timeless applies.[39] This changes our current mask to $\top \setminus \mathcal{N}$. We also obtain I, and perform case distinction on the disjunction.

If we are in the right-hand case (the one with shot), our resources are described by

$$(\exists n.\, x \mapsto \mathtt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma}) \quad * \quad \boxed{\mathsf{pending}(1/2)}^{\gamma}$$

From this, we can derive a contradiction: using ghost-op, we can obtain $\boxed{\mathsf{shot}(n) \cdot \mathsf{pending}(1/2)}^{\gamma}$ which equals $\boxed{\lightning}^{\gamma}$. According to ghost-valid, this is impossible.

Thus we can conclude that we are in the left disjunct, and the **CmpX** will succeed (hoare-cmpx-suc). After the **CmpX**, we have the following (using ghost-op to merge the two halves of $\mathsf{pending}$ back together):

$$x \mapsto \mathtt{inr}(n) * \boxed{\mathsf{pending}(1)}^{\gamma}$$

How can we reestablish the invariant I after this **CmpX**? Clearly, we must pick the right disjunct, since $x \mapsto \mathtt{inr}(n)$. Hence we have to update the ghost state to match the physical state. To this end, we apply ghost-update with the frame-preserving update oneshot-shoot, which allows us to update the ghost location to $\mathsf{shot}(n)$ if we own $\mathsf{pending}(1)$, which we do. We then have I again and can finish the proof. Since **CmpX** succeeded, the assertion is trivially verified with hoare-assert.

Notice that we could *not* complete the proof if `set` would ever change x again, since oneshot-shoot can only ever be used once on a particular ghost location. We have to be in the $\mathsf{pending}(\_)$ state if we want to pick the n in $\mathsf{shot}(n)$. This is exactly what we would expect, since `check` indeed relies on x not being modified once it has been set to $\mathtt{inr}(n)$.

Proof of `check`. What remains is to prove correctness of `check`. We open our invariant I to justify the safety of $!\,x$. This is immediate because the invariant provides $x \mapsto \_$ no matter which side of the disjunction we obtain. However, we will *not* immediately close I again. Instead, we will have to acquire some piece of ghost state that shows that *if* we read an $\mathtt{inr}(n)$, then x will not change its value. At this point in the proof, we have the following proposition:

$$\begin{array}{l} x \mapsto y \ * \\ \big((y = \mathtt{inl}(0) * \boxed{\mathsf{pending}(1/2)}^{\gamma}) \ \lor \ (\exists n.\, y = \mathtt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma})\big) \end{array}$$

We use the identity $\mathsf{shot}(n) = \mathsf{shot}(n) \cdot \mathsf{shot}(n)$ with ghost-op to show that this logically implies:

$$\begin{array}{l} x \mapsto y \ * \\ \big((y = \mathtt{inl}(0) * \boxed{\mathsf{pending}(1/2)}^{\gamma}) \ \lor \\ \quad (\exists n.\, y = \mathtt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma} * \boxed{\mathsf{shot}(n)}^{\gamma})\big) \end{array}$$

which in turn implies:

$$I \quad * \quad \underbrace{(y = \mathtt{inl}(0) \lor (\exists n.\, y = \mathtt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma}))}_{P}$$

[39] We indicate the nested nature of the invariant rule by indenting the part of the proof outline that happens “inside” the invariant.

We can thus reestablish the invariant I, but we keep P, the information we gathered about y. The plan is to use this in the proof of the closure that we return.

To do so, HOARE-CTX requires us to show that P is persistent: as discussed in §3.3, $\boxed{\mathsf{shot}(n)}^{\gamma}$ and equality are persistent, and both disjunction and separating conjunction preserve persistence. This matches the intuition that, once we observe that x has been set, we can then forever assume it will not change again.

To finish this proof, let us look at the closure returned by `check` in more detail: Again, we will open our invariant to justify the safety of $!\,x$. Our proposition then is $I * P$. In order to proceed, we now distinguish the possible cases in P and I.

Case 1 (P "on the left"): We have $I \;*\; y = \mathtt{inl}(0)$. In this case, the match will always pick the first arm, and there is nothing left to show.

Case 2 (P "on the right", I "on the left"): We have:

$$x \mapsto \mathtt{inl}(0) * \boxed{\mathsf{pending}(1/2)}^{\gamma} \;*\; (\exists n.\, y = \mathtt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma})$$

We own both $\mathsf{pending}(1/2)$ and $\mathsf{shot}(n)$, which compose to $\lightning$. By GHOST-VALID, this is a contradiction.

Case 3 (P "on the right", I "on the right"): We have:

$$(\exists m.\, x \mapsto \mathtt{inr}(m) * \boxed{\mathsf{shot}(m)}^{\gamma}) \;*\; (\exists n.\, y = \mathtt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma})$$

In particular, we obtain $\boxed{\mathsf{shot}(n) \cdot \mathsf{shot}(m)}^{\gamma}$. Using GHOST-VALID, this yields $\mathcal{V}(\mathsf{shot}(n) \cdot \mathsf{shot}(m))$, implying $n = m$ by SHOT-AGREE. We are hence left with:

$$x \mapsto \mathtt{inr}(n) * y = \mathtt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma}$$

In particular, we get that $y' = y = \mathtt{inr}(n)$, *i.e.*, the assertion in the second match arm will succeed.

Chapter 4

Ghost state constructions

In the previous chapter we have seen that user-defined ghost state plays an important role in Iris. Ghost state in Iris is defined through resource algebras (RAs), at which we will have a more thorough look in this chapter. First of all, in §4.1, we show that many frequently needed RAs like the one we defined in the previous chapter can be constructed by composing smaller, reusable pieces. In §4.2, we show how to make *state-transition systems with tokens*, a general class of protocols, available in Iris by defining them as an RA. In §4.3 we then show that the ownership connective $\boxed{a}^{\gamma}$ can in fact be defined in terms of an even more primitive notion of "global" ghost ownership, without a built-in idea of a "name" γ. We finish this chapter in §4.4 with a brief introduction to the *authoritative RA*, arguably the most important resource algebra construction in Iris.

4.1 RA constructions

One of the key features of Iris is that it leaves the structure of ghost state entirely up to the user of the logic. If there is the need for some special-purpose RA, the user has the freedom to directly use it. However, it turns out that many frequently needed RAs can be constructed by composing smaller, reusable pieces—so while we have the entire space of RAs available when needed, we do not have to construct custom RAs for every new proof.

For example, looking at the oneshot RA from §3.1, it really does three things:

1. It separates the allocation of an element of the RA from the decision about what value to store there (ONESHOT-SHOOT).
2. While the oneshot location is uninitialized, ownership is tracked with a fraction, *i.e.*, at most one thread can *fully* own the location.
3. Once the value has been decided on, ownership turns into knowledge; now the RA makes sure everybody *agrees* on that value.

We can thus decompose the oneshot RA into the *sum*, *exclusive* and *agreement* RAs as described below. (In the definitions of all the RAs, the omitted cases of composition and core are all ↯.)

Fraction. The task of the *fraction RA*, *Frac*, is to reflect the idea of a resource that can be taken apart by splitting the "full" resource (represented by 1) into smaller pieces:

$$
\begin{aligned}
\mathit{Frac} &:= \mathsf{frac}(q : \mathbb{Q} \cap (0,1]) \mid \lightning \\
\mathsf{frac}(q_1) \cdot \mathsf{frac}(q_2) &:= \begin{cases} \mathsf{frac}(q_1 + q_2) & \text{if } q_1 + q_2 \leq 1 \\ \lightning & \text{otherwise} \end{cases} \\
\mathcal{V}(a) &:= a \neq \lightning \\
|a| &:= \bot
\end{aligned}
$$

Notice the similarity between composition here and composition of $\mathsf{pending}$ in §3.1.

However, the problem with this definition is that to even write $\mathsf{frac}(q)$, we have to already establish that q is in the interval $(0,1]$. For this reason, we instead use the following definition, also avoiding an explicit invalid element $\lightning$:

$$
\begin{aligned}
\mathit{Frac} &:= \mathbb{Q}_+ \\
q_1 \cdot q_2 &:= q_1 + q_2 \\
\mathcal{V}(a) &:= a \leq 1 \\
|a| &:= \bot
\end{aligned}
$$

Here, $\mathbb{Q}_+$ is the set of all strictly positive rational numbers. The set of valid elements is the same as for our previous definition, but this time, we do not have to prove $q \leq 1$ to merely state that we own some resource: resources like 5 exist, but are invalid. This will be useful in §4.4.

Agreement. Given a set X, the task of the *agreement RA*, $\mathit{Ag}_0(X)$, is to make sure multiple parties can *agree* on which value $x \in X$ has been picked.[1] We define Ag_0 as follows:

$$
\begin{aligned}
\mathit{Ag}_0(X) &:= \mathsf{ag}_0(x : X) \mid \lightning \\
\mathcal{V}(a) &:= \exists x \in X.\, a = \mathsf{ag}_0(x) \\
\mathsf{ag}_0(x) \cdot \mathsf{ag}_0(y) &:= \begin{cases} \mathsf{ag}_0(x) & \text{if } x = y \\ \lightning & \text{otherwise} \end{cases} \\
|\mathsf{ag}_0(x)| &:= \mathsf{ag}_0(x)
\end{aligned}
$$

In particular, agreement satisfies the following property corresponding to shot-agree:

$$\mathcal{V}(\mathsf{ag}_0(x) \cdot \mathsf{ag}_0(y)) \Rightarrow x = y \qquad (\textsc{ag0-agree})$$

The only other properties we need are that ag_0 is injective, and that all *valid* RA elements are of the form $\mathsf{ag}_0(\_)$:

$$
\begin{aligned}
\mathsf{ag}_0(x) = \mathsf{ag}_0(y) &\Rightarrow x = y && (\textsc{ag0-inj}) \\
\mathcal{V}(a) &\Rightarrow \exists x.\, a = \mathsf{ag}_0(x) && (\textsc{ag0-uninj})
\end{aligned}
$$

[1] We call this Ag_0 because Ag—the full definition of the agreement RA—is more complex, because it has to support higher-order ghost state (see §6.1). It is described in "Iris from the ground up: A modular foundation for higher-order concurrent separation logic" [Jun+18b].

Sum. Given any two RAs M_1 and M_2, the *sum RA*, $M_1 +_{\lightning} M_2$, formalizes the idea of "either this or that":[2]

$$\begin{aligned}
M_1 +_{\lightning} M_2 &:= \mathsf{inl}(a_1 : M_1) \mid \mathsf{inr}(a_2 : M_2) \mid \lightning \\
\mathcal{V}(a) &:= (\exists a_1 \in M_1.\, a = \mathsf{inl}(a_1) \land \mathcal{V}_1(a_1)) \lor \\
&\ (\exists a_2 \in M_2.\, a = \mathsf{inr}(a_2) \land \mathcal{V}_2(a_2)) \\
\mathsf{inl}(a_1) \cdot \mathsf{inl}(a_2) &:= \mathsf{inl}(a_1 \cdot a_2) \\
\mathsf{inr}(a_1) \cdot \mathsf{inr}(a_2) &:= \mathsf{inr}(a_1 \cdot a_2) \\
|\mathsf{inl}(a_1)| &:= \begin{cases} \bot & \text{if } |a_1| = \bot \\ \mathsf{inl}(|a_1|) & \text{otherwise} \end{cases} \\
|\mathsf{inr}(a_2)| &:= \begin{cases} \bot & \text{if } |a_2| = \bot \\ \mathsf{inr}(|a_2|) & \text{otherwise} \end{cases} \\
|\lightning| &:= \lightning
\end{aligned}$$

Due to composition being a total function for RAs, we cannot immediately equip the sum type with an RA structure; we have to add a $\lightning$ element. If composition were a three-place relation,[3] this would not be needed; we will come back to this in §7.2.

Oneshot. We can now define the general idea of the oneshot RA as $\mathit{OneShot}(X) := \mathit{Frac} +_{\lightning} \mathit{Ag}_0(X)$, and recover the RA for the example as $\mathit{OneShot}(\mathbb{Z})$. The old $\mathsf{pending}(q)$ corresponds to $\mathsf{inl}(q)$ and the old $\mathsf{shot}(n)$ corresponds to $\mathsf{inr}(\mathsf{ag}_0(n))$. The new oneshot RA contains some extra invalid "junk" elements, like $\mathsf{inl}(\lightning)$, which did not exist before—but those do not change any of the relevant properties of the RA.

In other words, what first seemed like a very special-case tool that we had to introduce just to verify the example can actually be obtained by composing some very general constructions that can be reused for a wide range of verifications. The RA combinators that have been and will be presented in this dissertation cover the vast majority of use-cases. By composing RA combinators, the work of defining a resource algebra and stating and verifying its key properties has to be done only once; users of Iris mostly just have to plug together the pieces.

Frame-preserving updates and exclusive elements. Another advantage of decomposing RAs into separate pieces is that for the frame-preserving updates of these pieces, we can prove general laws. While the fraction and agreement RAs permit no interesting frame-preserving updates, the sum RA satisfies the following rules:

$$\textsc{inl-update}\ \frac{a_1 \rightsquigarrow B_1}{\mathsf{inl}(a_1) \rightsquigarrow \{\mathsf{inl}(b_1) \mid b_1 \in B_1\}} \qquad \textsc{inr-update}\ \frac{a_2 \rightsquigarrow B_2}{\mathsf{inr}(a_2) \rightsquigarrow \{\mathsf{inr}(b_2) \mid b_2 \in B_2\}}$$

(These are non-deterministic updates, of which we will see a concrete example later in this section.)

[2] As before, compositions not defined here yield $\lightning$.

[3] Dockins, Hobor, and Appel, "A fresh look at separation algebras and share accounting", 2009 [DHA09].

However, these two rules are not sufficient to derive ONESHOT-SHOOT, the frame-preserving update from our example in the previous chapter. This update changes from the left summand to the right one:

$$\mathsf{inl}(1) \rightsquigarrow \mathsf{inr}(\mathsf{ag}_0(x))$$

Such a frame-preserving update can be proven because $\mathsf{inl}(1)$ has *no* frame, *i.e.*, there is no c with $\mathcal{V}(\mathsf{inl}(1) \cdot c)$. The absence of a frame allows us to pick any valid RA element on the right-hand side. It turns out that we can easily generalize this idea to any RA: we say that an RA element a is *exclusive* (or *has no frame*) if:

$$\mathsf{exclusive}(a) := \forall c.\, \neg\mathcal{V}(a \cdot c)$$

For exclusive elements we have the following generic frame-preserving update:

$$\textsc{exclusive-update}\quad \frac{\mathsf{exclusive}(a) \qquad \mathcal{V}(b)}{a \rightsquigarrow b}$$

Notice that the above frame-preserving update is only useful for an RA with a partial core: $\mathsf{exclusive}(a)$ is always false for any valid RA element a with $|a| \neq \bot$, because then we have $\mathcal{V}(a \cdot |a|)$ (by RA-CORE-ID). Obtaining frame-preserving updates for switching the "side" of a sum is one of our key motivations for making the core a partial function instead of a total function.

For the RAs we have defined so far, the following rules identify exclusive elements:

$$\textsc{frac-exclusive}\quad \mathsf{exclusive}(1) \qquad\qquad \textsc{inl-exclusive}\quad \frac{\mathsf{exclusive}(a_1)}{\mathsf{exclusive}(\mathsf{inl}(a_1))} \qquad\qquad \textsc{inr-exclusive}\quad \frac{\mathsf{exclusive}(a_2)}{\mathsf{exclusive}(\mathsf{inr}(a_2))}$$

Products and finite partial maps. Products can easily be equipped with an RA structure by lifting composition and core pointwise, and requiring all elements to be valid for $\mathcal{V}$. Exclusiveness and frame-preserving updates are likewise propagated pointwise; note that it suffices for *any* element of the pair to be exclusive (this makes $(1, x)$ exclusive for any x):

$$\textsc{prod-update}\quad \frac{a_i \rightsquigarrow B_i}{(a_1, \ldots, a_n) \rightsquigarrow \{(a_1, \ldots, b_i, \ldots, a_n) \mid b_i \in B_i\}} \qquad\qquad \textsc{prod-exclusive}\quad \frac{\mathsf{exclusive}(a_i)}{\mathsf{exclusive}((a_1, \ldots, a_n))}$$

For finite partial maps $I \xrightharpoonup{\mathsf{fin}} M$ (or just finite maps for short) with some RA M as their codomain, we can similarly lift composition pointwise, taking care of partiality as needed. But first, we have to introduce some notation: for some $f : I \xrightharpoonup{\mathsf{fin}} M$, we write $f(i) = \bot$ to indicate that no value is associated with i, and we write $\mathsf{dom}(f)$ to denote the *domain* of f, *i.e.*, the set of all keys that have a value associated with them. The domain always has to be *finite*, and I is assumed to be infinite. We use $(i \mapsto a) \in f$ as notation for $i \in \mathsf{dom}(f) \land f(i) = a$.

Now we can define:

$$\mathcal{V}(f) := \forall (i \mapsto a) \in f.\, \mathcal{V}(a)$$

$$f_1 \cdot f_2 := \lambda i.\, \begin{cases} f_1(i) \cdot f_2(i) & \text{if } i \in \mathsf{dom}(f_1) \cap \mathsf{dom}(f_2) \\ f_1(i) & \text{if } i \in \mathsf{dom}(f_1) \setminus \mathsf{dom}(f_2) \\ f_2(i) & \text{if } i \in \mathsf{dom}(f_2) \setminus \mathsf{dom}(f_1) \\ \bot & \text{otherwise} \end{cases}$$

$$|f| := \lambda i.\, \begin{cases} |f(i)| & \text{if } i \in \mathsf{dom}(f) \wedge |f(i)| \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

It is easy to verify that $f_1 \cdot f_2$ and $|f|$ have finite domain if f_1, f_2 and f do, respectively.

The following laws hold for frame-preserving updates in finite maps:

fmap-update
$$\frac{a \rightsquigarrow B}{f[i \leftarrow a] \rightsquigarrow \{f[i \leftarrow b] \mid b \in B\}}$$

fmap-alloc
$$\frac{\mathcal{V}(a)}{f \rightsquigarrow \{f[i \leftarrow a] \mid i \notin \mathsf{dom}(f)\}}$$

fmap-alloc-dep
$$\frac{\forall i.\, \mathcal{V}(g(i))}{f \rightsquigarrow \{f[i \leftarrow g(i)] \mid i \notin \mathsf{dom}(f)\}}$$

Here, we use $f[i \leftarrow a]$ as notation for a function that is equal to f everywhere except for i, where it returns a. In other words, this "stores" the value a at i. The rule fmap-update expresses pointwise lifting of updates to the finite map.

Notice that fmap-alloc is our first original *non-deterministic* frame-preserving update: we cannot pick the index i of the new element in advance because it depends on the frame. Instead, we are guaranteed to extend f at *some* fresh index i being mapped to a. This is a form of *angelic* non-determinism[4] because we can assume that some value for i is picked that "makes everything work", and because I is infinite we know that such a choice is always possible.

fmap-alloc-dep takes this one step further by permitting the initial value of the freshly allocated map entry to depend on the index i it is allocated at. The proof of this tricky-looking rule is not complicated at all: just like for fmap-alloc, the definition of frame-preserving updates says that we learn what the frame g is; then we pick some fresh index i that is used in neither f nor g and extend f with $[i \leftarrow g(i)]$.

Exclusive RA and the heap. Before looking at a more complicated RA construction, we briefly want to describe how the standard separation logic *heap* with disjoint union for its composition can be defined.

For this, we need one more RA construction: given a set X, the task of the *exclusive RA*, *Ex*(X), is to make sure that one party *exclusively* owns a value $x \in X$. We define *Ex* as follows:

$$\begin{aligned} \mathit{Ex}(X) &:= \mathsf{ex}(x : X) \mid \lightning \\ \mathcal{V}(a) &:= a \neq \lightning \\ |\mathsf{ex}(x)| &:= \bot \end{aligned}$$

Composition is always $\lightning$ to ensure that ownership is exclusive.

[4] Broy and Wirsing, "On the algebraic specification of nondeterministic programming languages", 1981 [BW81].

We also have that elements of the exclusive RA are, well, exclusive:

$$\frac{\textsc{ex-exclusive}}{\mathsf{exclusive}(\mathsf{ex}(x))}$$

Now consider the following RA (for some notion of "locations" and "values"):

$$\mathit{Heap} := \mathit{Loc} \xrightarrow{\mathsf{fin}} \mathit{Ex}(\mathit{Val})$$

If we only focus on the *valid* elements of *Heap*, this is the same as a finite partial map from locations to values. When composing two valid heaps, we compose the exclusively owned values pointwise. This means that if they overlap, we will be composing two $\mathsf{ex}(v)$, which results in $\lightning$. So, the resulting heap is valid only if the two heaps we are composing are disjoint. The RA *Heap* corresponds exactly to the usual "heap with disjoint union" PCM that forms the basis of the standard model of separation logic. In Iris, we can obtain that same model of resources, but we obtain it by composing smaller pieces.

From here, it is only a small step to a heap for fractional permissions:[5]

$$\mathit{FHeap} := \mathit{Loc} \xrightarrow{\mathsf{fin}} \mathit{Frac} \times \mathit{Ag}_0(\mathit{Val})$$

Considering again only the valid elements of this RA, for every location we have a fraction and a value. When composing two heaps, Ag_0 demands that overlapping values must be equal, and *Frac* demands that the sum of the fractions does not exceed 1. This matches exactly the usual PCM used to model a heap with fractional permissions.

Obtaining these well-known notions of resources in a compositional way is extremely useful when coming up with new kinds of resources needed for some proof. We can just plug together the pieces and know that all RA axioms are satisfied by construction. This also helps to identify patterns: notice how we turned the exclusive heap into a fractional heap by replacing $\mathit{Ex}(X)$ by $\mathit{Frac} \times \mathit{Ag}_0(X)$. This same pattern applies any time we want to make an RA more fine-grained by introducing fractional permissions.[6]

To our knowledge, this decomposition is a novel contribution of Iris. The key idea is to permit *elements without a unit*, which prior work did not consider: sometimes different elements could have different units,[7] which somewhat corresponds to the core of a resource algebra, but that core still had to be *total*. As we have seen, even when defining RAs like *Heap* or *FHeap* that *do* have a unit, if we want to view those RAs as a composition of smaller pieces we end up with RAs like *Frac* or $\mathit{Ex}(X)$ that can only be defined when units do not have to exist.[8]

4.2 State-transition systems

One common and intuitive way to describe the interaction of concurrent systems is through *state-transition systems* (STSs). STSs consist of a set of *states* and a (directed) *transition* relation between them, describing how threads are allowed to move from state to state. On top of that, a proper description of system behavior often requires some form of exclusive *tokens* to introduce asymmetry: some transitions can only be taken by the

[5] Boyland, "Checking interference with fractional permissions", 2003 [Boy03]; Bornat *et al.*, "Permission accounting in separation logic", 2005 [Bor+05].

[6] In fact, the example RA from §3.1 was obtained via a similar generalization from the RA used in the previously published version of that example: $\mathit{Ex}(())$ turned into *Frac*. Note that $\mathit{Ag}_0(())$ carries neither knowledge nor ownership so there is no point in adding it.

[7] Dockins, Hobor, and Appel, "A fresh look at separation algebras and share accounting", 2009 [DHA09].

[8] It might be tempting to just add a trivial unit to any RA to make it a uRA (and indeed, that construction is occasionally useful, see §4.4). But this also implies that no element can be *exclusive*, thus removing the possibility of frame-preserving updates such as those based on frac-exclusive.

thread owning the token. In Iris, we follow the style of STSs pioneered by CaReSL[9] where states are treated abstractly and paired with a *state interpretation function* φ defining, for each state, the logical assertion that has to hold in that state.

For example, the one-shot protocol from §3 could be described by the STS and state interpretation given in Figure 4.1. Here, $\mathsf{pending}$ is the initial state, and in that state the protocol owns $x \mapsto \texttt{inl}(0)$.[10] $\mathsf{shot}(n)$ represents a *family* of states, one for every choice of $n \in \mathbb{Z}$, reflecting the possible values of x after $\texttt{set}$ was called. Finally, the STS comes with a *token* $[\mathsf{tok}]$. The fact that $\mathsf{shot}(n)$ contains that token (and $\mathsf{pending}$ does not) means that a thread transitioning from $\mathsf{pending}$ to $\mathsf{shot}(n)$ needs to relinquish ownership of said token: the *law of token conservation* says that the tokens owned by the thread before the transition, plus the tokens owned by the initial state, must be the same as the tokens owned by the thread after the transition plus the tokens owned by the final state. So, if the last state we saw is $\mathsf{pending}$ and we do not own any tokens, the STS might have transitioned to $\mathsf{shot}(n)$ (for some n) in the meantime. But if the last state we saw is $\mathsf{pending}$ and we *do* own the exclusive token $[\mathsf{tok}]$, we know that the STS will still be in the same state, since nobody else could have made that transition.

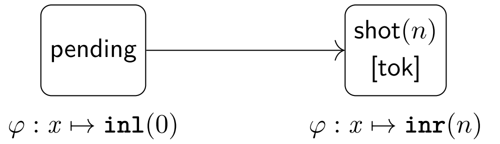
pending
shot(n)
[tok]
$\varphi : x \mapsto \texttt{inl}(0)$
$\varphi : x \mapsto \texttt{inr}(n)$

Figure 4.1: STS for the one-shot protocol.

The point of this section is to explain how STSs like the above can be obtained in Iris as a combination of the right resource algebra and an invariant. Given that STS-like mechanisms are the predominant protocol abstractions in concurrent separation logics, a logic like Iris that claims to subsume prior work has to provide support for this mechanism as well. Being able to encode STSs in terms of simpler mechanisms is a key innovation of Iris.

A general STS is defined by a set of states $\mathcal{S}$, a transition relation $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$, a set of tokens $\mathcal{T}$, a labeling $\mathcal{L} : \mathcal{S} \rightarrow \wp(\mathcal{T})$ of *protocol-owned* tokens for each state, and a state interpretation function $\varphi : \mathcal{S} \rightarrow \mathit{iProp}$.[11] We first define a resource algebra reflecting the abstract transition structure and tokens of the STS. Then we use an Iris invariant to tie the abstract STS to the state interpretation, similar to what we did in the example in §3.

A resource algebra for STSs. Similar to STSs in CaReSL,[12] we first lift the transition relation to a relation on states and tokens owned by the current thread (implementing the *law of token conservation*) and also define a stepping relation for the *frame* of a given token set:

$$(s, T) \rightarrow (s', T') := s \rightarrow s' \land \mathcal{L}(s) \uplus T = \mathcal{L}(s') \uplus T'$$

$$s \xrightarrow{T} s' := \exists T_1, T_2.\, T_1 \mathrel{\#} \bigl(\mathcal{L}(s) \cup T\bigr) \land (s, T_1) \rightarrow (s', T_2)$$

[9] 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].

[10] Protocols owning resources work entirely analogously to invariants owning resources, which we have seen in §3.4. In fact, an invariant is essentially a degenerate STS with just one state.

[11] We impose no restriction on the transition relation; in particular, cycles are allowed.

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

Intuitively, $(s, T) \rightarrow (s', T')$ means that a thread owning tokens T can transition from state s to state s', and will then own tokens T'. For example, in our one-shot STS, we have $(\mathsf{pending}, \{[\mathsf{tok}]\}) \rightarrow (\mathsf{shot}(n), \emptyset)$. The frame-step relation $s \xrightarrow{T} s'$ says that if a thread owns tokens T, *other* threads could step from s to s'. Here, $X \mathrel{\#} Y$ expresses disjointness of the two sets. In the one-shot STS, one crucial property is that $\mathsf{pending} \xrightarrow{\{[\mathsf{tok}]\}} \mathsf{shot}(n)$ does *not* hold, reflecting that if we own the token, the frame *cannot* step from $\mathsf{pending}$ to $\mathsf{shot}(n)$.

We further define *closed* sets of states (given a particular set of tokens) as well as the *closure* of a set:

$$\begin{aligned}\mathsf{closed}(S, T) &:= \forall s \in S.\, \mathcal{L}(s) \mathrel{\#} T \land \Big(\forall s'.\, s \xrightarrow{T} s' \Rightarrow s' \in S\Big)\\ \uparrow(S, T) &:= \Big\{s' \in \mathcal{S} \,\Big|\, \exists s \in S.\, s \xrightarrow{T}{}^{*} s'\Big\}\end{aligned}$$

A set of states with some set of tokens is closed if the frame cannot make any steps that leave the set. This means that all possible actions of other threads have already been taken into account in that set.[13] For example, we have $\mathsf{closed}(\{\mathsf{shot}(n)\}, \emptyset)$ reflecting the fact that once we are in some state $\mathsf{shot}(n)$, even if we do not own any tokens, we know we will stay in that state forever. We also have $\mathsf{closed}(\{\mathsf{pending}\}, \{[\mathsf{tok}]\})$ because if we *do* own the token and are in state $\mathsf{pending}$, we will stay in that state until we decide to use the token (the frame cannot step out of that state). But we do *not* have $\mathsf{closed}(\{\mathsf{pending}\}, \emptyset)$ because the frame could own the token and step from $\mathsf{pending}$ to $\mathsf{shot}(n)$.

It is easy to show that, if (S, T) is well-formed in the sense that T is disjoint from the tokens of states in S, then $\uparrow(S, T)$ is closed w.r.t. T.

The domain of the STS RA is now defined as follows:

$$\begin{aligned}M &:= \mathsf{auth}(s : \mathcal{S}, T : \wp(\mathcal{T}) \mid \mathcal{L}(s) \mathrel{\#} T)\\ &\quad\mid \mathsf{frag}(S : \wp(\mathcal{S}), T : \wp(\mathcal{T}) \mid \mathsf{closed}(S, T) \land S \neq \emptyset)\\ &\quad\mid \lightning\\ \mathcal{V}(a) &:= a \neq \lightning\end{aligned}$$

There are two kinds of elements in this resource algebra (next to the usual "invalid" element $\lightning$):

- frag represents a partial view of the protocol: S is a set of states, and owning $\mathsf{frag}(S, T)$ means that the current state of the protocol is *some* state in that set; moreover, we own the tokens T. Which of those states the protocol is in can change at any time, even without an action of our own, because other threads may take steps in the protocol. This set has to be closed to ensure that all possible such actions are taken into account.
- auth is owned by the "authority" that knows exactly what the current state of the protocol is. This element is exclusive, meaning only one such element exists per instance of the RA. auth also carries a set of tokens, which is only needed to properly define composition.

Based on this intuition, the following definitions for composition and core of STSs should not be too surprising:[14]

[13] For a set of states with some tokens to be closed, we also require that those tokens do not overlap with any of the tokens owned by any of the possible states. If we own a token, we cannot be in any of the states owning that token, so this excludes some ill-formed combinations of states and tokens. Folding that requirement into closed is mostly a technicality; it makes defining the RA later easier.

[14] As usual, the remaining cases are all invalid ($\lightning$).

$$\begin{aligned}
\mathsf{frag}(S_1, T_1) \cdot \mathsf{frag}(S_2, T_2) &:= \mathsf{frag}(S_1 \cap S_2, T_1 \cup T_2) \\
&\qquad\qquad \text{if } T_1 \mathrel{\#} T_2 \text{ and } S_1 \cap S_2 \neq \emptyset \\
\mathsf{frag}(S, T) \cdot \mathsf{auth}(s, T') &:= \mathsf{auth}(s, T') \cdot \mathsf{frag}(S, T) := \mathsf{auth}(s, T \cup T') \\
&\qquad\qquad \text{if } T \mathrel{\#} T' \text{ and } s \in S \\
|\mathsf{frag}(S, T)| &:= \mathsf{frag}(\uparrow(S, \emptyset), \emptyset) \\
|\mathsf{auth}(s, T)| &:= \mathsf{frag}(\uparrow(\{s\}, \emptyset), \emptyset)
\end{aligned}$$

The token sets are composed by disjoint union, reflecting their exclusivity.[15] State sets are composed via intersection: if we know that we are in one of the states of S_1, *and* we know that we are in one of the states of S_2, then the combination of these pieces of knowledge is that we are in one of the states in $S_1 \cap S_2$. That intersection can never be empty because we are always in *some* state. One key proof obligation here is to show that this composition always yields sets of states and tokens that are closed, which indeed turns out to be the case.

Composition of the authoritative element with a fragment works very similarly. Two authoritative elements of course cannot be composed; this is what ensures their exclusivity.

For the core, we "forget" all tokens and replace "authoritative" ownership by the per-thread view. We then need to close our new set of states under frame steps again to make sure we got a valid element of the RA.[16]

Based on this, we can show the following frame-preserving update:

$$\textsc{sts-step}\quad \frac{(s, T) \rightarrow^* (s', T')}{\mathsf{auth}(s, T) \rightsquigarrow \mathsf{auth}(s', T')}$$

This update is quite remarkable! It shows that the structure of frame-preserving updates on the STS RA mirrors the transition relation of the STS. One the one hand, that is of course exactly the point. On the other hand, this is in stark contrast to the other RAs we have considered so far: there, we started by defining the "things one can know or own" (the elements of the RA, such as $\mathsf{pending}(q)$ and $\mathsf{shot}(n)$), next defined how they compose, and finally *proved* which state transitions are possible. The STS RA as a construction lets us go the other way: we start by defining the states and their transitions, and then we can *prove* that a statement like "we are in one of the states of S and own tokens T" ($\mathsf{frag}(S, T)$) is actually something that one can "know and own", *i.e.*, that the set of states is closed under interference from other threads given the tokens we have.

RAs as a model of both ownership and knowledge are versatile enough to offer the user the choice whether they want to start by defining the statements one can make in the logic (and derive the transitions), or start by defining the transitions (and derive the statements).

Bringing in the state interpretation. Finally, we need to connect the abstract state of the STS to the rest of the logic through the state interpretation. This is where the "STS authority" comes into play. We

[15] If the same token was owned by two threads, that would make the overall composition of all resources $\lightning$, which is never the case.

[16] Although innocent-looking, there is something very curious about this definition: it is the only example we have of a core that is *not* a homomorphism. The core of the STS RA satisfies ra-core-mono like all cores do, but it does not satisfy the stronger, and perhaps more natural, condition $|a \cdot b| = |a| \cdot |b|$. See "The Iris 3.2 documentation" [Iri19] for a counterexample, which unfortunately is too large to fit in this margin.

define an invariant as follows:[17]

$$\mathsf{StsInv}_\gamma := \exists s.\, \boxed{\mathsf{auth}(s, \emptyset)}^{\gamma} * \varphi(s)$$

In other words, we make sure that whatever the current state s of the STS is, the corresponding state interpretation holds. Remember that "holds" here is not just a factual statement; the state interpretation can claim ownership of resources and those resources are then owned "by the protocol" and available to threads that partake in the protocol.

It is instructive to compare the shape of StsInv_γ, applied to the STS in Figure 4.1, with the actual invariant we used for that example:

$$I := (x \mapsto \mathtt{inl}(0) * \boxed{\mathsf{pending}(1/2)}^{\gamma}) \vee (\exists n.\, x \mapsto \mathtt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma})$$

Considering that a disjunction is the same as an existential quantification over a Boolean, we can see that both have the exact same structure! Intuitively, each of the two (families of) states in the STS corresponds to one of the disjuncts in the invariant, and the existential quantification over n represents the choice of n in the $\mathsf{shot}(n)$ states. Only the tokens work out somewhat differently: the STS token $[\mathsf{tok}]$ corresponds to the $\boxed{\mathsf{pending}(1/2)}^{\gamma}$ that (under the disguise of the abstract T) occurs in the precondition of `set`. The other half $\boxed{\mathsf{pending}(1/2)}^{\gamma}$ basically plays the role of the authoritative STS element, connecting the state in the invariant with the token.

To make the proof rule for STS accesses easier to state, we also define an assertion to express that we are *at least* in some state s of the STS, owning some tokens T:

$$\mathsf{StsSt}_\gamma(s, T) := \boxed{\mathsf{frag}(\uparrow(\{s\}, T), T)}^{\gamma}$$

Here we use the closure to turn "at least in state s" into the set of states that we could be in, taking into account the tokens we own. This is where we need to use the same γ as in StsInv to make sure that we are talking about the same instance of the protocol.

Finally, we can derive the following proof rule—basically a variant of hoare-inv-timeless for STSs:

hoare-sts-timeless

$$\frac{\begin{array}{c}\mathsf{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E} \qquad \mathsf{timeless}(\varphi) \\ \forall s.\, s_1 \xrightarrow{T} s \Rightarrow \{\varphi(s) * Q_1\}\, e\, \{v.\, \exists s_2, T_2.\, (s, T_1) \rightarrow (s_2, T_2) * \varphi(s_2) * Q_2\}_{\mathcal{E} \setminus \mathcal{N}}\end{array}}{\left\{\boxed{\mathsf{StsInv}_\gamma}^{\mathcal{N}} * \mathsf{StsSt}_\gamma(s_1, T_1) * Q_1\right\}\, e\, \left\{v.\, \exists s_2, T_2.\, \boxed{\mathsf{StsInv}_\gamma}^{\mathcal{N}} * \mathsf{StsSt}_\gamma(s_2, T_2) * Q_2\right\}_{\mathcal{E}}}$$

This rule shares with hoare-inv-timeless the restriction that the expression e needs to be physically atomic, and it also shares the administrative overhead: we need a mask ($\mathcal{E}$) to make sure that the same STS does not get accessed twice at the same time, and we require the state interpretation to be timeless to avoid the *later* modality.

But let us focus on the meat of this proof rule. As usual with Hoare triple rules, it is best read clockwise starting at the precondition of the conclusion. We start by knowing that the STS invariant is maintained,

[17] The γ here identifies the ghost state instance used to run this protocol; as we will see soon we need it to be able to connect StsInv with other statements about the same protocol.

having *some* view of the STS protocol telling us we are at least in state s_1 and own tokens T_1, and owning some additional resources described by Q_1. After applying the rule, we have to show a triple for every possible state the STS might be in right now, which are all the states s that *the frame* could have reached from s_1, taking into account that we own tokens T_1 (so the frame does not have access to those tokens). We also obtain the state interpretation $\varphi(s)$ of that state s, which corresponds to how we would usually obtain (temporary) ownership of an invariant I when opening it. We get to verify e under full ownership of the resources governed by the STS, but when e is done, we have to pick *some* state s_2 and corresponding token set T_2 such that (a) the transition from s to s_2 is permitted by the STS and follows the law of token preservation, and (b) we can produce (ownership of) the state interpretation of the new state. If there are any leftover resources Q_2, we are allowed to keep them. Finally, the rule says that when e is done we can continue our verification knowing that the STS is now *at least* in state s_2. This is the moment where we "forget" that the current state is *exactly* s_2, because we have to take into account that other threads might take transitions in the same protocol.

This corresponds exactly to the kind of rule that the built-in STSs of CaReSL have—showing that Iris can encode the reasoning principles provided by that logic.

4.3 One RA to rule them all

At this point, it should have become clear that in Iris there is a strong emphasis on only providing a *minimal* core logic, and deriving as much as possible *within* the logic rather than baking it in as a primitive. For example, both Hoare triples and propositions of the form $l \mapsto v$ are actually derived forms. This has the advantage that the model can be kept simpler, since it only has to justify soundness of a minimal core logic, the aforementioned *Iris base logic.*[18]

In this section we discuss the encoding of the proposition $\boxed{a}^{\gamma}$ for ghost ownership, which is not a built-in notion either. As we have seen, this proposition allows one to have *multiple* ghost locations γ, all of which can range over *different* RAs. As a primitive, Iris provides just a *single* global ghost location whose structure is described by a *single* global RA picked by the user. However, by picking an appropriate RA for that one location, we can *define* the proposition $\boxed{a}^{\gamma}$ in Iris and *derive* its rules as given in Figure 3.5 (on page 35) within the logic.

Iris's primitive construct for ghost ownership is $\mathsf{Own}\,(a)$, whose rules are given in Figure 4.2. It is worth noting that the global RA must be *unital*, which means it should have a unit element ε (Figure 3.4). The reason for this is twofold. First of all, it allows us to have the rule own-unit, which is used to prove ghost-alloc. Second of all, unital RAs enjoy the properties that the extension order $\preccurlyeq$ is reflexive and that the core $|-|$ is total, which simplify the model construction. However, the RAs used for the user-visible locations $\boxed{a}^{\gamma}$ do *not* have to be unital; we will define the global RA in a way that it has a unit even if the local RAs do not.

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

own-op

$$\mathsf{Own}\,(a \cdot b) \Leftrightarrow \mathsf{Own}\,(a) * \mathsf{Own}\,(b)$$

own-unit

$$\mathsf{True} \Rightarrow \mathsf{Own}\,(\varepsilon)$$

own-core

$$\mathsf{Own}\,(|a|) \Leftrightarrow \Box\, \mathsf{Own}\,(|a|)$$

own-valid

$$\mathsf{Own}\,(a) \Rightarrow \mathcal{V}(a)$$

own-update

$$\frac{a \rightsquigarrow B}{\mathsf{Own}\,(a) \Rrightarrow \exists b \in B.\, \mathsf{Own}\,(b)}$$

Figure 4.2: Primitive rules for ghost state.

In order to define the $\boxed{a}^{\gamma}$ connective, we need to instantiate the single global ghost state RA with a *heap* of ghost cells. To this end, we assume that we are given a family of local RAs $(M_i)_{i \in \mathcal{I}}$ for some index set $\mathcal{I}$,[19] and then we define the RA M of the global ghost state to be the indexed (dependent) product over "heaps of M_i" as follows:

$$M := \prod_{i \in \mathcal{I}} \mathbb{N} \xrightharpoonup{\mathsf{fin}} M_i$$

As shown in §4.1, RAs can be lifted pointwise through products and finite maps, so M is in fact an RA. It is even a uRA, with the unit being the product of all empty maps:[20]

$$\varepsilon := \lambda j.\, \emptyset$$

Using M as the uRA to instantiate Iris with allows us to (a) use all the M_i in our proofs, and (b) treat ghost state as a heap, where we can allocate new instances of any of the M_i at any time. We define the connective for ghost ownership of a single location as:

$$\boxed{a : M_i}^{\gamma} := \mathsf{Own}\left(\lambda j.\, \begin{cases} [\gamma \leftarrow a] & \text{if } i = j \\ \emptyset & \text{otherwise} \end{cases}\right)$$

In other words, $\boxed{a : M_i}^{\gamma}$ asserts ownership of the singleton heap $[\gamma \mapsto a]$ at position i in the product, and the empty heap for every other position in the product. We typically leave the concrete M_i implicit and write just $\boxed{a}^{\gamma}$. The rules for $\boxed{-}^{-}$ given in Figure 3.5 can now be derived from those for $\mathsf{Own}\,(-)$ shown in Figure 4.2.

Obtaining modular proofs. Even with multiple RAs at our disposal, it may still seem like we have a modularity problem: every proof is done in an instantiation of Iris with some particular global RA M, which in our case means some particular family of RAs $(M_i)_{i \in \mathcal{I}}$. As a result, if two proofs make different choices about the RAs, they are carried out in entirely different logics and hence cannot be composed.

To solve this problem, we *generalize* our proofs over the family of RAs that Iris is instantiated with. All proofs are carried out in Iris instantiated with some unknown $(M_i)_{i \in \mathcal{I}}$, showing something like "for all $(M_i)_{i \in \mathcal{I}}$, … ". If the proof needs a particular RA, it further assumes that there exists

[19] In our Coq formalization, the index set must be finite. This is needed to avoid classical axioms.

[20] We write n-ary products as functions from the index to the value at that position. This exploits the correspondence with dependently typed functions in type theory.

some j such that M_j is the desired RA. Composing two proofs is thus straightforward; the resulting proof works in any family of RAs that contains all the particular RAs needed by either proof. Finally, if we want to obtain a “closed form” of some particular proof in a concrete instance of Iris, we simply construct a family of RAs that contains only those that the proof needs.

Notice that the generalization over resource algebras happens at the *meta-level* here, so the set of available resource algebras must be fixed before the proof begins. If, for example, the *type* of the ghost state should depend on some program value determined only at run-time, all possible types would already have to be in $(M_i)_{i \in \mathcal{I}}$. We have not yet explored to what extent our approach is compatible with this situation, as it does not seem to arise in practice.

4.4 Authoritative ghost state

The most frequently used resource algebra construction in Iris is the *authoritative RA*. It is useful whenever a library maintains some piece of state in an Iris invariant, and wishes to reflect that state as ghost state that can be *owned*, with a library-defined notion of *how* ownership can be “separated”. In particular, the authoritative RA in Iris can be used to implement the idea of “fictional separation”.[21] This generalizes what we did in §3, and can be seen as an alternative to STSs (§4.2).

As a motivating example, we will explain how a specific instance of the authoritative RA can be used to define *fractional ownership* $\ell \overset{q}{\mapsto} v$ of a heap location starting from just full ownership with the standard points-to connective $\ell \mapsto v$ (§4.4.1). This is an example of fictional separation: $\ell \overset{q_1}{\mapsto} v * \ell \overset{q_2}{\mapsto} v$ provides the fiction that the underlying $\ell \mapsto v$ has been disjointly split into two pieces. In §4.4.2, we will generalize the underlying RA construction to the general authoritative RA.

4.4.1 Derived fractional heap

Concretely, our goal in this subsection is to define a library that provides the following specification for a *fractional heap*:[22]

fheap-sep
$$\ell \overset{q_1+q_2}{\longmapsto} v \iff \ell \overset{q_1}{\mapsto} v * \ell \overset{q_2}{\mapsto} v$$

fheap-agree
$$\ell \overset{q_1}{\mapsto} v * \ell \overset{q_2}{\mapsto} w \Rightarrow v = w$$

fheap-valid
$$\ell \overset{q}{\mapsto} v \Rightarrow q \leq 1$$

fheap-alloc
$$\{\mathsf{True}\}\, \texttt{ref}(v)\, \{\ell.\, \ell \overset{1}{\mapsto} v\}$$

fheap-load
$$\{\ell \overset{q}{\mapsto} v\}\ !\,\ell\, \{w.\, w = v * \ell \overset{q}{\mapsto} v\}$$

fheap-store
$$\{\ell \overset{1}{\mapsto} v\}\ \ell \leftarrow w\ \{\ell \overset{1}{\mapsto} w\}$$

Here, all q are implicitly (strictly) positive. This specification formalizes the idea described in §2.3: ownership of $\ell \overset{q}{\mapsto} v$ can be split into arbitrarily many pieces that still all “sum up” to 1 (fheap-sep). Ownership of any fraction of a location is sufficient for loading from that location (fheap-load), but *full* ownership of the location is required for a store (fheap-store).

[21] Dinsdale-Young, Gardner, and Wheelhouse, “Abstraction and refinement for local reasoning”, 2010 [DGW10].

[22] Technically, the library also requires a globally available invariant and a ghost name to tie invariant and ownership together. We omit those details for now to keep the discussion focused on the authoritative RA; at the end of the section we will show the precise specification.

Following the Iris approach, we do not want to bake this specification into the basic axioms that are available for our language; instead, we will show that the standard heap triples already presented in Figure 3.2 on page 27 are sufficient to *derive* the fractional heap specification. Even though the underlying heap has unsplittable locations, we can present to the user the *fiction* of being able to split them. The plan, roughly, is for the library to maintain ownership of the "real" point-to connective in an invariant, and then use the authoritative RA to let clients own some piece of the state tracked by that invariant.

A resource algebra for authorities. The key ingredient to this construction is the following resource algebra for an *authoritative heap*, based on *FHeap* as introduced in §4.1:

$$\begin{aligned} \mathit{FHeap} &:= \mathit{Loc} \overset{\mathsf{fin}}{\rightharpoonup} \mathit{Frac} \times \mathit{Ag}_0(\mathit{Val}) \\ M &:= \mathsf{auth}(a : \mathit{FHeap}, f : \mathit{FHeap}) \mid \mathsf{frag}(f : \mathit{FHeap}) \mid \lightning \end{aligned}$$

The RA comes with two kinds of resources:

- The *fragments* frag represent ownership of some part of the heap, *i.e.*, ownership of some fraction of some location(s). As we will see, these elements behave (in terms of their compositions and core) just like elements of *FHeap*.

  These resources will be used to give meaning to $\ell \overset{q}{\mapsto} v$ in the specification above. In other words, these resources are used by the *clients* of the library to represent ownership and knowledge of some part of the library's state.

- The *authoritative resource* auth represents ownership of the *entire heap* at once.[23] Crucially, as we will see, the authoritative heap is *kept in sync* with the fragments that are handed out to clients. Only one authoritative resource can exist; its ownership is exclusive.

  In the fractional heap library, the authoritative element will be owned by the library itself. An invariant will tie the heap in the authoritative resource to the underlying "physical" heap (described by the rules in Figure 3.2). This is basically the same underlying pattern as what we already saw for connecting an STS with its state interpretation (§4.2).

  Matching these intuitions, resource composition is defined as follows:

$$\begin{aligned} \mathsf{frag}(f_1) \cdot \mathsf{frag}(f_2) &:= \mathsf{frag}(f_1 \cdot f_2) \\ \mathsf{frag}(f_1) \cdot \mathsf{auth}(a, f_2) &:= \mathsf{auth}(a, f_1 \cdot f_2) \\ \mathsf{auth}(a, f_1) \cdot \mathsf{frag}(f_2) &:= \mathsf{auth}(a, f_1 \cdot f_2) \end{aligned}$$

Notice, in particular, that there is no way to compose two authoritative resources auth! This reflects their exclusive nature, similar to *Ex* (§4.1) and the auth of an STS. For frag, on the other hand, we use the composition operation from *FHeap*. This definition also explains why auth carries *both* an authoritative heap a and a fragment f: we have to somehow define what happens when an auth is composed with a frag without losing track of which fragment we own in addition to the authoritative part.

[23] The authoritative resource also contains a fragment f, for much the same reason that the authoritative part of an STS also contains a set of tokens. We will get back to this point shortly.

The core strips away the exclusive authoritative parts and otherwise uses the core of *FHeap*:

$$\begin{aligned} |\mathsf{frag}(f)| &:= \mathsf{frag}(|f|) \\ |\mathsf{auth}(a, f)| &:= \mathsf{frag}(|f|) \end{aligned}$$

So far, the heap in the authoritative resources and the fragments are entirely unconnected. Their relationship is established by the *validity predicate*:

$$\begin{aligned} \mathcal{V}(\mathsf{frag}(f)) &:= \mathcal{V}(f) \\ \mathcal{V}(\mathsf{auth}(a, f)) &:= \mathcal{V}(a) \land f \preccurlyeq a \end{aligned}$$

In other words, the composition of an authoritative heap a and a fragment f is only valid if f is *included* in a, written $f \preccurlyeq a$.[24] This key property implies that, if we own $\mathsf{frag}(f)$, we can be sure that whatever the authoritative a is, it is an "extension" of f—since all resources combined must still be valid, we have that $f \preccurlyeq a$.[25] In particular, if f says that some location ℓ has value v, we know that a ascribes the same value to ℓ:

$$f(\ell) = (\_, \mathsf{ag}_0(v)) \land \mathcal{V}(\mathsf{auth}(a, f)) \Rightarrow a(\ell) = (\_, \mathsf{ag}_0(v)) \qquad (\textsc{fheap-auth-ag0})$$

To see why this is true, notice that the definition of $\mathcal{V}$ for auth provides $\mathcal{V}(a)$ and $f \preccurlyeq a$. Thus $a = f \cdot f'$. We distinguish two cases:

- $\ell \notin \mathsf{dom}(f')$: then $a(\ell) = f(\ell) = (\_, \mathsf{ag}_0(v))$, and we are done.
- $\ell \in \mathsf{dom}(f')$: then $a(\ell) = f(\ell) \cdot f'(\ell) = (\_, \mathsf{ag}_0(v)) \cdot f'(\ell)$. Since we also know $\mathcal{V}(a(\ell))$, we have $\mathcal{V}(f'(\ell))$ (ra-valid-op) and thus $f'(\ell) = (\_, \mathsf{ag}_0(v'))$ for some v' (ag0-uninj on page 44). In fact, by ag0-agree we get that $v' = v$, and thus $a(\ell) = (\_, \mathsf{ag}_0(v))$, finishing the proof.

This lemma will be crucial when verifying the fheap rules, because it lets us draw a connection between ownership of a fragment (which is pure ghost state without any inherent "meaning") and the current physical state of the heap.

Invariants for fictional separation. With our RA in place, we can finally define the key invariant of the fractional heap library:

$$\begin{aligned} \mathsf{to\_fheap}(h) &:= (\lambda v.\, (1, \mathsf{ag}_0(v)))\ \langle\$\rangle\ h \\ \mathsf{FHeapInv} &:= \exists h : \mathit{Loc} \xrightarrow{\mathsf{fin}} \mathit{Val}. \\ &\qquad \boxed{\mathsf{auth}(\mathsf{to\_fheap}(h), \emptyset)}^{\gamma} * \mathop{\Huge *}_{[\ell \leftarrow v] \in h} \ell \mapsto v \end{aligned}$$

Here, we use $f\ \langle\$\rangle\ h$ as Haskell-style notation for "mapping f over h", *i.e.*, for applying f pointwise in h. We have to do this to convert h from $\mathit{Loc} \xrightarrow{\mathsf{fin}} \mathit{Val}$ to *FHeap*.

The invariant maintains that there always exists an h that ties together two resources:

- On the one hand, h matches the authoritative resource of ghost state γ.

[24] *FHeap* is unital, so $\preccurlyeq$ is reflexive on this RA.

[25] Note the analogy with STSs, where owning $\mathsf{frag}(S, T)$ means we can be sure that whatever the true (authoritative) current state of the STS is, it is contained in the set S. In fact, it could be possible to define the STS RA on top of the authoritative RA, but that is not something we have explored.

- On the other hand, every location in h is also physically owned by this invariant, meaning that in the physical heap, location ℓ contains value $h(\ell)$.

This means that the authoritative resource of γ is always in sync with the actual physical heap, in very much the same way that our one-shot ghost location in §3 was kept in sync with the physical value of x. Together with the fact that the *fragments* of an authoritative RA are kept in sync with the authoritative resource, this means that we can use the fragments to keep track of the physical state! This is best demonstrated by looking at the proof of fheap-load.

But first, we need to *define* $\ell \overset{q}{\mapsto} v$. Because we are implementing fractional points-to as a library inside Iris, we have to give meaning to the fractional points-to assertion in terms of lower-level Iris connectives. As mentioned before, our library uses ownership of a *fragment* of an authoritative RA to model fractional points-to:

$$\ell \overset{q}{\mapsto} v := \boxed{\mathsf{frag}([\ell \leftarrow (q, \mathsf{ag}_0(v))])}^{\gamma}$$

Proof of fheap-load. Now we can show that with this definition, and assuming FHeapInv is maintained as an invariant, we can derive fheap-load. Being fully explicit about invariants and namespaces, we want to show the following triple:

$$\boxed{\mathsf{FHeapInv}}^{\mathcal{N}} \vdash \left\{\boxed{\mathsf{frag}([\ell \leftarrow (q, \mathsf{ag}_0(v))])}^{\gamma}\right\} \; !\,\ell \; \left\{w.\, w = v * \boxed{\mathsf{frag}([\ell \leftarrow (q, \mathsf{ag}_0(v))])}^{\gamma}\right\}_{\mathcal{N}}$$

To perform this proof, we start by using hoare-inv-timeless to open the invariant and get access to FHeapInv. After using ghost-op to merge the two parts of γ, our current resources at this point are:

$$\boxed{\mathsf{auth}(\mathsf{to\_fheap}(h), [\ell \leftarrow (q, \mathsf{ag}_0(v))])}^{\gamma} * \mathop{\Large\ast}_{[\ell' \leftarrow v'] \in h} \ell' \mapsto v'$$

Note how the fragment we obtained via our precondition moved into the second component of auth. Now we use fheap-auth-ag0 to learn that $\mathsf{to\_fheap}(h)(\ell) = (\_, \mathsf{ag}_0(v))$. Some simple reasoning about the mapping operation $\langle\$\rangle$ and ag_0 shows that this implies $h(\ell) = v$. (This is where we exploit that the authoritative RA keeps the fragments and the authoritative resource in sync.)

Next, we use what we learned about h to conclude that, thanks to the second component of FHeapInv, we actually *own* $\ell \mapsto v$. (This is where we exploit that FHeapInv keeps the authoritative resource and the heap in sync.) In other words, thanks to the authoritative RA and the way FHeapInv is set up, we were able to start with ownership of the ghost resource $\boxed{\mathsf{frag}([\ell \leftarrow (q, \mathsf{ag}_0(v))])}^{\gamma}$ that has no physical meaning attached to it, and end up with actually owning the physical right to access ℓ!

Now all we have to do us use hoare-load, proving that we may actually execute $!\,\ell$ and that this will return v. We finish the proof by re-establishing FHeapInv (which is trivial because we did not change anything).

Proof of FHEAP-STORE. The proof of the store rule starts very similar to FHEAP-LOAD: we open the invariant, and use FHEAP-AUTH-AG0 to show that we actually own location l. However, completing the store operation will actually *change* the value stored in l, so we cannot just close the invariant again!

After the store operation, we are left with the following resources:

$$\boxed{\mathsf{auth}(\mathsf{to\_fheap}(h), [\ell \leftarrow (1, \mathsf{ag}_0(v))])}^{\gamma} * \ell \mapsto w * \mathop{\Huge *}_{[\ell' \leftarrow v'] \in h \setminus \{\ell\}} \ell' \mapsto v'$$

In other words, for locations in $h \setminus \{\ell\}$, the physical heap and h are still in agreement, but ℓ points to w.

The goal is to close the invariant with the new heap $h' := h[\ell \leftarrow w]$. But for this, we will need to update the ghost state with a frame-preserving update. This makes a lot of sense: we altered the physical state, so of course we have to alter the ghost state as well to keep the two in sync. Concretely, we need the following update:

$$\mathsf{auth}(\mathsf{to\_fheap}(h), [\ell \leftarrow (1, \mathsf{ag}_0(v))]) \rightsquigarrow \mathsf{auth}(\mathsf{to\_fheap}(h'), [\ell \leftarrow (1, \mathsf{ag}_0(w))])$$

Two things are changed at the same time by this update: the authoritative heap changes to h', and the value of the fragment changes from v to w. This keeps the authoritative resource and the fragment in sync.

To see why this is valid, let us consider what possible frames $c^?$ are compatible with $\mathsf{auth}(\mathsf{to\_fheap}(h), [\ell \leftarrow (1, \mathsf{ag}_0(v))])$. First of all, $c^?$ cannot be an auth as composing two of those is never valid. Secondly, if $c^?$ is $\mathsf{frag}(f)$ for some f, then $\ell \notin \mathsf{dom}(f)$ because otherwise, $f(\ell) \cdot (1, \mathsf{ag}_0(v))$ is invalid—this is based on the fact that *nothing* is compatible with the full fraction 1 (FRAC-EXCLUSIVE). Thus, the frame cannot make any statement about ℓ, and we are free to change its value to w.[26]

After the frame-preserving update is done, we can establish $\mathsf{FHeapInv}$ with our new heap h'. We keep $\mathsf{frag}([\ell \leftarrow (1, \mathsf{ag}_0(w))])$, which exactly matches the postcondition of FHEAP-STORE, and so we are done.

The remaining properties of fractional heaps are easy to show:

- FHEAP-ALLOC follows via HOARE-ALLOC, and then adding the freshly allocated ℓ to the h in $\mathsf{FHeapInv}$. Crucially, we know that $\ell \notin h$ because the invariant owns all locations in h, and we already own ℓ because we just allocated it (this argument relies on HEAP-EXCLUSIVE).
- FHEAP-VALID is a direct consequence of GHOST-VALID and how validity of *Frac* is defined. Note that with the original definition of *Frac* (the one with $\mathsf{frac}(q)$ where $q \leq 1$), this rule would be rather nonsensical: to even *state* an assertion like $\ell \overset{q}{\mapsto} v$ would require proving that $q \leq 1$! This turns out to be very inconvenient to work with in practice, which is why we settled for the alternative definition where the assertion can be *stated* with any positive q, but can only *hold* when $q \leq 1$.
- FHEAP-AGREE follows from GHOST-VALID and AG0-AGREE.
- FHEAP-SEP follows from GHOST-OP.

[26] Later in this section, we will see how we can replace this ad-hoc proof of a frame-preserving update by the composition of some general principles.

Fully formal specification. The actual specification that we ended up proving does not exactly match the one we gave at the beginning of this subsection: if we want to be fully precise, we have to account for the fact that $\ell \overset{q}{\mapsto} v$ needs to know about the ghost name γ that we used throughout the proof, so we will write $\ell \overset{q}{\mapsto}_\gamma v$ instead. Moreover, to prove the Hoare triples above we had to assume $\boxed{\mathsf{FHeapInv}_\gamma}^{\mathcal{N}}$. Finally, for completeness' sake, we also state explicitly that $\ell \overset{q}{\mapsto}_\gamma v$ is timeless.

fheap-init'
$$\mathsf{True} \Rrightarrow \exists \gamma.\, \boxed{\mathsf{FHeapInv}_\gamma}^{\mathcal{N}}$$

fheap-sep'
$$\ell \overset{q_1+q_2}{\longmapsto}_\gamma v \iff \ell \overset{q_1}{\mapsto}_\gamma v * \ell \overset{q_2}{\mapsto}_\gamma v$$

fheap-agree'
$$\ell \overset{q_1}{\mapsto}_\gamma v * \ell \overset{q_2}{\mapsto}_\gamma w \Rightarrow v = w$$

fheap-valid'
$$\ell \overset{q}{\mapsto}_\gamma v \Rightarrow q \leq 1$$

fheap-timeless'
$$\mathsf{timeless}(\ell \overset{q}{\mapsto}_\gamma v)$$

fheap-alloc'
$$\boxed{\mathsf{FHeapInv}_\gamma}^{\mathcal{N}} \vdash \{\mathsf{True}\}\, \texttt{ref}(v)\, \{\ell.\, \ell \overset{1}{\mapsto}_\gamma v\}_{\mathcal{N}}$$

fheap-load'
$$\boxed{\mathsf{FHeapInv}_\gamma}^{\mathcal{N}} \vdash \{\ell \overset{q}{\mapsto}_\gamma v\}\, !\,\ell\, \{w.\, w = v * \ell \overset{q}{\mapsto}_\gamma v\}_{\mathcal{N}}$$

fheap-store'
$$\boxed{\mathsf{FHeapInv}_\gamma}^{\mathcal{N}} \vdash \{\ell \overset{1}{\mapsto}_\gamma v\}\, \ell \leftarrow w\, \{\ell \overset{1}{\mapsto}_\gamma w\}_{\mathcal{N}}$$

When working in Coq, we can hide the dependency on γ through a typeclass—akin to assuming on paper that γ is globally known. However, the invariant $\boxed{\mathsf{FHeapInv}_\gamma}^{\mathcal{N}}$ has to be mentioned explicitly in Coq.[27] On paper, we also make that implicit, which is justified by the fact that (a) the invariant can be trivially established (fheap-init') and (b) invariants are persistent and can thus be duplicated arbitrarily often. Together, this means we can pretend to be working with the specification from the beginning of this subsection.

4.4.2 General rules for authoritative ghost state

The RA we used in the previous subsection was an ad-hoc RA specifically designed for fractional heaps. However, it turns out that this can be easily generalized to an authoritative variant of *any* uRA M by defining an RA combinator as follows:[28]

$$\begin{aligned}
\mathit{Auth}(M) &:= \mathsf{auth}(a : \mathit{Ex}(|M|)^?, f : M) \\
\mathsf{auth}(a_1, f_1) \cdot \mathsf{auth}(a_2, f_2) &:= \mathsf{auth}(a_1 \cdot a_2, f_1 \cdot f_2) \\
\mathcal{V}(\mathsf{auth}(a, f)) &:= \begin{cases} \mathcal{V}(f) & \text{if } a = \bot \\ \mathcal{V}(a) \land f \preccurlyeq a & \text{otherwise} \end{cases} \\
|\mathsf{auth}(a, f)| &:= \mathsf{auth}(\bot, |f|)
\end{aligned}$$

$\mathit{Auth}(M)$ is basically the same RA as $\mathit{Ex}(|M|)^? \times M$, with one exception: validity of $\mathit{Auth}(M)$ demands that the authoritative element (if present) must include the fragment.[29] As we have seen with fheap-auth-ag0, this is the key property that binds the fragments and the authoritative elements together.

This is conceptually very close to the previous definition in §4.4.1, except that we reuse the exclusive RA Ex (§4.1) to model the exclusive nature of authoritative elements, and instead of having two constructors (auth and frag) we just have one, and make the presence of the authoritative element optional. Up to these minor differences, we can recover the RA of the fractional heap library via $\mathit{Auth}(\mathit{FHeap})$.

[27] In fact, in Coq we build fractions into the lowest-level points-to assertion precisely to avoid this extra invariant. The same pattern still arises elsewhere though: for example, the lifetime logic (§11) relies on an invariant that we keep implicit on paper but have to mention in Coq.

[28] Note that we do require M to have a unit; if the RA we would like to use is not unital, we can always just add a unit element.

[29] It seems plausible that this could be defined using a generic "subset" construction based on restricting validity.

As before, the intuition is that the authoritative element can only be owned *once* because it is exclusive, while the fragments behave exactly like elements of M. In fact, talking about these two kinds of elements is so common that we define some syntactic sugar for them:[30]

$$\begin{aligned} \bullet a &:= \mathsf{auth}(\mathsf{ex}(a), \varepsilon) \\ \circ f &:= \mathsf{auth}(\bot, f) \end{aligned}$$

Notice how $\circ f$ corresponds to $\mathsf{frag}(f)$ from §4.4.1: this is ownership of a fragment without also owning the authoritative element. In defining $\bullet a$, we exploit the fact that M has a unit.

Local updates. To implement the fractional heap, we needed some frame-preserving updates for the authoritative RA: we had to change the value of a location in the authoritative heap (for FHEAP-STORE), or to add a new location to the authoritative heap (for FHEAP-ALLOC). Like the other RAs in §4.1, the authoritative RA comes with some frame-preserving updates proven once and for all. However, for frame-preserving updates on *Auth*(M), it turns out that which updates are possible heavily depends on M. This should not be surprising—after all, M defines how the fragments of *Auth*(M) can be (de)composed. The more surprising part is that, as we will see, these updates can be very different from the frame-preserving updates of M itself.

To state a general frame-preserving update for *Auth*(M), we introduce the notion of a *local update* on M:

$$(a, f) \rightsquigarrow_{\mathsf{L}} (a', f') := \forall c^? \in M^?.\, \mathcal{V}(a) \wedge a = f \cdot c^? \Rightarrow \mathcal{V}(a') \wedge a' = f' \cdot c^?$$

The way to think about this as follows: $(a, f) \rightsquigarrow_{\mathsf{L}} (a', f')$ states that we can do an update from authoritative state a to authoritative state a' if we also own the fragment f, which in this process is replaced by f'. The quantified $c^?$ in the update is the composition of all the other fragments that we do *not* own; they have to stay the same.

This is basically exactly what a frame-preserving update for *Auth*(M) needs, so it is not very hard to show the following rule:

$$\textsc{auth-update}\quad \frac{(a, f) \rightsquigarrow_{\mathsf{L}} (a', f')}{\bullet a \cdot \circ f \rightsquigarrow \bullet a' \cdot \circ f'}$$

The actually interesting question, of course, is which local updates we can show for the various RAs and RA combinators that we defined so far.

Local updates for maps and exclusive elements. Here are the local updates we need for the fractional heap:

$$\textsc{fmap-update-local}\quad \frac{(a, b) \rightsquigarrow_{\mathsf{L}} (a', b')}{(f[i \leftarrow a], g[i \leftarrow b]) \rightsquigarrow_{\mathsf{L}} (f[i \leftarrow a'], g[i \leftarrow b'])}$$

$$\textsc{exclusive-update-local}\quad \frac{\mathsf{exclusive}(f) \qquad \mathcal{V}(a')}{(a, f) \rightsquigarrow_{\mathsf{L}} (a', a')} \qquad\qquad \textsc{fmap-alloc-local}\quad \frac{i \notin \mathsf{dom}(f) \qquad \mathcal{V}(a')}{(f, \emptyset) \rightsquigarrow_{\mathsf{L}} (f[i \leftarrow a'], [i \leftarrow a'])}$$

[30] For the rare case that we own both an authoritative element and a fragment, we just write $\bullet a \cdot \circ f$.

fmap-update-local just says that finite maps propagate the local updates pointwise, similar to fmap-update. exclusive-update-local corresponds to exclusive-update, and together with the previous rule this is what we need when verifying fheap-store: the fragment we own at location ℓ (namely, $(1, \mathsf{ag}_0(v))$) is exclusive, and thus can be replaced by anything we like.

fmap-alloc-local is what we need for fheap-alloc. It is basically the local-update version of fmap-alloc, but there is one key difference: fmap-alloc is a non-deterministic update where some fresh index i gets picked for us, while fmap-alloc-local lets us *pick i ourselves.* This extra power comes from the fact that fmap-alloc has to work with any possible frame, and the fresh i is chosen such that the index has not been used by *anyone* yet. In contrast, when using fmap-alloc-local we *know* that f is the full map containing everything owned by anyone, and thus any index $i \notin \mathsf{dom}(f)$ is guaranteed to be unused. This freedom to pick i ourselves was crucial for fheap-alloc, where a fresh ℓ was chosen by the operational semantics, and now we have to allocate exactly that ℓ in our ghost heap to make sure they remain in sync.

Local updates for sums and products. For completeness' sake, we give the rules for local updates for the other RAs introduced in §4.1.

inl-update-local
$$\frac{(a_1, f_1) \rightsquigarrow_{\mathsf{L}} (a'_1, f'_1)}{(\mathsf{inl}(a_1), \mathsf{inl}(f_1)) \rightsquigarrow_{\mathsf{L}} (\mathsf{inl}(a'_1), \mathsf{inl}(f'_1))}$$

inr-update-local
$$\frac{(a_2, f_2) \rightsquigarrow_{\mathsf{L}} (a'_2, f'_2)}{(\mathsf{inr}(a_2), \mathsf{inr}(f_2)) \rightsquigarrow_{\mathsf{L}} (\mathsf{inr}(a'_2), \mathsf{inr}(f'_2))}$$

prod-update-local
$$\frac{(a_i, f_i) \rightsquigarrow_{\mathsf{L}} (a'_i, f'_i)}{((a_1, \dots, a_n), (f_1, \dots, f_n)) \rightsquigarrow_{\mathsf{L}} ((a_1, \dots, a'_i, \dots, a_n), (f_1, \dots, f'_i, \dots, f_n))}$$

Cancellativity. The original Iris paper[31] did not use the notion of local updates; instead, we gave the following rule for frame-preserving updates of the authoritative RA (or rather the authoritative monoid; resource algebras were only introduced later):

auth-update-cancellative
$$\frac{\mathsf{cancellative}(M) \qquad \mathcal{V}(f' \cdot c)}{\bullet(f \cdot c) \cdot \circ f \rightsquigarrow \bullet(f' \cdot c) \cdot \circ f'}$$

An RA M is *cancellative* if the following holds:

$$\forall a, b, c.\, a \cdot b = a \cdot c \Rightarrow b = c$$

This rule still holds (and can be expressed as a local update), but has been superseded by monoid-specific rules for local updates like the ones introduced above. One reason for this is that we sometimes work with non-cancellative monoids (such as natural numbers with composition taking the maximum of the two operands). Another reason is that auth-update-cancellative is not very well-suited for use in a proof assistant: we can only use it when our local resources are of the form $\bullet(f \cdot c) \cdot \circ f$, and fitting resources into that shape can be non-trivial (*e.g.*, during the proof of fheap-store).

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

Chapter 5
Invariants and modalities

In this chapter, we will take a closer look at invariants and masks. First we will show the rules for *general* invariants without the restriction to timeless propositions (§5.1). Then we will show how to encode a form of invariants that comes with a *cancellation* mechanism, providing a way to *temporarily* manage some resources in an invariant and re-claiming full ownership later (§5.2). This will motivate an extension of the notion of view shifts that we have seen so far to *mask-changing* view shifts (§5.3). We will explain why it is useful to view view shifts as a *modality* instead of a binary, implication-like connective (§5.4 and §5.5). We will also briefly talk about a reoccurring pattern in Iris that arises frequently when specifying libraries like cancellable invariants, dubbed *accessors* (§5.6). Finally, we will collect in Figure 5.6 all the Iris proof rules that form the foundation for the rest of this dissertation.

5.1 General invariants and the later *modality*

In §3.2, we used a restricted form of the invariant rules HOARE-INV and VS-INV that requires the invariant to be *timeless*. This rules out taking some arbitrary proposition P and putting it inside an invariant, which is sometimes needed. In that case, the stronger rules in Figure 5.1 can be used.[1]

These rules use the *later* modality[2] ($\triangleright$) to side-step the unsoundness that would arise with entirely unrestricted impredicative invariants.[3] After opening the invariant, instead of P we only obtain the weaker $\triangleright P$ (pronounced "later P"). The simpler rules we saw before can be derived via VS-TIMELESS, which lets us strip away a $\triangleright$ in front of a timeless proposition. The only other way to get rid of a later is to *take a step* in the program, which is reflected by HOARE-STEP: this variant of the frame rule lets us remove a later from some framed proposition R if we are framing around a computation that actually performs a step of execution, *i.e.*, if e is not a value.

Note that the extra $\triangleright$ in INV-ALLOC actually makes the rule *stronger* because, by $\triangleright$-INTRO, *adding* a later modality to a proposition is always permitted.

[1] Notably, P can really be *any* proposition. CSL originally required resource invariants to be *precise* (*i.e.*, in any given heap, there can be only one sub-heap that satisfies the invariant), but this was only needed to justify the conjunction rule—and in Iris, that rule is already broken by frame-preserving updates. Also see the discussion of *cancellativity* in §7.2.

[2] Appel *et al.*, "A very modal model of a modern, major, general type system", 2007 [App+07].

[3] Jung *et al.*, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b], §8.2.

INV-ALLOC

$$\triangleright P \Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}$$

VS-INV

$$\frac{\triangleright P * Q_1 \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}} \triangleright P * Q_2 \qquad \mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} * Q_1 \Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}} * Q_2}$$

▷-INTRO

$$P \vdash \triangleright P$$

▷-MONO

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

VS-TIMELESS

$$\frac{\mathsf{timeless}(P)}{\triangleright P \Rrightarrow_{\mathcal{E}} P}$$

▷ commutes around □, ∨, ∧, ∗, ∀,
and ∃ with non-empty domain

HOARE-INV

$$\frac{\{\triangleright P * Q_1\}\, e\, \{v.\, \triangleright P * Q_2\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathsf{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E}}{\left\{\boxed{P}^{\mathcal{N}} * Q_1\right\} e \left\{v.\, \boxed{P}^{\mathcal{N}} * Q_2\right\}_{\mathcal{E}}}$$

HOARE-STEP

$$\frac{\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}} \qquad e \text{ is not a value}}{\{P * \triangleright R\}\, e\, \{v.\, Q * R\}_{\mathcal{E}}}$$

Figure 5.1: Rules for invariants and the later modality.

5.2 Cancellable invariants

Iris invariants are *persistent*, which means that once established, they are maintained forever. This can be very useful as we have seen in §3. Sometimes, however, some resources have to be put under the control of a shared invariant *temporarily*, and later, the protocol is "torn down" again.[4] To this end, we can equip invariants with a fractional *token* (similar to the fractional points-to assertion from §4.4) that is split among all parties that wish to access the invariant: owning any fraction of the token is sufficient to access the invariant, but *canceling* the invariant requires full ownership of the entire token.

The logical interface of cancellable invariants is given in Figure 5.2. CINV-ALLOC allocates a new cancellable invariant. Here, $\mathsf{CInv}^{\gamma,\mathcal{N}}(P)$ expresses existence of a cancellable invariant for proposition P. Like normal invariants, cancellable invariants are allocated into some namespace $\mathcal{N}$ that is used to ensure that one invariant is not opened twice at the same time. On top of that, they carry a ghost identifier γ which ties the token and the invariant together: $[\mathsf{CInv} : \gamma]_1$ expresses full ownership of the token for the cancellable invariant with identifier γ. That token can be split according to CINV-SPLIT, with the fraction q tracking "how much" of the total token is owned (similar to what we already saw with $\mathsf{pending}(q)$ in §3 or $\ell \overset{q}{\mapsto} v$ in §4.4). CINV-VALID says that it is impossible to own more than the full fraction of the token.

As usual with these fractions, the absolute value of q does not really matter (there is no discernible difference between owning a half or a third of the full token), but what *does* matter is that all the pieces sum up to exactly 1: only ownership of the *full* token is sufficient to apply CINV-CANCEL, which cancels the invariant and re-gains full ownership of P (well, $\triangleright P$).

Finally, VS-CINV and HOARE-CINV can be used to access cancellable invariants in very much the same way as VS-INV and HOARE-CINV are used for normal invariants. The key difference is that accessing a cancellable invariant requires *some* fraction of the token for that invariant. This is

[4] This situation arises frequently when reasoning about programs written in the style of structured parallelism, *i.e.*, with parallel composition: in such a setting, it makes sense for invariants to also be structural so that when all threads are joined together, resources are all exclusively owned again. Correspondingly, in logics like FCSL [Nan+14] that are designed for structural parallelism, the built-in invariants are cancellable. As usual, Iris provides a similar mechanism as a derived form.

cinv-timeless

$$\mathsf{timeless}(\left[\mathsf{CInv} : \gamma\right]_q)$$

cinv-persistent

$$\mathsf{persistent}(\mathsf{CInv}^{\gamma,\mathcal{N}}(P))$$

cinv-split

$$\left[\mathsf{CInv} : \gamma\right]_{q_1+q_2} \Leftrightarrow \left[\mathsf{CInv} : \gamma\right]_{q_1} * \left[\mathsf{CInv} : \gamma\right]_{q_2}$$

cinv-valid

$$\left[\mathsf{CInv} : \gamma\right]_q \Rightarrow q \leq 1$$

cinv-alloc

$$\triangleright P \Rrightarrow_{\mathcal{E}} \exists \gamma.\, \left[\mathsf{CInv} : \gamma\right]_1 * \mathsf{CInv}^{\gamma,\mathcal{N}}(P)$$

cinv-cancel

$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\mathsf{CInv}^{\gamma,\mathcal{N}}(P) * \left[\mathsf{CInv} : \gamma\right]_1 \Rrightarrow_{\mathcal{E}} \triangleright P}$$

vs-cinv

$$\frac{\triangleright P * Q_1 \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}} \triangleright P * Q_2 \qquad \mathcal{N} \subseteq \mathcal{E}}{\mathsf{CInv}^{\gamma,\mathcal{N}}(P) * \left[\mathsf{CInv} : \gamma\right]_q * Q_1 \Rrightarrow_{\mathcal{E}} \mathsf{CInv}^{\gamma,\mathcal{N}}(P) * \left[\mathsf{CInv} : \gamma\right]_q * Q_2}$$

hoare-cinv

$$\frac{\{\triangleright P * Q_1\}\, e\, \{\triangleright P * Q_2\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathsf{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E}}{\left\{\mathsf{CInv}^{\gamma,\mathcal{N}}(P) * \left[\mathsf{CInv} : \gamma\right]_q * Q_1\right\} e \left\{\mathsf{CInv}^{\gamma,\mathcal{N}}(P) * \left[\mathsf{CInv} : \gamma\right]_q * Q_2\right\}_{\mathcal{E}}}$$

Figure 5.2: Rules for cancellable invariants.

necessary to make sure that the invariant cannot be accessed any more after cinv-cancel was used.

Implementing cancellable invariants. Deriving these proof rules in Iris is fairly straightforward. In terms of ghost state, we will use *Frac* to represent the tokens. Then we define:

$$\left[\mathsf{CInv} : \gamma\right]_q := \boxed{q : \mathit{Frac}}^{\gamma} \qquad \mathsf{CInv}^{\gamma,\mathcal{N}}(P) := \boxed{P \vee \boxed{1 : \mathit{Frac}}^{\gamma}}^{\mathcal{N}}$$

With this, cinv-split and cinv-valid are trivial. cinv-alloc is also easy, using ghost-alloc and inv-alloc. For cinv-cancel, we use vs-inv to open the invariant and perform a case distinction on the disjunction $\triangleright P \vee \triangleright \boxed{1 : \mathit{Frac}}^{\gamma}$.[5] We cannot be in the second case, as we own the full token ourselves and owning it twice leads to a contradiction.[6] Thus we have $\triangleright P$. We close the invariant again by depositing our $\boxed{1 : \mathit{Frac}}^{\gamma}$. Finally, for vs-cinv and hoare-cinv, we use the corresponding rule to open the underlying invariant and again exclude the second disjunct similar to cinv-cancel, based on the fact that we own *some* fraction of the token so it is impossible for the *entire* token to be inside the invariant.

A more complex variant of cancellable invariants will be the heart of how we model Rust-style borrowing and lifetimes in §11. But for now, we will focus on something else: one dissatisfying aspect of the specification above is that we had to *separately* prove vs-cinv and hoare-cinv, and we were doing basically the same reasoning both times. It would be much more satisfying to be able to provide a single proof rule that can be used *both* to open cancellable invariants around view shifts, and around Hoare triples (and around anything else that might support opening invariants around it, like logically atomic triples[7]). For this purpose, Iris provides support for *mask-changing view shifts.*

[5] Remember that $\triangleright$ commutes with disjunction.

[6] $\boxed{1 : \mathit{Frac}}^{\gamma}$ is timeless, so we can view-shift away the $\triangleright$.

[7] 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+20c].

5.3 Mask-changing view shifts

To explain mask-changing view shifts, let us recall the rule for opening invariants around Hoare triples:

$$\frac{\{\triangleright P * Q_1\}\, e\, \{v.\, \triangleright P * Q_2\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathsf{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E}}{\left\{\boxed{P}^{\mathcal{N}} * Q_1\right\}\, e\, \left\{v.\, \boxed{P}^{\mathcal{N}} * Q_2\right\}_{\mathcal{E}}}$$

The idea behind mask-changing view shifts is to view an application of this rule as consisting of three separate steps:

1. Open the invariant, obtaining $\triangleright P$ in the process.
2. Verify e (with a smaller mask).
3. Close the invariant, consuming $\triangleright P$ in the process.

The second step can already be expressed as an assertion in Iris, which can be found in the premises of the proof rule above:

$$\{\triangleright P * Q_1\}\, e\, \{v.\, \triangleright P * Q_2\}_{\mathcal{E}\setminus\mathcal{N}}$$

The purpose of mask-changing view shifts is to also be able to represent the first and third step as their own assertions, instead of them happening "implicitly" when the rule is used. The core idea is that they can be viewed as a *view-shift that changes the current mask.* We will write $P \;{}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} Q$ for a view shift with precondition P and postcondition Q that requires the current mask to be $\mathcal{E}_1$ and changes it to $\mathcal{E}_2$. We interpret the previously introduced non-mask-changing view shifts $P \Rrightarrow_{\mathcal{E}} Q$ as sugar for $P \;{}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}} Q$.

With that, we can assign to each of the three steps a corresponding Iris assertion:

1. Open the invariant, obtaining $\triangleright P$ in the process: $\mathsf{True} \;{}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \triangleright P$.
2. Verify e (with a smaller mask): $\{\triangleright P * Q_1\}\, e\, \{v.\, \triangleright P * Q_2\}_{\mathcal{E}\setminus\mathcal{N}}$.
3. Close the invariant, consuming $\triangleright P$ in the process: $\triangleright P \;{}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}} \mathsf{True}$.

The core proof rules for mask-changing view shifts and their interaction with Hoare triples and invariants are given in Figure 5.3. The key rule here is HOARE-VS-ATOMIC, which says that *for physically atomic expressions*, we obtain a "rule of consequence" for *mask-changing* view shifts.[8] The rule essentially composes the opening view shift, Hoare triple and closing view shift into a single Hoare triple—notice how at the composition points, the masks are matching (the mask is $\mathcal{E}_2$ in both cases). Somewhat similarly, VS-TRANS lets us compose two mask-changing view shifts if the final mask of the first and the initial mask of the second view shift are the same.

With this, it seems like we can now state the rule for opening and closing invariants *without* being concerned with what the invariant is "opened around":

INV-OPEN-FLAWED

$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \;{}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \triangleright P}$$

INV-CLOSE-FLAWED

$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} * \triangleright P \;{}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}} \mathsf{True}}$$

[8] The general rule of consequence HOARE-VS still works only for non-mask-changing view shifts. In other words, the key difference between mask-changing and non-mask-changing view shifts is that the latter can only be eliminated in pairs around an atomic expression.

$$\textsc{hoare-vs-atomic}\quad \frac{P \;{}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} P' \qquad \{P'\}\, e\, \{v.\, Q'\}_{\mathcal{E}_2} \qquad \forall v.\, Q' \;{}^{\mathcal{E}_2}\!\!\Rrightarrow^{\mathcal{E}_1} Q \qquad \mathsf{atomic}(e)}{\{P\}\, e_1\, \{v.\, Q\}_{\mathcal{E}_1}}$$

$$\textsc{vs-refl}\quad P \;{}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}} P \qquad\qquad \textsc{vs-trans}\quad \frac{P \;{}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} Q \qquad Q \;{}^{\mathcal{E}_2}\!\!\Rrightarrow^{\mathcal{E}_3} R}{P \;{}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_3} R} \qquad\qquad \textsc{vs-frame}\quad \frac{P \;{}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} Q}{P * R \;{}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} Q * R}$$

Figure 5.3: Proof rules for mask-changing view shifts.

However, it turns out that we are not quite there yet. In particular, inv-close-flawed does not actually hold! To see this, remember that inv-alloc lets us allocate an invariant *in any namespace $\mathcal{N}$ of our choice*:

$$P \Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}$$

In particular, we could choose $P := \mathsf{True}$, so in combination with inv-close-flawed, we obtain $\mathsf{True} \;{}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}} \mathsf{True}$, which is quite a disaster: this rule lets us "close" any namespace $\mathcal{N}$ without actually proving anything.

What went wrong here? The problem is that by making inv-open-flawed and inv-close-flawed separate rules, there is no longer anything ensuring that the opening and closing of invariants "matches up", *i.e.*, that we are closing the same invariant which we were opening before. To fix this, we will instead have a *single* rule that combines both opening and closing of an invariant—but still employ mask-changing view shifts so that the same rule is good for opening invariants around Hoare triples and view shifts alike.[9]

We *could* give such a rule using our existing notion of view shifts, but it would be quite convoluted:[10]

$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \;{}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \Big(\triangleright P * \exists R.\, R * \big(R * \triangleright P \;{}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}} \mathsf{True}\big)\Big)}$$

Instead of this approach based on higher-order quantification, we will observe that view shifts as we have seen them so far can be reduced to a *modality*, expressing the "essence" of what a view shift does, in combination with some other (more-or-less) standard Iris connectives. This will let us simplify the above rule, avoiding quantification over R (and indeed the same simplification is also possible for the §3 example).

But first, we perform the same reduction on Hoare triples.

5.4 Weakest preconditions and the persistence modality

Consider the precondition P of a Hoare triple $\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}}$: all P really does is describe the *assumptions* under which the remainder of the statement has to hold. Basically, P has the same role as the context Γ in the turnstile $\Gamma \vdash Q$ of standard propositional or higher-order logic. This makes sense in traditional Hoare logic where specifications (like Hoare triples) are distinct from assertions (which work much like standard

[9] In earlier versions of Iris, we managed to avoid this issue (mostly by accident) by associating invariants not with a namespace but with a single *name* that was existentially quantified, so two different invariants would never have the same name. However, the bookkeeping for all those unpredictable names turned out to be quite tedious, which is why we introduced namespaces to get rid of the existential quantifier in inv-alloc, replacing it by a namespace that we can pick ahead of time.

[10] The R here ensures that the closing part of this rule is used only once, similar to the T in the example spec in §3.

logics), but in Iris, there is no such distinction. Iris already *has* a notion of "logical context", and we do not want to duplicate all that for Hoare triple preconditions.

So instead, we *define* Hoare triples using more primitive Iris connectives: *weakest preconditions* and the *magic wand* (written $-\!\!*$, also sometimes called "separating implication"). Weakest preconditions express that some piece of code is safe to execute and achieves a given postcondition. The magic wand is the natural separation-logic counterpart to implication, it expresses that the weakest precondition is proven under some extra assumptions:[11]

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

To ensure that Hoare triples are persistent, the entire definition is wrapped in the *persistence modality* $\square$; we will discuss this modality shortly.

By defining Hoare triples in terms of weakest preconditions like this, some of the structural rules (such as the disjunction rule, the existential rule, but also HOARE-CTX) can be proven in terms of general properties of the involved logical connectives. They do not need to be primitive rules of the logic any more. We also gain the freedom to easily change this definition; for example, "one-shot" Hoare triples can be obtained simply by omitting the $\square$ modality.

It is crucial to understand that $\mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, Q\}$ is not just a statement about the possible executions of e. Just like all separation logic propositions, this proposition asserts *ownership* of some resources; and in the case of the weakest-precondition proposition, the resources owned must be sufficient to justify that e can be executed safely, after which resources described by Q are owned. Using the weakest precondition to make progress in a proof consumes these resources. For example, proving $\mathsf{wp}_{\mathcal{E}}\ !\ell\, \{\_\}$ will consume ownership of $\ell \mapsto v$, or require knowledge of some invariant that governs ℓ. Weakest preconditions "capture" resources from the environment they are proven in.

The persistence modality. Back in §3.3, we introduced the notion of persistent propositions and explained that some propositions, such as view shifts, are always persistent. To obtain a persistent variant of *any* Iris proposition, we can use the *persistence modality*: we have $\mathsf{persistent}(\square\, P)$. The key rules of the modality are shown in Figure 5.4. Intuitively, $\square\, P$ (pronounced "persistently P") expresses that we have a proof of P that does *not* rely on any exclusive ownership, and thus can be freely duplicated. This is not always possible, so for example $\square\, \ell \mapsto v$ leads to a contradiction.

In fact, the notion of a persistent proposition is defined in terms of this modality:

$$\mathsf{persistent}(P) := P \vdash \square\, P$$

With this, $\square$-IDEMP is exactly $\mathsf{persistent}(\square\, P)$.

This modality is useful because, in general, a proposition like $P -\!\!* Q$ can only be used once because the proof of the magic wand could be relying on exclusive ownership of resources that were present in the context where the wand was proven.[12] By wrapping a magic wand in

[11] The magic wand has an entirely undeserved reputation of being complicated, but it really is no more complicated than implication: both work exactly the same, except where implication interacts with (standard) conjunction, magic wand interacts with *separating conjunction.*

So, where implication is uniquely defined by the following bidirectional rule:

$$\frac{\Gamma \wedge P \vdash Q}{\Gamma \vdash P \Rightarrow Q}$$

Magic wand is uniquely defined by:

$$\frac{\Gamma * P \vdash Q}{\Gamma \vdash P -\!\!* Q}$$

During a separation logic proof, magic wands can be applied to the goal just like implications are applied during a proof in propositional logic.

[12] This kind of "hiding" of resources in a magic wand is exactly analogous to the environment of a closure implicitly—without it being reflected in the *type* of the closure—hiding the data that it captures.

□-ELIM
$$\Box\, P \vdash P$$

□-IDEMP
$$\Box\, P \vdash \Box\,\Box\, P$$

□-MONO
$$\frac{P \vdash Q}{\Box\, P \vdash \Box\, Q}$$

□-AND-SEP
$$\Box\, P \land Q \dashv\vdash \Box\, P * Q$$

□-WAND-KEEP
$$(P \mathrel{-\!\!*} \Box\, Q) \vdash (P \mathrel{-\!\!*} \Box\, Q * P)$$

$\Box$ commutes around $\lor$, $\land$, $*$, $\forall$, and $\exists$

Figure 5.4: Rules for the persistence modality

the persistence modality (like we did for Hoare triples above), we require that only persistent resources be "captured" in the proof of the magic wand. This is best seen by considering the following introduction rule for □, which can be derived from □-MONO and □-IDEMP:

□-INTRO
$$\frac{\Box\, P \vdash Q}{\Box\, P \vdash \Box\, Q}$$

The rule says that to prove proposition $\Box\, Q$, we have to remove everything non-persistent from our context. Only then we can remove the modality blocking our goal. In contrast, removing the modality from an *assumption* is trivial (□-ELIM).

□-AND-SEP states that, since proofs of $\Box\, P$ consume no resources, it does not matter whether one uses conjunction or separating conjunction when combining them with other propositions.[13] We omitted from Figure 5.4 the rules showing that persistence commutes around most base connectives (conjunction, separating conjunction, disjunction, universal and existential quantification, even the later modality).

Last but certainly not least, □-WAND-KEEP is a curious rule: it says that when we have a magic wand whose *conclusion* is persistent, then if we use up some resources P to apply that wand, we can actually keep these resources. This might be surprising, why would we get P back even though it is not persistent? The answer is that $P \mathrel{-\!\!*} \Box\, Q \land P$ can be easily shown from $P \mathrel{-\!\!*} \Box\, Q$ with the normal rules for conjunction, and then □-AND-SEP says this is equivalent to $P \mathrel{-\!\!*} \Box\, Q * P$. Intuitively, a magic wand with a persistent right-hand side is unable to really "use up" any of the exclusive resources of its left-hand side, so we can just keep them.

5.5 View shifts as a modality

After this brief interlude, we come back to the topic of view shifts.

So far, we have treated view shifts $P \;{}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} Q$ as a logical connective stating the persistent fact that, if we own resources described by proposition P and the current mask is $\mathcal{E}_1$, then we can perform an update to the ghost state that changes the mask to $\mathcal{E}_2$, consumes P and provides resources described by Q in return. This is somewhat similar to a Hoare triple with precondition P and postcondition Q, except that there is no code (and thus also no return value and no binder in the postcondition) because only the ghost state is being affected.

[13] Our usual style in Iris is to use separating conjunction when both conjunctions are equivalent. This makes the use of normal conjunction a clear signal that we rely on its overlapping nature. The only example of that in this dissertation is in §12.4, in the interpretation of the continuation context of a λ_{Rust} typing judgment.

Just like Iris Hoare triples are defined in terms of the lower-level notion of a weakest precondition (that does not have a built-in notion of "assumption"), Iris view shifts are defined in terms of the *(fancy) update modality*:[14]

$$P \;{}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\; Q := \Box\big(P \mathrel{-\!\!*} {}^{\mathcal{E}_1}{\mathrel{|\!\!\Rrightarrow}}^{\mathcal{E}_2}\, Q\big)$$

The first thing to notice is the similarity with Hoare triples: the persistence modality and the magic wand are used in exactly the same way, for the same purpose.

The heart of this definition, however, is the modality ${}^{\mathcal{E}_1}{\mathrel{|\!\!\Rrightarrow}}^{\mathcal{E}_2}$: the proposition ${}^{\mathcal{E}_1}{\mathrel{|\!\!\Rrightarrow}}^{\mathcal{E}_2}\, Q$ asserts ownership of resources *that can be updated* to resources satisfying Q, while simultaneously changing the current mask from $\mathcal{E}_1$ to $\mathcal{E}_2$. This process may access invariants whose namespace is contained in $\mathcal{E}_1$. Similar to weakest preconditions, ${}^{\mathcal{E}_1}{\mathrel{|\!\!\Rrightarrow}}^{\mathcal{E}_2}\, Q$ can "capture" arbitrary resources from the environment that it might need to perform this update and eventually produce Q.

The key proof rules for the fancy update modality are given in Figure 5.5 (we use ${\mathrel{|\!\!\Rrightarrow}}_{\mathcal{E}}$ as sugar for ${}^{\mathcal{E}}{\mathrel{|\!\!\Rrightarrow}}^{\mathcal{E}}$). From this, the view shift rules in Figure 5.3 and Figure 3.5 can be derived; in particular, vs-refl can be derived from $\mathrel{|\!\!\Rrightarrow}$-intro-mask and $\mathrel{|\!\!\Rrightarrow}$-trans. We also obtain the "rules of consequence" hoare-vs and hoare-vs-atomic from wp-fup and wp-fup-atomic, respectively. Notice how these rules can now be stated without explicitly naming all the intermediate propositions that "connect" the view shifts and the Hoare triple.

One interesting aspect of the fancy update modality is that it forms a *monad*: $\mathrel{|\!\!\Rrightarrow}$-trans in combination with $\mathrel{|\!\!\Rrightarrow}$-mono is sufficient to derive "bind", and with $\mathrel{|\!\!\Rrightarrow}$-intro-mask we can also derive "return":

$\mathrel{|\!\!\Rrightarrow}$-bind

$$\frac{P \vdash {}^{\mathcal{E}_2}{\mathrel{|\!\!\Rrightarrow}}^{\mathcal{E}_3}\, Q}{{}^{\mathcal{E}_1}{\mathrel{|\!\!\Rrightarrow}}^{\mathcal{E}_2}\, P \vdash {}^{\mathcal{E}_1}{\mathrel{|\!\!\Rrightarrow}}^{\mathcal{E}_3}\, Q}$$

$\mathrel{|\!\!\Rrightarrow}$-return

$$P \vdash {\mathrel{|\!\!\Rrightarrow}}_{\mathcal{E}}\, P$$

Correspondingly, a helpful intuition for fancy updates is that ${}^{\mathcal{E}_1}{\mathrel{|\!\!\Rrightarrow}}^{\mathcal{E}_2}\, P$ is something like a "suspended ghost action" that, when executed, consumes its implicitly "captured" resources and produces resources described by P. The rule $\mathrel{|\!\!\Rrightarrow}$-frame says that, additionally, fancy updates are a *strong monad* with respect to separating conjunction.

The bind rule also demonstrates how we usually *eliminate* the fancy update modality: if the goal is itself below an update modality, we can just remove it from the context[15] if the masks match up (and doing so can change our mask as well). The bind rule is basically dual in shape to $\Box$-intro. Thus, another good name for $\mathrel{|\!\!\Rrightarrow}$-bind would have been $\mathrel{|\!\!\Rrightarrow}$-elim.

5.6 Accessors

With the fancy update modality at our hands, we are *finally* ready to state the rule for accessing invariants:

inv-acc

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

[14] One may wonder what is so "fancy" about this modality. Well, this fancy update modality is defined in terms of some even lower-level modality, and we had to distinguish the two *somehow*—and we ended up calling them the "basic" and "fancy" update modalities. This is not the most aptly named Iris connective, but it is also not the worst.

[15] Using $\mathrel{|\!\!\Rrightarrow}$-frame, we can even remove it from any of a bunch of separately-conjuncted propositions in the context.

$\vert\!\!\Rrightarrow$-MONO

$$\frac{P \vdash Q}{{}^{\mathcal{E}_1}{\vert\!\!\Rrightarrow}^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}{\vert\!\!\Rrightarrow}^{\mathcal{E}_2} Q}$$

$\vert\!\!\Rrightarrow$-INTRO-MASK

$$\frac{\mathcal{E}_2 \subseteq \mathcal{E}_1}{\mathsf{True} \vdash {}^{\mathcal{E}_1}{\vert\!\!\Rrightarrow}^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}{\vert\!\!\Rrightarrow}^{\mathcal{E}_1} \mathsf{True}}$$

$\vert\!\!\Rrightarrow$-TRANS

$${}^{\mathcal{E}_1}{\vert\!\!\Rrightarrow}^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}{\vert\!\!\Rrightarrow}^{\mathcal{E}_3} P \vdash {}^{\mathcal{E}_1}{\vert\!\!\Rrightarrow}^{\mathcal{E}_3} P$$

$\vert\!\!\Rrightarrow$-FRAME

$$Q * {}^{\mathcal{E}_1}{\vert\!\!\Rrightarrow}^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1 \uplus \mathcal{E}_\mathrm{f}}{\vert\!\!\Rrightarrow}^{\mathcal{E}_2 \uplus \mathcal{E}_\mathrm{f}} (Q * P)$$

WP-FUP

$${\vert\!\!\Rrightarrow}_{\mathcal{E}}\, \mathsf{wp}_{\mathcal{E}}\; e \,\{v.\, {\vert\!\!\Rrightarrow}_{\mathcal{E}} Q\} \vdash \mathsf{wp}_{\mathcal{E}}\; e \,\{v.\, Q\}$$

WP-FUP-ATOMIC

$$\frac{\mathsf{atomic}(e)}{{}^{\mathcal{E}_1}{\vert\!\!\Rrightarrow}^{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\; e \left\{v.\, {}^{\mathcal{E}_2}{\vert\!\!\Rrightarrow}^{\mathcal{E}_1} Q\right\} \vdash \mathsf{wp}_{\mathcal{E}_1}\; e \,\{v.\, Q\}}$$

Figure 5.5: Rules for the fancy update modality.

To understand this rule, we break it into its constituent parts: assuming that we know of some invariant in namespace $\mathcal{N}$ that maintains P, we can obtain a mask-changing fancy update that removes $\mathcal{N}$ from the mask and gives us two resources in return. The first is $\triangleright P$, the content of the invariant. The second is a magic wand that, *consuming* $\triangleright P$, provides a fancy update to change the mask back to what it was. We call this second part the "closing" part, because "executing" that fancy update is when the invariant gets closed again. This rule is, as it turns out, equivalent to the one at the end of §5.3. The details of that argument are left as an exercise to the reader.[16]

Together with WP-FUP-ATOMIC and $\vert\!\!\Rrightarrow$-TRANS, we can derive both HOARE-INV and VS-INV from INV-ACC. We now have a single proof rule that says how invariants can be accessed, without regard for whether we are accessing them from a Hoare triple or a view shift.

Following the same pattern, we can specify accessing *cancellable* invariants as follows:

CINV-ACC

$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\mathsf{CInv}^{\gamma,\mathcal{N}}(P) \vdash [\mathsf{CInv} : \gamma]_q \mathrel{-\!\!*} {}^{\mathcal{E}}{\vert\!\!\Rrightarrow}^{\mathcal{E}\setminus\mathcal{N}} \big(\triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}{\vert\!\!\Rrightarrow}^{\mathcal{E}} [\mathsf{CInv} : \gamma]_q)\big)}$$

Again this lets us open cancellable invariants both around Hoare triples and around view shifts.

We can, in fact, also prove a slightly stronger rule that gives back the $[\mathsf{CInv} : \gamma]_q$ token immediately instead of waiting until the invariant has been closed again:[17]

CINV-ACC-STRONG

$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\begin{array}{l}\mathsf{CInv}^{\gamma,\mathcal{N}}(P) \vdash [\mathsf{CInv} : \gamma]_q \mathrel{-\!\!*} {}^{\mathcal{E}}{\vert\!\!\Rrightarrow}^{\mathcal{E}\setminus\mathcal{N}} \\ \qquad \big(\triangleright P * [\mathsf{CInv} : \gamma]_q * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}{\vert\!\!\Rrightarrow}^{\mathcal{E}} \mathsf{True})\big)\end{array}}$$

Accessors. The three proof rules we have just described all follow the same pattern:

$$R \vdash Q_1 \mathrel{-\!\!*} {}^{\mathcal{E}_1}{\vert\!\!\Rrightarrow}^{\mathcal{E}_2} \big(P_1 * (P_2 \mathrel{-\!\!*} {}^{\mathcal{E}_2}{\vert\!\!\Rrightarrow}^{\mathcal{E}_1} Q_2)\big)$$

[16] Hint: think of closure conversion; R is the "captured environment" of the magic wand.

[17] To be clear, this stronger rule can also be expressed in the style of VS-CINV and HOARE-CINV. But we felt the weaker rule is more intuitive for the initial explanation.

We call this an *accessor*, as its purpose is to encode some way to access some resources. Here, R is some persistent proposition that is needed to enable the desired rule (but, being persistent, it does not actually get consumed by the rule); typically, R is knowledge of some kind of invariant. Q_1 gets consumed when the rule is applied. It is like a "precondition" to the accessor. This is used by cancellable invariants to demand ownership of (some fraction of) the invariant token. Next, the rule says that we can execute a mask-changing fancy update from $\mathcal{E}_1$ to $\mathcal{E}_2$ in order to obtain ownership of P_1—in INV-ACC, this is the content of the invariant; in CINV-ACC-STRONG the rule also gives back ownership of the invariant token. On top of these resources, the rule also provides ownership of a magic wand with precondition P_2, forcing the client to reestablish the invariant. After satisfying those preconditions, the client may execute another mask-changing fancy update to switch the mask back to $\mathcal{E}_1$. In so doing it obtains ownership of Q_2, the "postcondition" of the accessor.

It helps to think of the library providing the accessor, and the client using it, as being two parties that both follow a common protocol. So after R establishes that the two parties are indeed bound by the rules of the protocol, their interaction proceeds as follows:

1. The client has to give Q_1 to the library.
2. The library changes the mask, and gives P_1 to the client.
3. The client has to give P_2 to the library.
4. The library changes the mask back, and gives Q_2 to the client.

Accessors arise frequently when defining higher-order abstractions in Iris that involve managing resources for the client. We have not proven any proof rules about accessors in general; it does not seem like that would actually be any simpler than working directly with the rules for magic wand and the update modality. Instead, we treat accessors more like a *design pattern*: a common and well-tried solution for a reoccurring class of problems.

Accessors are quite syntactically heavy, involving two update modalities, magic wands, and no less than five Iris propositions. Nevertheless, we hope this section helps to demonstrate that there is a fairly simple underlying intuition.

Accessor for STSs. Another example of an accessor arises from HOARE-STS-TIMELESS (page 52). The following rule expresses that an STS can be accessed around either a view shift or a Hoare triple (and it also lifts the "timeless" restriction by adding $\triangleright$ instead):

STS-ACC

$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\begin{array}{c}\boxed{\mathsf{StsInv}_\gamma}^{\mathcal{N}} \vdash \mathsf{StsSt}_\gamma(s_1, T_1) \mathrel{-\!\!*} {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}\setminus\mathcal{N}} \exists s.\, s_1 \xrightarrow{T} s * \triangleright\varphi(s) * \\ \big(\forall s_2, T_2.\, (s, T_1) \rightarrow (s_2, T_2) * \triangleright\varphi(s_2) \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow}^{\mathcal{E}} \mathsf{StsSt}_\gamma(s_2, T_2)\big)\end{array}}$$

This is quite a mouthful, but it still pattern-matches with the general shape of an accessor, albeit now with some extra quantifiers:

$$R \vdash Q_1 \mathrel{-\!\!*} {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} \exists x.\, P_1 * (\forall y.\, P_2 \mathrel{-\!\!*} {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_1} Q_2)$$

Here, x is a binder for information that the client of the accessor *learns* from the library when using the accessor (like the current state s of the STS), and y is a binder for information that the client *chooses* when using the accessor (like the new state s_1 of the STS).

It remains an open problem to find a suitable syntax for accessors that makes them less daring to look at in general, but there is a special case that has a useful short-hand syntax.

Symmetric accessors. With inv-acc and cinv-acc, we have seen two accessors that are *symmetric* in the sense that the resources which are taken back in the "closing" phase of the accessor are the same as the ones we started with, *i.e.*, $P_2 = P_1$ and $Q_2 = Q_1$. This pattern turns out to be common enough that it is worth introducing some syntactic sugar to avoid having to repeat P and Q:

$$Q \;\; {}^{\mathcal{E}_1}{\propto}^{\mathcal{E}_2} \;\; P := Q \mathrel{-\!\!*} {}^{\mathcal{E}_1}{\models\!\!\!\Rrightarrow}^{\mathcal{E}_2} \big(P * (P \mathrel{-\!\!*} {}^{\mathcal{E}_2}{\models\!\!\!\Rrightarrow}^{\mathcal{E}_1} Q)\big)$$

With this, we can state the two aforementioned rules concisely as:

$$\textsc{inv-acc}\quad \frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \vdash \mathsf{True} \;\; {}^{\mathcal{E}}{\propto}^{\mathcal{E}\setminus\mathcal{N}} \;\; \triangleright P} \qquad\qquad \textsc{cinv-acc}\quad \frac{\mathcal{N} \subseteq \mathcal{E}}{\mathsf{CInv}^{\gamma,\mathcal{N}}(P) \vdash [\mathsf{CInv} : \gamma]_q \;\; {}^{\mathcal{E}}{\propto}^{\mathcal{E}\setminus\mathcal{N}} \;\; \triangleright P}$$

Prior work. The specification pattern we dubbed "accessors" is closely related to some existing ideas described in the literature.

The most directly related prior work is the idea of a "ramification".[18] Ramifications have the shape $R \vdash P * (Q \mathrel{-\!\!*} R')$; the similarity to accessors should be quite obvious. Ramifications were proposed as a way to specify operations that let a program access some part of a large data structure, mutate that part, and then see those changes reflected in the data structure as a whole. For example, ramifications might be used to specify a function that returns a pointer to one particular node in a tree or graph. The accessors we saw above do basically the same, except that there is no physical operation being performed; the "data structure" is entirely ghost state (and there are update modalities inserted in strategic places to reflect those ghost state changes).

A slight variant of this pattern also arises in the work on Mezzo[19] when specifying a `find` operation for lists (again, a way to access one mutable element of a container): the postcondition of `find` says it returns a "focused" element, which in particular involves a "wand" (encoded as a function) to convert ownership of the element back to ownership of the entire list. This exactly corresponds to the wand in a ramification or the "closing" part of an accessor. In fact, even the original work on "Adoption and Focus"[20] already contains some indication of this pattern.

Incidentally, the authors of Mezzo also call this pattern "borrowing", which immediately raises the question of there being any relation to the kind of borrowing that Rust offers (see §8.2). And indeed, the Rust verification tool Prusti[21] models some forms of borrowing with exactly the same pattern. Namely, this applies to functions with a type like

[18] Hobor and Villard, "The ramifications of sharing in data structures", 2013 [HV13].

[19] Balabonski, Pottier, and Protzenko, "The design and formalization of Mezzo, a permission-based programming language", 2016 [BPP16], §2.4.

[20] Fähndrich and DeLine, "Adoption and focus: Practical linear types for imperative programming", 2002 [FD02].

[21] Astrauskas *et al.*, "Leveraging Rust types for modular specification and verification", 2019 [Ast+19].

`fn(&mut T) -> &mut U`, which basically means that the function takes a reference to some large data structure and returns a reference to a part of it—so, this is again the same kind of situation as ramifications and Mezzo's "focused" elements.

5.7 Summary: Iris proof rules

To summarize the Iris features that we have introduced so far, we collect all the basic language-independent proof rules for weakest preconditions, fancy updates, invariants, ghost state, later, and the persistence modality in Figure 5.6. Some rules look slightly different than before: wp-frame, wp-value and wp-bind are the weakest precondition versions of hoare-frame, hoare-value and hoare-bind, respectively. Similarly, $|\!\!\Rrightarrow$-timeless is the modality variant of vs-timeless, and ghost-alloc as well as ghost-update have been adapted as well. Finally, wp-step is a version of hoare-step that also supports eliminating mask-changing fancy updates together with the $\triangleright$ modality when taking a step. (Note how similar it is to wp-frame, except for the extra modalities.)

These are *not* the most primitive proof rules of Iris; most of them are in fact derived from the lower-level *Iris base logic.*[22] Instead, these rules form the foundation of "high-level" Iris. Understanding their encoding in terms of lower-level primitives (of which we have seen a glimpse in §4.3) is not usually necessary when working with Iris, and will not be necessary for the purpose of this dissertation.

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

Weakest preconditions.

wp-frame

$$Q * \mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, P\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, Q * P\}$$

wp-value

$$P[v/w] \dashv\vdash \mathsf{wp}_{\mathcal{E}}\, v\, \{w.\, P\}$$

wp-bind

$$\mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, \mathsf{wp}_{\mathcal{E}}\, K[v]\, \{w.\, P\}\} \dashv\vdash \mathsf{wp}_{\mathcal{E}}\, K[e]\, \{w.\, P\}$$

Fancy updates.

$\mathrel{|\!\!\Rrightarrow}$-mono

$$\frac{P \vdash Q}{{}^{\mathcal{E}_1}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_2} Q}$$

$\mathrel{|\!\!\Rrightarrow}$-intro-mask

$$\frac{\mathcal{E}_2 \subseteq \mathcal{E}_1}{\mathsf{True} \vdash {}^{\mathcal{E}_1}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_1} \mathsf{True}}$$

$\mathrel{|\!\!\Rrightarrow}$-elim

$$\frac{P \vdash {}^{\mathcal{E}_2}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_3} Q}{{}^{\mathcal{E}_1}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_3} Q}$$

$\mathrel{|\!\!\Rrightarrow}$-trans

$${}^{\mathcal{E}_1}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_3} P \vdash {}^{\mathcal{E}_1}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_3} P$$

$\mathrel{|\!\!\Rrightarrow}$-frame

$$Q * {}^{\mathcal{E}_1}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1 \uplus \mathcal{E}_\mathsf{f}}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_2 \uplus \mathcal{E}_\mathsf{f}} (Q * P)$$

wp-$\mathrel{|\!\!\Rrightarrow}$

$$\mathrel{|\!\!\Rrightarrow}_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, \mathrel{|\!\!\Rrightarrow}_{\mathcal{E}} Q\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, Q\}$$

wp-$\mathrel{|\!\!\Rrightarrow}$-atomic

$$\frac{\mathsf{atomic}(e)}{{}^{\mathcal{E}_1}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\, e\, \left\{v.\, {}^{\mathcal{E}_2}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_1} Q\right\} \vdash \mathsf{wp}_{\mathcal{E}_1}\, e\, \{v.\, Q\}}$$

Ghost state.

ghost-alloc

$$\frac{\mathcal{V}(a)}{\mathrel{|\!\!\Rrightarrow}_{\mathcal{E}} \exists \gamma.\, \boxed{a}^{\gamma}}$$

ghost-op

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

ghost-valid

$$\boxed{a}^{\gamma} \Rightarrow \mathcal{V}(a)$$

ghost-update

$$\frac{a \rightsquigarrow B}{\boxed{a}^{\gamma} \mathrel{-\!\!*} \mathrel{|\!\!\Rrightarrow}_{\mathcal{E}} \exists b \in B.\, \boxed{b}^{\gamma}}$$

Persistence modality.

$\square$-elim

$$\square P \vdash P$$

$\square$-intro

$$\frac{\square P \vdash Q}{\square P \vdash \square Q}$$

$\square$-mono

$$\frac{P \vdash Q}{\square P \vdash \square Q}$$

$\square$-and-sep

$$\square P \wedge Q \dashv\vdash \square P * Q$$

$\square$-wand-keep

$$(P \mathrel{-\!\!*} \square Q) \vdash (P \mathrel{-\!\!*} \square Q * P)$$

$\square$ commutes around $\vee$, $\wedge$, $*$, $\forall$, and $\exists$.

Timeless propositions and the later modality.

$\triangleright$-intro

$$P \vdash \triangleright P$$

$\triangleright$-mono

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

$\triangleright$ commutes around $\square$, $\vee$, $\wedge$, $*$, $\forall$, and $\exists$ with non-empty domain

wp-step

$$\frac{e \text{ is not a value} \qquad \mathcal{E}_2 \subseteq \mathcal{E}_1}{{}^{\mathcal{E}_1}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_2} \triangleright\, {}^{\mathcal{E}_2}\mathrel{|\!\!\Rrightarrow}^{\mathcal{E}_1} Q * \mathsf{wp}_{\mathcal{E}_2}\, e\, \{x.\, P\} \vdash \mathsf{wp}_{\mathcal{E}_1}\, e\, \{x.\, Q * P\}}$$

$\mathrel{|\!\!\Rrightarrow}$-timeless

$$\frac{\mathsf{timeless}(P)}{\triangleright P \mathrel{-\!\!*} \mathrel{|\!\!\Rrightarrow}_{\mathcal{E}} P}$$

timeless-ghost

$$\mathsf{timeless}(\boxed{a}^{\gamma})$$

$\mathsf{timeless}$ propagates structurally through $\vee$, $\wedge$, $*$, $\forall$, $\exists$.

Invariants.

inv-alloc

$$\triangleright P \Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}$$

inv-acc

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

Figure 5.6: Language-independent Iris proof rules.

Chapter 6

Paradoxes

In this chapter, we describe two logical paradoxes that help motivate two Iris design choices. The first paradox demonstrates why we have a $\triangleright$ modality, by showing that outright removing it from some key proof rules leads to a logical contradiction. The second paradox explains why Iris is an *affine* logic, *i.e.*, why Iris comes with the weakening rule ($P * Q \Rightarrow P$). Usually, the absence of this rule enables a separation logic to ensure that a program has no memory leaks. We show that in a logic with impredicative invariants such as Iris, this is not the case—even without weakening, memory leaks are still possible.

6.1 Naive higher-order ghost state paradox

As we have seen, the setup of Iris is such that the user provides a resource algebra M by which the logic is parameterized. This gives a lot of flexibility, as the user is free to decide what kind of ghost state to use. We now discuss whether we could stretch this flexibility even further, by allowing the construction of the resource algebra M to depend on the type of Iris propositions *iProp*. We dubbed this phenomenon *higher-order ghost state*[1] and showed that it has useful applications in program verification. In follow-on work,[2] we then showed that higher-order ghost state has applications beyond program verification: it can be used to encode invariants in terms of plain ghost state, removing them from the core logic.

The way we model higher-order ghost state puts certain restrictions on the way *iProp* can be used in the construction of the user-supplied resource algebra. Originally,[3] these restrictions appeared as a semantic artifact of modeling the logic. However, in this section we show that restrictions of *some* kind are in fact necessary: allowing higher-order ghost state in a naive or unrestricted manner leads to a paradox (*i.e.*, an unsound logic). In order to demonstrate this paradox, we use the simplest instance of higher-order ghost state, *saved propositions* (also called “stored” or “named” propositions).[4]

In order to explain saved propositions, let us have another look at the agreement resource algebra $Ag_0(X)$. As part of the decomposition of the oneshot RA into small, reusable pieces, we also generalized the set of values that the proof can pick from (*i.e.*, the parameter of Ag_0) from $\mathbb{Z}$ to any given set X. Instead of choosing X to be a “simple” set like $\mathbb{Z}$ or $\wp(\mathbb{N})$, we could try choosing it to be *iProp*, the type of Iris propositions

[1] Jung *et al.*, “Higher-order ghost state”, 2016 [Jun+16].

[2] Krebbers *et al.*, “The essence of higher-order concurrent separation logic”, 2017 [Kre+17].

[3] Jung *et al.*, “Higher-order ghost state”, 2016 [Jun+16].

[4] Dodds *et al.*, “Verifying custom synchronization constructs using higher-order separation logic”, 2016 [Dod+16].

itself. This is an example of higher-order ghost state.[5] The RA $Ag_0(\mathit{iProp})$ would then allow us to associate a ghost location (or "name") γ with a proposition P. However, this ability to "name" propositions leads to a paradox.

Theorem 1 (Higher-order ghost state paradox)**.** *Assume we can instantiate the construction from §4.3 with $Ag_0(\mathit{iProp})$ being in the family of RAs. Then we have:*[6]

$$\mathsf{True} \Rrightarrow \mathsf{False}$$

Before proving this paradox, let us take a look at the proof rules of Iris that are crucial for the proof:

$$\mathsf{True} \Rrightarrow \exists \gamma.\, \boxed{\mathsf{ag}_0(A(\gamma))}^{\gamma} \qquad \text{(sprop0-alloc)}$$

$$\boxed{\mathsf{ag}_0(P_1)}^{\gamma} * \boxed{\mathsf{ag}_0(P_2)}^{\gamma} \Rightarrow P_1 = P_2 \qquad \text{(sprop0-agree)}$$

The rule sprop0-agree, which states that saved propositions with the same name are the same, follows easily from ag0-agree. The rule sprop0-alloc, which allows one to allocate a name for a proposition, looks a lot like an instance of ghost-alloc, except that the initial state of the new ghost location may depend on the fresh γ. Said dependency is expressed through A, a function mapping ghost names to *iProp*. In fact, the rule sprop0-alloc follows from the following generalized rule for ghost state allocation:

$$\frac{\forall \gamma.\, \mathcal{V}(g(\gamma))}{\mathsf{True} \Rrightarrow \exists \gamma.\, \boxed{g(\gamma)}^{\gamma}} \qquad \text{(ghost-alloc-dep)}$$

This rule allows the initial state of the freshly allocated ghost location (described by the function g) to depend on the location γ that has been picked. The rule ghost-alloc-dep can be proven from the basic rules for the $\mathsf{Own}\,(-)$ connective as shown in Figure 4.2 and fmap-alloc-dep (page 47). This completes the proof of sprop0-alloc.

We now turn to the proof of Theorem 1. First, we define

$$\begin{aligned} A(\gamma) &:= \exists P.\, \Box(P \Rightarrow \mathsf{False}) * \boxed{\mathsf{ag}_0(P)}^{\gamma} \\ Q(\gamma) &:= \boxed{\mathsf{ag}_0(A(\gamma))}^{\gamma} \end{aligned}$$

Intuitively, $A(\gamma)$ asserts that ghost location γ names some proposition P that does not hold. (Since our ghost locations are described by the RA $Ag_0(\mathit{iProp})$, we know that everybody agrees on which proposition is named by a ghost location γ, so it makes sense to talk about "the proposition with name γ".)

The proposition $Q(\gamma)$ says that the proposition with name γ is $A(\gamma)$. After unfolding the definition of A, we can see that $Q(\gamma)$ means the following: "the proposition with name γ says that the proposition with name γ does not hold". In other words, $Q(\gamma)$ means that the proposition with name γ states its own opposite ($P \Leftrightarrow \neg P$). This is an instance of the classic "liar's paradox", so it should not be surprising that it leads to a contradiction. Concretely, we show the following lemma, from which Theorem 1 is a trivial consequence.

[5] The set *iProp* of Iris propositions actually depends on which RA the user picks, so we cannot refer to *iProp* when picking the RA. The point of this paradox is to show what would go wrong if we could.

[6] Masks do not matter in this section, hence we will omit them.

Lemma 1. *We have the following properties:*

1. $Q(\gamma) \Rightarrow \Box(A(\gamma) \Rightarrow \mathsf{False})$,
2. $Q(\gamma) \Rightarrow A(\gamma)$, *and,*
3. $\mathsf{True} \Rrightarrow \exists \gamma.\, Q(\gamma)$.

Proof. Notice that all propositions we are using for this paradox, except the existentially quantified P in the definition of $A(\gamma)$, are persistent, so we can mostly ignore the substructural aspects of Iris in this proof.

For 1, we can assume $Q(\gamma)$ and $A(\gamma)$ and we have to show False. After unfolding both of them, we obtain some P such that:

$$\boxed{\mathsf{ag}_0(A(\gamma))}^{\gamma} * \Box(P \Rightarrow \mathsf{False}) * \boxed{\mathsf{ag}_0(P)}^{\gamma}$$

From sprop0-agree, we can now obtain that $A(\gamma) = P$. Since we have a proof of $A(\gamma)$, and we have $\Box(P \Rightarrow \mathsf{False})$, this yields the desired contradiction.

For 2, we can assume $Q(\gamma)$. Our goal is $A(\gamma)$, so we start by picking $P := A(\gamma)$. We now have to show $\Box(A(\gamma) \Rightarrow \mathsf{False})$, which easily follows from (1) and our $Q(\gamma)$. We further have to show $\boxed{\mathsf{ag}_0(A(\gamma))}^{\gamma}$, which is exactly $Q(\gamma)$. Hence we are done.

Finally, for 3, we use sprop0-alloc. □

So, what went wrong here? Where does this contradiction originate from? The problem is that higher-order ghost state, and saved propositions in particular, allow us to express concepts like "the proposition with name γ". This allows us to define a proposition as being its own negation. Usually, such nonsense is prevented because the definition of a proposition cannot refer back to this proposition in a cyclic way. The fact that higher-order ghost state lets us give *names* to propositions lets us circumvent this restriction and close the cycle. In some sense, this is very similar to Landin's knot, where a recursive function is defined by indirectly setting up recursive references of the function to itself through a location ("name") on the (higher-order) heap.

This is the simplest later-related paradox in Iris, but not the only one: removing the $\triangleright$ from the rules for invariants (inv-alloc, vs-inv, hoare-inv) also leads to a contradiction.[7] This shows that higher-order ghost state is not the only feature that needs *some* mechanism to avoid these paradoxes: even "just" having impredicative invariants (which can be modeled without general higher-order ghost state[8]) is unsound when the $\triangleright$ is removed without replacement.

Saved propositions, soundly. The paradox shows that one has to be very careful if one wishes to support higher-order ghost state. In particular, one cannot allow resource algebras like $Ag_0(iProp)$ to be used to instantiate the Iris logic. In Jung *et al.*,[9] we show that higher-order ghost state is sound if the recursive occurrences of *iProp* are suitably *guarded*.

To model saved propositions, we then use $Ag(\blacktriangleright iProp)$. Here, Ag is a refined definition of the agreement RA Ag_0 that supports recursive definitions,[10] and the "later" type constructor $\blacktriangleright$ (whose value constructor

[7] Jung *et al.*, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b], §8.2.

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

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

[10] The uses of Ag_0 and ag_0 in §4 can all be replaced by Ag and ag since we maintain the key properties ag0-agree, ag0-inj, and ag0-uninj, as well as $a \cdot a = a$.

is next) plays the role of a guard. Based on that, we can define saved propositions as follows:

$$\gamma \Mapsto P := \boxed{\mathsf{ag}(\mathsf{next}(P))}^{\gamma}$$

The following proof rules hold:[11]

$$\mathsf{True} \Rrightarrow \exists \gamma.\, \gamma \Mapsto P \qquad (\textsc{sprop-alloc})$$

$$\gamma \Mapsto P_1 * \gamma \Mapsto P_2 \Rightarrow \triangleright(P_1 = P_2) \qquad (\textsc{sprop-agree})$$

In particular, note how sprop-agree is weaker than sprop0-agree: this rule gives us the same equality, but only one step of execution "later", which prevents the paradox. Moreover, neither $\gamma \Mapsto P$ nor equality of propositions are timeless, so we cannot use vs-timeless to work around this extra $\triangleright$. If we tried to re-prove Theorem 1, we would end up defining an assertion that is equivalent to its *later* negation (*i.e.*, $P \Leftrightarrow \neg \triangleright P$), and there is nothing contradictory about that.

This sound form of higher-order ghost state is also used to define Iris invariants in terms of ghost state, but the details of that encoding are beyond the scope of this dissertation.[12]

6.2 Linear impredicative invariants paradox

One common question about Iris is why Iris is an *affine* (or *intuitionistic*) separation logic. Why does $P*Q \Rightarrow P$ (the *weakening* rule) hold? Initially, this was mostly a choice of convenience: having weakening available meant we did not have to worry about "cleaning up" (we could just throw away any leftover resources). The typical motivation for using a separation logic without weakening (a *linear* or *classical* separation logic) is to verify the absence of memory leaks, but the tricky concurrent algorithms we were verifying anyway assumed a garbage collector—so memory leaks were not a concern. As Iris got applied to a broader set of problems, the first linear variants of Iris were proposed.[13] However, it turned out that suitably generalizing the axioms of Iris to fit the linear setting is far from trivial.[14]

But this is not the only reason that Iris for now remains an affine logic: it also turns out that in a program logic with impredicative invariants like Iris, linearity is basically useless! Even if Iris was linear, it could *not* be used to verify the absence of memory leaks. That is demonstrated by the paradox we describe in this section.

In a linear separation logic, the proposition Emp expresses that we do not own *any* resources. This is useful because it means that when a Hoare triple has post-condition Emp, there cannot be any memory leaks: if any memory was still allocated, ownership of that memory would have to be accounted for in the post-condition. The problem is that when mixing linearity and impredicative invariants, this is not true any more. We will demonstrate this by deriving $P \Rrightarrow_{\top} \mathsf{Emp}$, letting us "throw away" *any* resource, which should never be possible in a linear separation logic. In particular, a trivial consequence is that we obtain $\{\mathsf{Emp}\}\, \mathtt{\textbf{ref}}(0)\, \{\mathsf{Emp}\}$, "proving" that this clearly leaky program has no memory leak.

[11] A dependent version of sprop-alloc is possible, but will not be necessary for the purpose of this dissertation.

[12] Krebbers *et al.*, "The essence of higher-order concurrent separation logic", 2017 [Kre+17].

[13] Tassarotti, Jung, and Harper, "A higher-order logic for concurrent termination-preserving refinement", 2017 [TJH17].

[14] Krebbers *et al.*, "MoSeL: A general, extensible modal framework for interactive proofs in separation logic", 2018 [Kre+18].

Impredicative invariants break linearity. Before we can get into the details, we have to determine a suitable notion of invariants for a linear logic. The invariants of Iris as presented in §3 are clearly not a good fit, as they let us put any proposition into an invariant and then entirely lose track of the invariant's existence: $\boxed{P}^{\mathcal{N}}$ can be arbitrarily duplicated and discarded. Instead, we will use the *cancellable invariants* introduced in §5.2. In particular, we will treat the invariant token $[\mathsf{CInv} : \gamma]_q$ as *linear*, meaning $[\mathsf{CInv} : \gamma]_q \nvdash \mathsf{Emp}$. This means that invariant tokens cannot be "thrown away". Intuitively, to prove a Hoare triple with post-condition Emp, we *have to* cancel all invariants and then appropriately dispose of all the resources stored in these invariants, as that is the only way to get rid of the token—except that this intuition does not actually work out, as we will see.

Theorem 2 (Linear impredicative invariants paradox)**.** *Assume a linear separation logic (*i.e.*, a separation logic without the weakening rule $P * Q \Rightarrow P$) with impredicative cancellable invariants (specifically,* cinv-alloc *and* cinv-acc-strong *with* Emp *replacing* True*).*

Then we can prove:

$$P \Rrightarrow_{\top} \mathsf{Emp}$$

This paradox is not a logical contradiction in the sense of deriving a proof of False, but it *does* break the important meta-theoretic property of the logic that is used to verify the absence of memory leaks.

The proof of this paradox is not very complicated. Essentially, we can put the token for the invariant into the invariant itself. Thus we can have a postcondition of Emp, but still have an invariant that owns and hides arbitrary resources. The key result is the following lemma, from which Theorem 2 is a trivial consequence via cinv-alloc:

Lemma 2. *We can show the following:*

$$\mathsf{CInv}^{\gamma,\mathcal{N}}(\mathsf{True}) * [\mathsf{CInv} : \gamma]_1 \Rrightarrow_{\top} \mathsf{Emp}$$

Proof. We use cinv-acc-strong, which we can do because we own $[\mathsf{CInv} : \gamma]_1$. That means the invariant gets opened, and we have to prove:

$$[\mathsf{CInv} : \gamma]_1 * \triangleright \mathsf{True} * \Big((\triangleright \mathsf{True}) \mathrel{-\!\!*} {}^{\top\setminus\mathcal{N}}{\mid\!\!\Rrightarrow}^{\top} \mathsf{Emp}\Big) \mathrel{-\!\!*} \\ {}^{\top\setminus\mathcal{N}}{\mid\!\!\Rrightarrow}^{\top} \mathsf{Emp}$$

This means our current resources are the token $[\mathsf{CInv} : \gamma]_1$ and the content of the invariant $\triangleright \mathsf{True}$, together with the closing part of the accessor. We could use that closing view shift to close the invariant again using our $\triangleright \mathsf{True}$ (putting the same resources back into the invariant that we just got out), but then we would be left with $[\mathsf{CInv} : \gamma]_1$. So instead, we will use *both* $\triangleright \mathsf{True}$ and $[\mathsf{CInv} : \gamma]_1$ to satisfy the premise of the closing view shift, effectively adding $[\mathsf{CInv} : \gamma]_1$ to the resources held by the invariant.

To this end, we simply apply the closing view shift (its conclusion matches our goal), which changes the goal to:

$$[\mathsf{CInv} : \gamma]_1 * \triangleright \mathsf{True} \mathrel{-\!\!*} \triangleright \mathsf{True}$$

By ▷-intro, this holds trivially (because $Q \mathrel{-\!\!*} \mathsf{True}$ for any Q). □

Iris itself, being an affine logic, is not directly affected by this theorem. But other logics are: one example of a linear separation logic with impredicative invariants is the logic of the VST framework.[15] In VST, invariants are available in the form of "resource invariants" that are associated with locks. Opening and closing the invariant in Iris corresponds to acquiring and releasing the lock. So, unlike in Iris, invariants are directly tied to some part of the physical state. Still, Theorem 2 can probably be translated to VST. There is no rule directly corresponding to cinv-acc-strong, but Hobor's PhD thesis[16] describes a strong variant of the proof rule for releasing a lock that can be used to let one thread send *all* its resources to another thread, including the resources needed to access the lock itself. In personal conversation,[17] we got confirmation that this is likely sufficient to "leak" resources in VST, in the same vein as Theorem 2. And indeed, VST does not have a theorem stating that memory-leak freedom can be established by showing a Hoare triple with postcondition Emp.

One natural question in this context is whether there is some way to weaken impredicative invariants such that they do not subvert linearity. This question is answered positively by Iron,[18] a linear logic defined on top of Iris. Iron has impredicative invariants, but they come with an extra side-condition: the invariant needs to be "uniform", *i.e.*, not every possible assertion may be used as an invariant. And, in particular, the token required to deallocate an invariant is *not* uniform and as such may not be put into an invariant. Indeed, if the uniformity condition were to be removed from Iron, the logic would fully satisfy the assumptions of Theorem 2. The theorem thus demonstrates that the side-condition is, in some sense, necessary.

A note on terminology. Above, we use the terms *affine* and *linear* separation logics to distinguish logics that do enjoy weakening and those that do not. The standard term for this distinction is "intuitionistic" *vs.* "classical" logic, but as others argued before,[19] that terminology is mostly based on a coincidence: for the kind of simple separation logic models that were used when the terminology was coined, it is indeed the case that the logic could have *either* weakening *or* excluded middle, but you could not usefully have both in the same logic.[20] Thus, the *absence* of the law of excluded middle (indicating an intuitionistic logic) was used to indicate the *presence* of the weakening rule. However, in logics such as VST and Iris, the excluded middle does not hold for reasons entirely unrelated to weakening: both of these logics are intuitionistic because they use step-indexing in their model. And yet, Iris has weakening and VST does not—clearly, the terminology does not work any more for today's logics.

One proposal for an alternative terminology[21] is to say that logics with weakening are "garbage-collected", while logics without weakening are "malloc/free". This draws a parallel between *programming languages* where just forgetting about some memory location is entirely harmless (*i.e.*, garbage-collected languages) and *logics* where forgetting about some ownership is permitted (*i.e.*, logics with weakening). However, we argue

[15] Appel, "Program logics – for certified compilers", 2014 [App14]; Cao *et al.*, "VST-Floyd: A separation logic tool to verify correctness of C programs", 2018 [Cao+18].

[16] Hobor, "Oracle semantics", 2008 [Hob08], §4.7, p. 88.

[17] Cao, 2017 [Cao17].

[18] Bizjak *et al.*, "Iron: Managing obligations in higher-order concurrent separation logic", 2019 [Biz+19].

[19] Cao, Cuellar, and Appel, "Bringing order to the separation logic jungle", 2017 [CCA17].

[20] Having both weakening and excluded middle in a separation logic implies that separating conjunction and normal conjunction are equivalent, which makes the separation part of separation logic useless [IO01].

[21] Cao, Cuellar, and Appel, "Bringing order to the separation logic jungle", 2017 [CCA17].

that this, too, is not a great choice of terminology. For one, as we have shown above, VST (a "malloc/free" logic in this terminology) does not actually prove that all allocated memory is freed again. Secondly, we have used Iris (a "garbage-collected" logic) without any issue to reason about malloc/free programming languages. All it takes is a proof rule like the following for the deallocation operation:[22]

$$\{\ell \mapsto v\}\,\texttt{free}(\ell)\,\{\mathsf{True}\}$$

With this setup, our proofs in Iris do not verify that all memory is adequately freed, but our proofs *do* show that the program is free of use-after-free or double-free errors.

For these reasons, we argue that neither of the previously proposed terminologies are a good choice to distinguish separation logics with and without weakening. We propose that, instead, we should borrow terminology from the linear logic community, and call a separation logic *affine* if it has weakening, and *linear* if it has not.

[22] In fact, the latest version of HL in our Coq formalization contains such an operation.

Chapter 7

Key differences to prior work

A detailed review of prior work on separation logics (beyond §2) is outside the scope of this dissertation, but in this chapter we will look at a few key aspects in which Iris *differs* from many of its predecessor logics: Iris has no "unstable" assertions (§7.1), Iris has its own distinct set of axioms for its model of resources (§7.2), and Iris uses a substitution-based language with a strict separation of "logic" and "program" variables instead of one with an environment or mutable stack variables (§7.3). Another key difference is that Iris is an affine logic, but this was already discussed in §6.2.

7.1 Stability

One key property of Iris is that every proposition is *stable*: when doing a proof where the current thread owns assertion P, then no action of other threads can invalidate P. This is in contrast to the majority of other CSLs,[1] where the core points-to assertion $\ell \mapsto v$ is stable under actions of other threads, but which also have some form of *unstable* assertion. For example, in iCAP, STSs (called *regions* in the tradition of original CAP) are very similar to what we presented in §4.2, with one key difference: the equivalent of $\mathsf{StsSt}_\gamma(s)$ in iCAP (tokens are handled separately) asserts that we are *exactly* in state s (which is unstable), whereas the Iris variant says we are *at least* in state s (which is stable).

In particular, in the STS from Figure 4.1 (page 49), $\mathsf{StsSt}_\gamma(\mathsf{pending})$ would say that the state, right now, *is* $\mathsf{pending}$. The problem with this statement is that some other thread could make a transition of $\mathsf{pending}$ to $\mathsf{shot}(n)$ any time! To account for this, whenever there could be interference by other threads, logics like iCAP have an explicit side-condition requiring the assertion to be *stable*. Unstable assertions are only allowed within the verification of a single atomic instruction; when composing multiple such Hoare triples, all pre- and postconditions need to be stable.

These logics then typically come with some set of rules defining when an assertion is stable. For example, $\mathsf{StsSt}_\gamma(\mathsf{pending})$ is not stable, but $\mathsf{StsSt}_\gamma(\mathsf{pending}) \lor \exists n.\ \mathsf{StsSt}_\gamma(\mathsf{shot}(n))$ is; another example of a stable assertion is $\mathsf{StsSt}_\gamma(\mathsf{pending}) * [\mathsf{tok}]_\gamma$ (where $[\mathsf{tok}]_\gamma$ expresses ownership of the token of protocol instance γ). These examples show that checking if an assertion is stable requires putting together different parts of the assertion and arguing that, even though these assertions *individually* are all unstable, their *composition* is stable.

[1] Dinsdale-Young *et al.*, "Concurrent abstract predicates", 2010 [Din+10]; Svendsen, Birkedal, and Parkinson, "Modular reasoning about separation of concurrent data structures", 2013 [SBP13]; Raad, Villard, and Gardner, "CoLoSL: Concurrent local subjective logic", 2015 [RVG15]; Svendsen and Birkedal, "Impredicative concurrent abstract predicates", 2014 [SB14]; da Rocha Pinto, Dinsdale-Young, and Gardner, "TaDA: A logic for time and data abstraction", 2014 [dDG14]; Nanevski *et al.*, "Communicating state transition systems for fine-grained concurrent resources", 2014 [Nan+14].

When working on paper, this is "just" an inconvenience, but when working in a proof assistant, stability needs to be formally established *everywhere*. The only mechanized separation logic with unstable assertions we are aware of is FCSL, and in their proofs stability reasoning frequently makes up 20-30% of the size of the proof.[2] This does not even account for the extra cost of tracking stability in the kind of higher-order specification that we often use in Iris (even the example specification from §3 involved a higher-order quantification), where we would need to ensure that we only quantify over *stable* propositions.

In contrast, Iris follows the approach that *everything you can state is stable*. This removes any need for a stability side-condition. In some sense, this is closest to the spirit of original separation logic, which likewise does not have unstable assertions about owned resources[3]—enabling stable statements about mutable locations was arguably one of its key achievements. The price we pay for this is that *changes* become harder as we have to make sure they do not invalidate assertions another thread could be making. Indeed, the notion of a frame-preserving update *precisely* captures those changes that make sure that every possible statement the rest of the proof could make remains true—any possible resource the frame could own remains valid. We find that to be a worthy trade-off, given that most proofs only require a small handful of frame-preserving updates that can be quickly proven by composing general RA properties. This is in contrast to having to verify stability of *every single* proposition that is used in a context where interference from other threads has to be taken into account.

However, the downside of this is that users of Iris have to learn how to wield RA combinators to express the interactions they need for the proof at hand. This is probably the biggest hurdle someone has to overcome to become proficient in Iris. The mindset has to shift from thinking primarily about permissible *transitions* between the states of the data structure, to thinking more about which stable statements each party needs to be able to make and which transitions they have to *rule out* (such as "this one-shot has been initialized to value n and n cannot change any more", or "I own the right to initialize this not-yet-initialized one-shot and nobody else does"), and then turning those statements into RA elements ($\mathsf{shot}(n)$ and $\mathsf{pending}$).

Most of the time, we can express what we need through the combinators described in §4.1 and *Auth*.[4] For the rare case where that does not work, we can always apply the STS construction to turn a description of permissible transitions into an RA that can be used with Iris. This entails a lot of (tedious to mechanize) reasoning about the closure of some set of states under possible transitions of other threads—basically, reasoning about stability. Iris does not remove the cost of that reasoning whenever it is truly needed, but Iris demonstrates that in the vast majority of cases, the power of a general state-transition system is not needed and we can get away with simpler mechanisms that avoid reasoning about stability.

[2] Sergey, Nanevski, and Banerjee, "Mechanized verification of fine-grained concurrent programs", 2015 [SNB15], Table 1.

[3] It *does* however have unstable assertions about local variables; see §7.3 for what we do in Iris instead.

[4] Interestingly, we do not usually obtain an RA whose possible interactions are *equivalent* to the STS one might intuitively draw. The RA usually permits more transitions, but those extra transitions do not actually break anything as long as certain key transitions are ruled out.

7.2 Resource algebra axioms

The axioms for resource algebras (Figure 3.4 on page 31) are geared towards their use in Iris. Here, we compare those axioms with the algebraic structures used by other separation logics.

Core. In prior work, there have been several presentations of generalizations of PCMs that have "multiple units" or include a notion of a "duplicable core".

The terminology of a *(duplicable) core* was introduced by Pottier in the context of *monotonic separation algebras*.[5] VST[6] also comes with its own notion of *separation algebras*[7] that, unlike PCMs, permit every element to have a *different unit*, defined through the "core" function. A related notion also appears in GPS.[8]

Some common properties hold in most of these works (either as axioms or as admissible rules): The core must produce a unit (ra-core-id), be idempotent (ra-core-idem), and be a homomorphism, *i.e.*, be compatible with composition: $|a \cdot b| = |a| \cdot |b|$. The last property is stronger than our monotonicity axiom (ra-core-mono), and as such, the axioms of RAs are weaker than those demanded by prior work. In fact, RAs are *strictly* weaker: as mentioned before, the core of the STS RA (§4.2) is *not*, in general, a homomorphism. This shows that demanding the core to be a homomorphism, as prior work has done, rules out useful instances (*i.e.*, STSs could not have a useful core).

The core axioms of separation algebras underlying VST demand that any two compatible resources have the *same* core. Basically, this means that the core only contains "static", unchanging information that was true from the beginning of the proof. (In VST, this is used to model the persistent knowledge of which global functions exist in the program.) This is unlike in Iris where, for example, $\mathsf{shot}(n)$ is its own core even though n was only picked *during* the verification: the core of an RA can be "dynamic", it can contain information that is true forever into the future but was not necessarily true forever into the past.

Another difference is the fact that our core may be *partial*, whereas in prior work it was always a total function: every element had to have *some* unit. As discussed in §4.1, partial cores play an instrumental role in our approach to compose RAs out of simpler constructions and combinators like sums.

Finally, Bizjak and Birkedal[9] have shown that our notion of the core is precisely what is needed to get a well-behaved notion of persistent propositions, *i.e.*, a persistence modality ($\square$) that commutes with both universal and existential quantification.

Cancellativity. Many separation logics, including VST,[10] require resource composition to be *cancellative*:

$$\forall a, b, c.\, a \cdot b = a \cdot c \Rightarrow b = c$$

This implies that the separating conjunction itself is also cancellative, which is needed (together with *precision* of resource invariants, as men-

[5] Pottier, "Syntactic soundness proof of a type-and-capability system with hidden state", 2013 [Pot13].

[6] Appel, "Program logics – for certified compilers", 2014 [App14].

[7] Dockins, Hobor, and Appel, "A fresh look at separation algebras and share accounting", 2009 [DHA09].

[8] Turon, Vafeiadis, and Dreyer, "GPS: Navigating weak memory with ghosts, protocols, and separation", 2014 [TVD14].

[9] Bizjak and Birkedal, "On models of higher-order separation logic", 2017 [BB17].

[10] Calcagno, O'Hearn, and Yang, "Local action and abstract separation logic", 2007 [COY07]; Dockins, Hobor, and Appel, "A fresh look at separation algebras and share accounting", 2009 [DHA09].

tioned in §5.1) to justify soundness of separation logics that support the conjunction rule:[11] $\{P\}\, e\, \{Q_1\} \wedge \{P\}\, e\, \{Q_2\}$ implies $\{P\}\, e\, \{Q_1 \wedge Q_2\}$. However, it is increasingly common for separation logics to not provide the conjunction rule, and many of these logics do not require cancellativity.[12] Recently, this axiom has also been dropped from VST as it turned out not to be necessary.

The same applies to Iris, where the conjunction rule is invalidated by frame-preserving updates: the proofs of the two Hoare triples could make different frame-preserving updates, making the conjunction of their postconditions contradictory. Hence, RAs are not required to be cancellative. In fact, we regularly use non-cancellative RAs such as natural numbers with max for composition (which models a monotone counter).

Composition as relation or partial/total function. A notable difference between VST's separation algebras and RAs is that we present the monoidal operation $(\cdot)$ as a *total* function, whereas they represent this operation as a ternary functional relation (that may be partial). It is also common, *e.g.*, in Views,[13] to model composition as a *partial* function (partial functions and functional relations are isomorphic in set theory, but still fundamentally different when formalized in a type theory such as the one of Coq). In RAs, the validity predicate carries the information that would usually be modeled through partiality.

The biggest benefit of making composition a total function is that we can reason equationally, which makes it more convenient to carry out proofs. Moreover, it is easy to implicitly assume that a partial function is defined everywhere we care about when working on paper; however, when carrying out mechanized proofs, this becomes quite burdensome. The approach of using a total function with a validity predicate also scales neatly to the notion of "step-indexed resource algebras" that is required for higher-order ghost state.[14]

On the other hand, defining RAs like Ag (the variant of Ag_0 from §4.1 that supports higher-order ghost state) is quite challenging when composition has to be a function in Coq's type theory. Still, we find that being able to directly compute and work with *the* uniquely defined composition of two elements outweighs those problems.

One interesting recent development worth mentioning here is the work by Krishna, Shasha, and Wies[15] on *flow*: a PCM-like structure that provides interesting ways to decompose a graph. Flow comes with a composition relation that is *not functional*, and as such does not fit in any of the algebraic structures discussed above. Flow has since been used in Iris as well,[16] but the authors had to resort to making an RA for *sets* of flows such that composition could be cast as a function.

Frame-preserving updates. Frame-preserving updates in RAs are basically the same as *semantic entailment* in the Views framework,[17] which correspond to the *action judgment* of an operation that does nothing (much like view shifts are very similar to Hoare triples for an expression that does nothing). However, where Views is basically a general guide for

[11] Gotsman, Berdine, and Cook, "Precision and the conjunction rule in concurrent separation logic", 2011 [GBC11]; Vafeiadis, "Concurrent separation logic and operational semantics", 2011 [Vaf11].

[12] Jensen and Birkedal, "Fictional separation logic", 2012 [JB12]; Pottier, "Syntactic soundness proof of a type-and-capability system with hidden state", 2013 [Pot13]; Dinsdale-Young *et al.*, "Views: Compositional reasoning for concurrent programs", 2013 [Din+13].

[13] Dinsdale-Young *et al.*, "Views: Compositional reasoning for concurrent programs", 2013 [Din+13].

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

[15] Krishna, Shasha, and Wies, "Go with the flow: Compositional abstractions for concurrent data structures", 2018 [KSW18].

[16] Krishna, Summers, and Wies, "Local reasoning for global graph properties", 2020 [KSW20].

[17] Dinsdale-Young *et al.*, "Views: Compositional reasoning for concurrent programs", 2013 [Din+13].

how to build a *model* of a separation logic with the user "bringing their own logic", Iris is designed to be itself a very general separation logic.

The main limitation of Views is that it effectively requires the user to bake in a single, fixed invariant tying logical to *physical* resources (the "reification function"), with no support for layering further invariants on top of those logical resources. Iris, in contrast, provides logical support for user-defined invariants over *logical* (ghost) resources. We have seen in §4.4 how this mechanism can be used to build an abstraction layer that employs *fictional separation*; this approach can be layered on top of itself any number of times.

The closest equivalent to Views' *reification function* in Iris is the *state interpretation* of the weakest precondition;[18] both are basically relations between logical and physical states. But the state interpretation is just an implementation detail of weakest preconditions, and most Iris users do not have to be concerned with it at all. However, recent discussions on the Iris-Club mailing list[19] also raised some interesting ideas of generalizing *Auth* (§4.4) to incorporate a reification function, which could help cover some use-cases that so far required specialized constructions. Indeed, the authoritative RA can be seen as a special case of a "view" where reification is the identity function. Generalizing this to a more user-controlled form of reification would, in some sense, "embed" a simple form of the Views framework as an RA into Iris. Exploring this is left as future work.

7.3 Substitution-based language

The idealized language we use in Iris (HL/HeapLang, §3) is rather atypical when compared with the idealized languages of other concurrent separation logics: local variables are immutable and defined through substitution like in the lambda calculus, instead of being mutable and defined via some kind of "environment" which maps variable names to their current value.

The environment-based approach is meant to model mutable stack-allocated variables of languages such as Java or C.[20] These variables are typically deeply integrated into the program logic in the sense that the pre- and postconditions of Hoare triples can directly refer to their current value, implicitly referring to the environment that maps variable names to values. As a consequence, these logics have to add side-conditions to rules such as the frame rule to make sure that the framed-away part does not refer to variables that get changed, and the rules for mutating local variables also become rather awkward as they have to ensure that *every* reference to this variable is now adequately updated.

Already the original paper on CSL[21] remarks:

> Finally, the presence of the variable side conditions in the frame rule and the concurrency proof rules is a continuing source of embarrassment.

Iris does not require any such side-conditions, and since local variables cannot be mutated, there is no need for an awkward proof rule to account for this operation. All mutation has to happen on the heap. This sounds like it could become very tedious, and indeed the CSL paper theorizes that this approach "trades simplicity for the logician (simpler looking

[18] Krebbers *et al.*, "The essence of higher-order concurrent separation logic", 2017 [Kre+17].

[19] Malecha, "Two monoid questions", 2019 [Mal19].

[20] However, the model is inadequate for C because these stack-allocated variables are not addressable, *i.e.*, one cannot create a pointer to them.

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

proof theory) for complexity for the programmer". However, in practice, we find this system very easy to work with.

For the programmer, this results in an ML-style language. While implicitly making all local variables mutable (as is the case in most languages) can sometimes be useful, OCaml programs do not become prohibitively more complex just because such mutation has to be performed explicitly on the heap. Moreover, OCaml is not alone here: for example, Rust (a modern language with a C-like programming model for mutable local variables) makes local variables immutable by default. Programmers can opt-in to mutability, but in idiomatic Rust code, only a fraction of variables actually gets mutated; the rest can be treated like an immutable ML-style variable just fine. Of course, any Haskell programmer would be bewildered by the idea that local variables could ever be mutable. All of this goes to show that immutable local variables are perfectly practical.

On the specification side, restricting mutation to the heap is usually not a great burden either, because only variables that actually need to be mutated require dealing with points-to-assertions. When variables are used to carry around pure values (*e.g.*, for arithmetic), they are immutable and can be directly referenced in specifications without any friction.

The most prominent prior approach to solve the problem of the frame rule side conditions is to treat stack variables as a resource.[22] However, this requires explicitly stating in specifications which local variables are owned. In "Variables as resource in separation logic" [BCY05], the authors remark that the alternative of treating *every* variable as allocated on the heap would turn even simple assertions such as $x > y$ into $\exists X, Y.\, x \mapsto X * y \mapsto Y * X > Y$, and indeed that would be quite painful—but since substitution-based languages still support referencing *immutable* local variables directly in specifications,[23] this is not typically what happens. Instead, x and y will be passed around by-value, leading to a specification of $x > y$, which is even simpler than variables-as-resource. Passing around points-to facts is only required for pass-by-reference, when the callee is expected to actually mutate the pointed-to value as a side-effect—such operations are typically avoided anyway to keep the code cleaner, and the vast majority of specifications in HeapLang are as simple as they would be with original CSL.

Another benefit of using substitution is that distinguishing "program" variables (that are bound in the verified expression and can only be used there) from "logical" variables (that are bound by logical quantifiers or in the current logical context and can be used in any logical term, including pre-/postconditions and, crucially, the expression that is being verified) does not actually introduce any complications, not even during mechanization. One might expect that having to care about the distinction between program variables and logical variables would create some extra mental and formal burden that one can only gloss over when working on paper. However, it turns out that this is not the case, as is demonstrated by the following derived proof rule:

[22] Bornat, Calcagno, and Yang, "Variables as resource in separation logic", 2005 [BCY05]; Parkinson, Bornat, and Calcagno, "Variables as resource in Hoare logics", 2006 [PBC06]; Brookes, "Variables as resource for shared-memory programs: Semantics and soundness", 2006 [Bro06].

[23] Or rather, we use *logical* variables to abstract over the values of these program variables—see the next paragraph.

$$\frac{\{P\}\, e_1\, \{v.\, Q\} \qquad \forall v.\, \{Q\}\, e_2[v/x]\, \{w.\, R\}}{\{P\}\, \texttt{let}\, x = e_1; e_2\, \{w.\, R\}}\ \text{HOARE-LET}$$

This rule basically says that when stepping over a let-bound variable x in the program, that program variable "becomes" a logical variable v![24] The use of v in the substitution operation $e_2[v/x]$ is just like the possible uses of v in Q and R: the logical variable represents *any* actual program-level value. The substitution replaces the term x of type "program variable" (as far as the logic is concerned, this is a closed term) with "whatever v represents", which is of type *Val*.

In practice, this means that one does not have to think about two kinds of variables: as soon as a program-level binder comes into scope, it turns into a logical binder and can be used freely in any logical term. We exploited this on paper before having the infrastructure to carry out Iris proofs in Coq, going so far as to write x instead of v, entirely conflating the two kinds of variables. We were worried that this sloppiness would cause problems when mechanizing these proofs, but the opposite was the case: even when doing fully mechanized proofs in Coq, this causes absolutely no hassle.

To summarize, immutable local variables do not impose an undue burden on the programmer, and defining their semantics by substitution means that one never has to think about the distinction between logical and program-level variables. The main benefit of introducing this distinction is a simpler program logic (no side-conditions about free program variables). Immutable local variables also lead to a simpler programming language with only one source of mutability.[25] In our opinion, one should have very good reasons to deviate from this approach, such as having to precisely model some real-world language in order to verify real-world code. Otherwise, the simplifications afforded by relying on substitution and restricting mutability to the heap are a clear win both for the language and the program logic.

[24] In Coq, what happens is that x is a string, representing a variable name in the deeply embedded formalization of HL, and v is just a Coq-level variable.

[25] One might be concerned about running into the usual issues around representing binders and handling capturing when one has to formalize substitution in a proof assistant. However, the programming languages we are considering are all call-by-value and we are not verifying open programs, so substitution only ever acts on *closed* terms and capturing is not an issue. Using simple strings to represent binders and a naive implementation of substitution works fine. This is inspired by "Software foundations (volume 2): Programming language foundations" [Pie+19].

Part II

RustBelt

RustBelt provides a formal model of Rust, with the goal of establishing the safety of the core Rust type system and some key **unsafe** parts of the Rust standard library. For motivation and key challenges of RustBelt, see the introduction in §1.2. At the heart of RustBelt is λ_{Rust}, a Rust core language, with a type system modeling Rust-style ownership and borrowing, together with a semantic interpretation ("logical relation") of that type system.

The second part of this dissertation begins with a brief Rust tutorial in §8, enough so that readers unfamiliar with the language are able to follow the subsequent chapters. Those chapters explain the various technical aspects of RustBelt: the formal language that serves as a model of Rust and its type system (§9), the lifetime logic that enables separation-logic reasoning about borrowing (§11), the logical relation that gives semantic meaning to the type system (§10 and §12), and finally the verification of `Cell` and `Mutex` as two representative (but still comparatively simple) unsafely implemented abstractions in the Rust standard library (§13). Finally, we consider related work (§14).

Chapter 8

Rust 101

In this chapter, we give a brief overview of some of the central features of the Rust type system, insofar as they are needed to understand RustBelt and Stacked Borrows. We do not assume any prior familiarity with Rust.

The key principle of the Rust type system is the following *exclusion principle*, also known as *mutability XOR aliasing*: data can *either* be mutated exclusively through *one* unique pointer, *or* it can be *immutably* shared amongst many parties—but not both at the same time. (There is an exception to this that we will come back to later.) This obviously prevents data races by ruling out mutable state shared across thread boundaries, but as we will see the exclusion principle also rules out other errors commonplace in low-level pointer-manipulating programs, like use-after-free and double-free.

For most of the rest of this chapter, we will discuss the two key mechanisms through which Rust enforces this principle: *ownership* and *borrowing*. Afterwards, we will briefly mention the exception to the rule.

8.1 Ownership and ownership transfer

In Rust, the type of a variable not only represents information about the values it can take—it also represents *ownership* of resources related to the variable such as memory or file descriptors. When a variable gets passed between functions or threads, the associated ownership is *transferred* or "moved" with it. To see this principle in practice, consider the following sample program:

```
let (snd, rcv) = channel();
join(
  move || {
    let mut v = Vec::new(); v.push(0); // v: Vec<i32>
    snd.send(v);
    // v.push(1); gives compiler error: v has been moved
  },
  move || {
    let v = rcv.recv().unwrap(); // v: Vec<i32>
    println!("Received: {:?}", v);
  }
);
```

Before we take a detailed look at the way the Rust type system handles ownership, we briefly discuss the syntax and functions used here. `let` is used to introduce local, stack-allocated variables. These can be made mutable by using `let mut`. The function `channel` creates a typed multi-producer single-consumer channel and returns the two endpoints as a pair. The first line uses a pattern to immediately destruct this pair into its components. The function `join` is essentially parallel composition; it takes two closures and executes them in parallel, returning when both are done.[1] The vertical bars `||` mark the beginning of an anonymous closure; if the closure would take arguments, they would be declared between the bars. The `move` keyword indicates that ownership of the captured variables is fully moved into the closure.[2]

In this example, the first thread creates a new empty vector `v` of type `Vec<i32>` (a resizable heap-allocated array of 32-bit signed integers, corresponding to `std::vector<int>` in C++) and pushes an element onto it. Next, it sends `v` over the channel by passing it to `send`. Meanwhile, the second thread receives a vector from the channel by calling `recv`. However, `recv` is a fallible operation, so we call `unwrap` to trigger a *panic* (which aborts execution of the current thread) in case of failure. Finally, we print a debug representation of the vector (as indicated by the format string `"{:?}"`).

To see the type system in action, we consider what happens when we change the sending thread to access `v` again in line 6 after sending it to the other side: in that case, the typechecker shows an error, saying that `v` has been "moved". And it is indeed important that the typechecker shows an error here, because when we sent `v` to the other thread, we did not copy the underlying array that was allocated on the heap. In Rust, parameter passing is *shallow*, which means that data behind pointer indirections is not duplicated. This is much more efficient than a full copy, but it means that both variables `v` in both threads actually point to the same underlying data![3] Since accesses in Rust are *non-atomic* by default, there is no synchronization and concurrent accesses to that shared data would lead to data races.[4] To avoid such problems, Rust prevents access to `v` after it has been sent.

But how is the typechecker detecting this problem? In Rust, having a value of some type indicates that we are the *exclusive owner* of the data described by said type, and thus that *nobody else* has any kind of access to this array, *i.e.*, no other part of the program can write to or even read from the array. When passing a variable like `v` to a function like `send`, ownership of the data is considered to have *moved*, and is thus no longer available in the caller.[5] Thus, in line 6 `v` has been "given away" and cannot be used any more, which leads to the error flagged by the typechecker.

Automatic destructors. One aspect of low-level programming that is distinctively absent in the code above is memory management. Rust does not have garbage collection, so it may seem like our example program leaks memory, but that is not actually the case: due to ownership tracking, Rust can tell when a variable (say, the vector `v` in the receiving thread) goes out of scope without having been moved elsewhere. When that is the case, the compiler automatically inserts calls to a *destructor*, called `drop` in Rust. For example, when the second thread finishes in line 11, `v`

[1] `join` is not in the Rust standard library, but part of Rayon [SM17], a library for parallel data processing.

[2] The alternative would be for the closure to *borrow* the variables, which we will get to later.

[3] Of course, Rust provides a way to do a *deep copy* that actually duplicates the array on the heap, but it will not do this implicitly—deep copies can become performance bottlenecks, so Rust wants them to be visible in the code.

[4] This refers to the distinction the C++ concurrent memory model (which Rust inherits) does between *atomic* accesses, which are permitted to take part in race conditions and can be used for communication between threads, and *non-atomic* accesses. A race condition on a non-atomic access is called a *data race* and considered undefined behavior, which means the compiler is free to assume that data races do not happen.

[5] In technical terms, Rust's variable context is *substructural*.

is dropped. Similarly, the sending and receiving ends of the channel are dropped at the end of their closures. This way, Rust provides automatic memory management without garbage collection, and with predictable runtime behavior. The same approach can also automatically manage other resources such as file descriptors which typical garbage collectors do not provide any help with.

8.2 Mutable references

Exclusive ownership is a fairly straightforward mechanism for ensuring data-race freedom. However, it is also very restrictive. In fact, close inspection shows that even our first sample program does not strictly follow this discipline. Observe that in line 4, we are calling the method `push` on the vector `v`—and we keep using `v` afterwards. It would be very inconvenient if pushing onto a vector required explicitly passing ownership to `push` and back. Rust's solution to this issue is *borrowing*, which is the mechanism used to handle reference types.[6] The idea is that `v` is not *moved* to `push`, but instead *borrowed*—granting `push` access to `v` *for the duration of the function call.*

This is expressed in the type of `push`: `fn(&mut Vec<i32>, i32)`.[7] The syntax `v.push(0)`, as used in the example, is just syntactic sugar for `Vec::push(&mut v, 0)`, where `&mut v` creates a mutable reference to `v`, which is then passed to `push` (*i.e.*, `v` is passed by reference). A mutable reference grants *temporary exclusive access* to the vector, which in the example means that access is restricted to the duration of the call to `push`. Another way to say this is that a mutable reference is a *unique pointer*—there cannot be an alias elsewhere in the program that can be used to access the same data. Because the access is temporary, our program can keep using `v` when `push` returns. Moreover, the exclusive nature of this access guarantees that no other party will access the vector in any way during the function call, and that `push` cannot keep copies of the pointer to the vector. Mutable references are always unique pointers.

The type of `send`, `fn(&mut Sender<Vec<i32>>, Vec<i32>)`, shows another use of mutable references. The first argument is just *borrowed*, so the caller can use the channel again later. In contrast, the second argument is *moved*, using ownership transfer as already described above.

8.3 Shared references

Rust's approach to guaranteeing the absence of races and other memory safety issues is to rule out the combination of aliasing and mutation—what we called the *exclusion principle* above. So far, we have seen unique ownership (§8.1) and (borrowed) mutable references (§8.2), both of which allow for mutation but prohibit aliasing. In this section we discuss another form of references, namely *shared references*, which form the dual to mutable references: they allow aliasing but prohibit mutation.

Like mutable references, shared references grant *temporary* access to a data structure, and operationally correspond to pointers. The difference is in the guarantees and permissions provided to the receiver of the reference.

[6] Other solutions to this problem exist; *e.g.*, Mezzo [BPP16] provides a mechanism for making ownership transfer more implicit.

[7] We follow the usual Rust style and omit the return type if it is the unit type `()`.

While mutable references are exclusive (non-duplicable), shared references can be *duplicated*. In other words, shared references permit aliasing. As a consequence, to ensure data-race freedom and memory safety, shared references are *read-only*.

We can see shared references in action in the following example:

```
1 let mut v = Vec::new(); v.push(1);
2 join(
3   || println!("Thread 1: {:?}", &v),
4   || println!("Thread 2: {:?}", &v)
5 );
6 v.push(2);
```

This program starts by creating and initializing a vector `v`. Then it passes a shared reference `&v` to two threads, which concurrently print the contents of the vector. This time, the closures are not marked as `move`, which leads to `v` being captured by-reference, *i.e.*, at type `&Vec<i32>`. As discussed above, this type is duplicable, so the type checker accepts using `&v` in both threads.

The concurrent accesses to `v` use *non-atomic* reads, which have no synchronization. This is safe because when a function holds a shared reference, it can rely on the data-structure not being mutated—so there cannot be any data races.[8]

Finally, when `join` returns, the example program re-gains full access to the vector `v` and can mutate `v` again in line 6. This is safe because `join` will only return when both threads have finished their work, so there cannot be a race between the `push` and the `println`. This demonstrates that shared references are powerful enough to *temporarily* share a data structure and permit unrestricted copying of the pointer, but regain exclusive access later.

Duplicable types and `Copy`*.* In Rust, properties of a type such as "duplicable" are expressed using *marker traits*. Traits in Rust are a form of typeclasses and they are the primary abstraction mechanism of the language. In general, `T: Trait` is a way to say that trait `Trait` is implemented for type `T`, and thus the operations of the trait can be used on variables of type `T`. In the special case of a *marker trait*, there actually is no operation—the trait simply classifies types into those that implement the trait and those that do not.

In particular, the built-in `Copy` marker trait is used to express that a type is duplicable.[9] Specifically, if a type implements `Copy`,[10] it means that doing a shallow copy (which, remember, is what Rust does to pass arguments) suffices to duplicate elements of the type. Both `&T` and `i32` are `Copy` (for any `T`)—however, `Vec<i32>` is not! The reason for this is that `Vec<i32>` stores data on the heap, and a shallow copy does not duplicate this heap data.

8.4 Interior pointers

So far, we used prevention of data races as the primary motivation for Rust's exclusion principle, and indeed ruling out shared mutable state is

[8] Notice that this is a much stronger guarantee than what C provides with `const` pointers: in C, `const` pointers prevent mutation by the current function, however, they do *not* rule out mutation by *other* functions.

[9] Technically speaking, `Copy` types behave like unrestricted variables in linear type systems.

[10] We also sometimes say "the type *is* `Copy`", because there is nothing to actually implement.

exactly what we need to ensure that non-atomic accesses do not cause data races. A frequent question at this point is: can the rules be relaxed for sequential programs? Do we really need to ensure that mutable references are unique when everything is happening in the same thread, so there cannot be any data races? The answer is yes, the exclusion principle is crucial even in sequential code, because of *interior pointers*—pointers that point *into* another data structure.

This is demonstrated by the following example, which we have already seen in the introduction:[11]

```
1  let mut v = vec![10, 11];
2  let vptr = &mut v[1]; // Points *into* v.
3  v.push(12); // May reallocate the backing store of v.
4  println!("v[1] = {}", *vptr); // Compiler error!
```

Here, `v` is again of type `Vec<i32>`. `vptr` is a reference of type `&mut i32` pointing *into* the allocation where `v` stores its data. Such *interior pointers* are generally not supported in garbage-collected languages, but they are widely used in languages such as C, C++, and Rust to achieve a more compact and cache-friendly layout of data structures.

The problem with interior pointers becomes apparent in line 3 of the example: `v.push` might have to allocate new space elsewhere on the heap if there is not enough space for a third element in the original location. The situation looks just like in the C++ example in Figure 1.1 (page 2). By mutating `v`, we inadvertently deallocated the memory that `vptr` points to. This is a memory safety violation, so the Rust typechecker has to be able to detect it!

And indeed the Rust compiler refuses to compile this program, complaining that we “cannot borrow `v` as mutable more than once at a time” in line 4. The way it does that is by detecting a violation of the exclusion principle: in line 3, *both* `vptr` and `v` point (directly or indirectly) to the same memory, namely the array on the heap where the content of `v` is stored. And in that same line, we are also performing a *mutation* of `v`. This means we are trying to mutate data even though our pointer is not unique, and this is a condition that the Rust typechecker can detect and prevent.

The example demonstrates that the exclusion principle is crucial even in sequential code.[12] However, to explain better how exactly the Rust typechecker goes about detecting the conflict between `vptr` and `v` here, we need to introduce the concept of a *lifetime*.

8.5 Lifetimes

As previously explained, (mutable and shared) references *borrow* ownership and thus grant *temporary* access to a data structure. Lifetimes answer the question: just how long is *temporary*? The full form of a reference type is actually `&'a mut T` or `&'a T`, where `'a` is the lifetime of the reference. Rust uses a few conventions so that lifetimes can be elided in general, which is why they did not show up in the programs and types we considered so far. However, lifetimes play a crucial role in explaining how the example in §8.4 got rejected.

[11] As discussed before, this is basically a simplified variant of the classical “iterator invalidation” problem, where the pointer `vptr` plays the role of the iterator.

[12] Many other examples of this exist, even some that do not require heap allocation and work exclusively with variables on the stack. They always involve mutating operations that “invalidate” pointers or iterators in some way—*e.g.*, by reallocating some array on the heap as in our example, or by changing a variable of sum type to a different variant, which invalidates interior pointers into the previously active variant.

To ensure references get used correctly, the compiler enforces the following two constraints:

1. The reference can only be used while its lifetime is ongoing, and
2. the original referent is not used at all (for mutable references) or does not get mutated (for shared references) until the lifetime of the newly created reference has expired.

The analysis checking these properties is called *borrow checking*. The borrow checker essentially collects all these constraints and ensures that there is a consistent way to assign lifetimes to references. To check (2), the borrow checker also attaches a *loan* to the original referent of a new reference. The loan has the same lifetime as the created reference, and using the referent again induces a constraint that at this program point, the lifetime of all loans (for a mutating use) or of all mutable loans (for a read-only use) must have ended.[13]

In our slightly desugared example program, the compiler infers the following lifetimes:[14]

```
1  let mut v = vec![10, 11];
2  let vptr = &'a mut v[1];
3  Vec::push(&'b mut v, 12);             Lifetime 'b
4  println!("v[1] = {}", *vptr);              Lifetime 'a
```

The lifetime 'b for the second borrow just spans the call of push in line 3; there are no further restrictions, so the compiler makes 'b as short as possible. However, in line 4 we use the reference vptr with lifetime 'a, so to satisfy rule (1) we are forced to extend 'a until line 4. But this means that when a reference to v gets created in line 3 (which counts as a "use" of v), we still have an outstanding loan of v[1] for lifetime 'a! This violates condition (2) of that loan, so the program gets rejected.

One way to avoid this problem is to reorder lines 2 and 3 of the original example:

```
1  let mut v = vec![10, 11];
2  Vec::push(&'b mut v, 12);           Lifetime 'b
3  let vptr = &'a mut v[1];
4  println!("v[1] = {}", *vptr);       Lifetime 'a
```

Now each time a new reference gets created, there is no outstanding loan: the loan of v for lifetime 'b ends before vptr gets created. Hence the borrow checker accepts this program.

Reborrowing. From the above example, it may seem like lifetimes of mutable references can never overlap, but that is not actually the case: when we create a reference from an existing reference, their lifetimes will actually be *nested*. For example:

```
1  let mut v = vec![10, 11];
2  let v2 = &'a mut v;
3  let vptr = &'b mut (*v2)[1];
4  println!("v[1] = {}", *vptr);       Lifetime 'b
5  Vec::push(v2, 12);                        Lifetime 'a
```

Here, vptr gets *derived from* v2. This operation is also called a *reborrow*. To handle reborrowing, condition (1) from above gets extended to say

[13] Notably absent from these rules is any mentioning of aliasing. Indeed, everything the compiler needs to know about aliasing is abstracted away through lifetimes and loan tracking.

[14] &'b mut v is meant to say "create a mutable reference with lifetime 'b". This is not actual Rust syntax, we just use it for illustration purposes.

that both the reference with lifetime 'a *and all references derived from it* may be used only while 'a is ongoing. This ensures that we cannot use reborrowing to "escape" usage restrictions of the parent pointer. Practically speaking, this means that the lifetime 'b of the "inner" reference vptr must *end before* the lifetime 'a of the "outer" reference v2. This gives rise to an "inclusion relation" between lifetimes: we say that 'b is *included in* 'a.[15]

Notice how in the above example v2 and vptr are in some sense aliasing, even though they are both mutable references which ought to be unique. This is allowed because vptr is explicitly created from v2, so the compiler *knows* that these mutable references alias. At any given place in the code, one or the other may be used, but never both: while 'b lasts, vptr may be used but v2 may not, and once 'b ends, v2 may be used again but vptr may not. In that sense, both references are still unique.

This can also be seen when we try to add another use of vptr at the end. Indeed, this would introduce a bug into our program: pushing to v2 in line 5 might invalidate vptr, just like in our very first example! Rust statically detects such a bug just as before: when creating vptr, a loan gets attached to v2 remembering that this reference has been "given away" and may not be used until 'b is over. So if 'b were to be extended past line 5, then in line 5 the borrow checker would detect that v2 is being used while there exists an outstanding loan of it, and that would be flagged as an error for violating condition (2).

Shared references. As we have already seen, many shared references may co-exist at the same time. This means they can give rise to overlapping (non-nested) lifetimes:

```
1  let mut v = vec![10, 11];
2  let vptr = &'a v[1];
3  let vptr2 = &'b v[1];
4  println!("v[1] = {}", *vptr);          Lifetime 'a
5  println!("v[1] = {}", *vptr2);   Lifetime 'b
```

Here, we create two shared references to the same element of v, and the calls to println demonstrate that they can both be used in an interleaved fashion.

The borrow checker performs almost the same checks as for mutable references, except that condition (2) gets weakened to ensuring that the referent does not get *mutated* until the lifetime of the loan has expired. So in line 2, a *shared* loan with lifetime 'a gets attached to v[1], which is still outstanding when v[1] gets used again in line 3. However, creating another shared reference to v[1] counts as a non-mutating use of it, and so there is no conflict between the loan and the operation. Thus we just add a second loan to v[1], this time for lifetime 'b. Only once *both* loans have expired may v[1] be mutated again.

If we added a mutation of v (say, v.push(12)) between lines 3 and 4, the program would get rejected: such a mutation would be in conflict with the outstanding shared loans with lifetimes 'a and 'b. The compiler will also reject any attempt to write through a shared reference.

[15] Rust actually uses the dual relation and speaks about 'a *outliving* 'b. This regularly leads to confusion when talking about variance. Here we follow the common convention in mathematics to make the "less-than (or equal)" relation the core primitive.

Raw pointers. Sometimes, the aliasing discipline for references that the borrow checker enforces is too strict. One typical example of this is a data structure where the pointers form a cycle: this leads to there being many paths to the same data (*i.e.*, there is aliasing), and hence the borrow checker will rule out all mutation. In this case, one option is to not use references but instead employ *raw pointers*. Raw pointers in Rust basically act like pointers in C: they are not checked by the compiler, and there are no aliasing restrictions. For example, the following code demonstrates a legal way to cause aliasing with raw pointers:

```
1  let mut x = 42;
2  let ptr1 = &mut x as *mut i32; // create first raw pointer
3  let ptr2 = ptr1; // create aliasing raw pointer
4  // Write through one pointer, observe with the other pointer.
5  unsafe { *ptr1 = 13; }
6  unsafe { println!("{}", *ptr2); } // Prints "13".
```

In line 2, we cast a reference to raw pointer type. Raw pointers can be duplicated (there is no ownership attached to them), so we can just copy that pointer in line 3. Then we can use both pointers despite their aliasing, just as we would in C. However, we need to put these uses of raw pointers inside an `unsafe` block to acknowledge that we did our due diligence and manually ensured safety of all involved operations: since the borrow checker does not do any tracking for raw pointers, the compiler cannot itself guarantee that dereferencing them will not cause problems.

8.6 Interior mutability

So far, we have seen how Rust ensures memory safety and data-race freedom by ruling out the combination of aliasing and mutation. However, there are cases where shared mutable state is actually needed to (efficiently) implement an algorithm or a data structure. To support these use-cases, Rust provides some primitives providing shared mutable state. All of these have in common that they permit *mutation* through a *shared reference*—a concept called *interior mutability*.

At this point, one may be wondering—how does this fit together with the exclusion principle and the story of mutation and aliasing being the root of all memory and thread safety problems? The key point is that these primitives have a carefully controlled API surface. Even though mutation through a shared reference is unsafe *in general*, it can still be safe when appropriate restrictions are enforced by either static or run-time checks. This is where we can see Rust's "extensible" approach to safety in action. Interior mutability is not wired into the type system; instead, the types we are discussing here are implemented in the standard library using `unsafe` code (which we will verify in §13).

Cell. The simplest type with interior mutability is `Cell`:

```
let c1: &Cell <i32> = &Cell::new(0);
let c2: &Cell <i32> = c1;
c1.set(2);
println!("{:?}", c2.get()); // Prints 2
```

The `Cell<i32>` type provides operations for storing and obtaining its content: `set`, which has type `fn(&Cell<i32>, i32)`, and `get`, which has type `fn(&Cell<i32>) -> i32`. Both of these only take a shared reference, so they can be called in the presence of arbitrary aliasing. After we just spent several pages explaining that safety in Rust arises from the exclusion principle, now we have `set`, which seems to completely violate this principle. How can this be safe?

The answer to this question has two parts. First of all, `Cell` only allows getting a *copy* of the content via `get`; it is not possible to obtain an *interior pointer* into the content. This rules out the host of problems related to invalidation of interior pointers that we mentioned in §8.4. To permit these copies, `get` requires the content of the `Cell` to be `Copy`. In particular, `get` cannot be used with cells that contain non-`Copy` types like `Vec<i32>`.[16]

However, there is still a potential source of problems, which arises from Rust's support for multithreading. In particular, the following program must *not* be accepted:

```
let c = &Cell::new(0);
join(|| c.set(1), || println!("{:?}", c.get()));
```

The threads perform conflicting unsynchronized (non-atomic) accesses to `c`, *i.e.*, this program has a data race. To rule out programs like the one above, Rust employs two marker traits: `Send` and `Sync`.

The `Send` marker trait encodes if elements of a type are "sendable to another thread". For example, `Vec<i32>` is `Send` because when the vector is moved to another thread, the previous owner is no longer allowed to access the vector—so it is fine for the new owner, in a different thread, to perform any operation whatsoever on the vector. Likewise, `Cell<i32>` is `Send`, reflecting that moving ownership of a `Cell<i32>` to another thread is safe: the original thread loses access to it and the new thread gains access, *i.e.*, there is no risk of a data race.[17]

The type of `join` demands that the environment captured by the closure satisfies `Send`. However, in our example, the closures are not capturing a `Cell<i32>`—they are capturing a shared reference of type `&Cell<i32>`.[18] This is where the second marker trait comes in, `Sync`.

`Sync` encodes if elements of a type may be *shared* across thread boundaries. Where `Send` is about accessing data first in one thread and later in another thread, `Sync` is about accessing the same data from multiple threads at the same time. For example, `Vec<i32>` is `Sync` because shared references only permit reading the vector, and it is fine if multiple threads do that at the same time. However, `Cell<i32>` is *not* `Sync` because `set` (which requires only a shared reference) is not thread safe. Whether `T` is `Sync` determines whether `&T` is `Send`: a shared reference is "sendable" if and only if the referent may be shared. This means that our reference `&Cell<i32>` is not `Send`, which leads to the program above being rejected.

Mutex. The `Cell` type is a great example of interior mutability and a zero-cost abstraction as it comes with no overhead: `get` and `set` compile to plain unsynchronized accesses, so the compiled program is just as efficient as a C program using shared mutable state. However, as we have seen,

[16] Other methods can still be used on `Cell<Vec<i32>>` though, such as `replace` which overwrites the content of a `Cell` and returns the old content.

[17] An example of a type that is not `Send` is `Rc<i32>`, which is a reference-counted pointer to an `i32`. This reference counting is not thread safe, so even full ownership of one pointer to this object is not sufficient to safely send it to another thread. (Rust also provides the thread-safe alternative `Arc<i32>`.)

[18] The compiler notices that the closures only use shared references of `c`, and hence captures it by-reference.

`Cell` pays for this advantage by not being thread safe. The Rust standard library also provides primitives for thread-safe shared mutable state. Here we take a closer look at `Mutex`, which implements mutual exclusion (via a standard lock) for protecting access to some shared memory. Consider the following example:

```
 1  let mutex = Mutex::new(Vec::new());
 2  join(
 3    || {
 4      let mut guard = mutex.lock().unwrap();
 5      guard.push(0)
 6    },
 7    || {
 8      let mut guard = mutex.lock().unwrap();
 9      println!("{:?}", guard)
10    }
11  );
```

This program starts by creating a mutex of type `Mutex<Vec<i32>>` initialized with an empty vector. In Rust, somewhat unconventionally, we think of `Mutex` as "containing" the data that it guards.[19] The mutex is then shared between two threads (implicitly relying on `Mutex<Vec<i32>>` being `Sync`). The first thread acquires the lock, and pushes an element to the vector. The second thread acquires the lock just to print the contents of the vector.

The `guard` variables are of type `MutexGuard<'a, Vec<i32>>` where `'a` is the lifetime of the shared mutex reference passed to `lock` (this ensures that the mutex itself will stay around for at least as long as the guard). Mutex guards serve two purposes. Most importantly, if a thread owns a guard, that means it holds the lock. To this end, guards provide a method `deref_mut` which turns a mutable reference of `MutexGuard<T>` into a mutable reference of `T`. Very much unlike `Cell`, the `Mutex` type permits obtaining interior pointers into the data guarded by the lock. In fact, the compiler will insert calls to `deref_mut` automatically where appropriate, making `MutexGuard<'a, Vec<i32>>` behave essentially like `&'a mut Vec<i32>`. That explains why we can write `guard.push(0)` in line 5.

Moreover, the guards are set up to release the lock when their destructors are called, which will happen automatically when the guards go out of scope.[20] To understand why there cannot be interior pointers into the lock content any more at that point, we have to consider the type of `deref_mut`:[21]

```
for<'a, 'b> fn(&'b mut MutexGuard<'a, T>) -> &'b mut T
```

In other words, the lifetime of the returned reference is restricted to the lifetime of the original borrow of `MutexGuard`. Or, put differently, as long as the returned reference is alive, there will also be an outstanding loan for the guard. When the guard is dropped, the compiler ensures that there are no outstanding loans and thus interior pointers obtained through `deref_mut` have all expired.

[19] This puts the `Mutex` API in control of all accesses to its contents, which it uses to ensure that access is only possible when the lock has been acquired.

[20] We already mentioned these destructors when discussing automatic resource management at the end of §8.1. This provides another example of what they can be useful for.

[21] `for<'a, 'b>` is Rust syntax for universal quantification over lifetimes `'a` and `'b`.

Chapter 9

The $\lambda_{\textsf{Rust}}$ language and type system

In this chapter, we introduce $\lambda_{\textsf{Rust}}$: our formal version of Rust. The Rust surface language comes with significant syntactic sugar (some of which we have already seen in §8). To simplify the formalization, $\lambda_{\textsf{Rust}}$ features only a small set of primitive constructs and requires the advanced sugar of Rust's surface language to be desugared into these primitives. Indeed, something very similar happens in the compiler itself, where surface Rust is lowered into the *Mid-level Intermediate Representation (MIR).*[1] $\lambda_{\textsf{Rust}}$ is much closer to MIR than to surface Rust.

Before we present the syntax (§9.1), operational semantics (§9.2), and type system (§9.3) of $\lambda_{\textsf{Rust}}$, we highlight some of its key features:

- Programs are represented in continuation-passing style. This choice enables us to represent complex control-flow constructs, like labeled `break` and early `return`, as present in the Rust surface language. Furthermore, following the correspondence of CPS and control-flow graphs,[2] this makes $\lambda_{\textsf{Rust}}$ easier to relate to MIR.
- The individual instructions of our language perform a single operation. By keeping the individual instructions simple and avoiding large composed expressions, it becomes possible to describe the type system in a concise way.
- The memory model of $\lambda_{\textsf{Rust}}$ supports pointer arithmetic and ensures that programs with data races or illegal memory accesses can reach a stuck state in the operational semantics. In particular, programs that cannot get stuck in any execution—a guarantee established by the adequacy theorem of our type system (Theorem 5)—are data-race free.

As in Part I, we are using `typewriter` font for language-level terms, sans-serif font for terms of the logic and type system, and *italic* font for domains (corresponding to types in our Coq formalization). We additionally use SMALL-CAPS font to denote the fields of a record (which will become relevant in sec:lrust:semtypes).

[1] Matsakis, "Introducing MIR", 2016 [Mat16a].

[2] Appel, "Compiling with continuations", 2007 [App07].

9.1 Syntax

The syntax of λ_{Rust} is as follows:

$$\begin{aligned}
\mathit{Path} \ni p &::= x \mid p.n \\
\mathit{Val} \ni v &::= \mathtt{false} \mid \mathtt{true} \mid z \mid \ell \mid \mathtt{funrec}\, f(\bar{x})\, \mathtt{ret}\, k := F \\
\mathit{Instr} \ni I &::= v \mid p \mid p_1 + p_2 \mid p_1 - p_2 \mid p_1 \leq p_2 \mid p_1 == p_2 \\
&\mid \mathtt{new}(n) \mid \mathtt{delete}(n, p) \mid {}^*p \mid p_1 := p_2 \mid p_1 :=_n {}^*p_2 \\
&\mid p :\overset{\mathsf{inj}\ i}{==} () \mid p_1 :\overset{\mathsf{inj}\ i}{==} p_2 \mid p_1 :\overset{\mathsf{inj}\ i}{==}_n {}^*p_2 \\
\mathit{FuncBody} \ni F &::= \mathtt{let}\, x = I\, \mathtt{in}\, F \mid \mathtt{letcont}\, k(\bar{x}) := F_1\, \mathtt{in}\, F_2 \\
&\mid \mathtt{newlft}; F \mid \mathtt{endlft}; F \\
&\mid \mathtt{if}\, p\, \mathtt{then}\, F_1\, \mathtt{else}\, F_2 \mid \mathtt{case}\, {}^*p\, \mathtt{of}\, \overline{F} \\
&\mid \mathtt{jump}\, k(\bar{x}) \mid \mathtt{call}\, f(\bar{x})\, \mathtt{ret}\, k
\end{aligned}$$

We let path offsets n and integer literals z range over the integers, and sum indices i range over the natural numbers. The language has two kinds of variables: program variables, which are written as x or f, and continuation variables, which are written as k.

We distinguish four classes of expressions: *function bodies* F consist of *instructions* I that operate on *paths* p and *values* v.

Values. Values only include the most basic kinds of data: the Booleans **false** and **true**, integers z, locations ℓ (see §9.2 for further details), and functions **funrec** $f(\bar{x})$ **ret** $k := F$.[3] There are no literals for products or sums as these only exist in memory, represented by sequences of values and tagged unions, respectively.

Paths. Paths are used to express the values that instructions operate on. The common case is to directly refer to a local variable x. Beyond this, paths can refer to parts of a compound data structure laid out in memory: Offsets $p.n$ perform pointer arithmetic, incrementing the pointer expressed by p by n memory cells.

Function bodies. Function bodies mostly serve to chain instructions together and manage control flow, which is handled through continuations. Continuations are declared using **letcont** $k(\bar{x}) := F_1$ **in** F_2, and called using **jump** $k(\bar{x})$. The parameters $\bar{x}$ are instantiated when calling the continuation.[4] We allow continuations to be recursive, in order to model looping constructs like `while` and `for`.[5]

The "ghost instructions" **newlft** and **endlft** start and end lifetimes. These instructions have interesting typing rules, which is why they cannot be handled like normal instructions, but do not do anything operationally.[6]

Functions can be declared using **funrec** $f(\bar{x})$ **ret** $k := F$, where f is a binder for the recursive call, $\bar{x}$ is a list of binders for the arguments, and k is a binder for the return continuation. The return continuation takes one argument for the return value. Functions can be called using

[3] Here, $\bar{x}$ denotes a list of variables.

[4] This makes continuations look a lot like "basic blocks with arguments", which are closely related to SSA form. However, MIR is *not* an SSA IR, and neither is λ_{Rust}. In SSA, these arguments would represent the current value of some local variable; in λ_{Rust}, they represent the *location* of some local variable, and the current value is stored at that location.

[5] Note that λ_{Rust} does not support mutually recursive continuations, which means we cannot model irreducible control flow. However, such control flow does not arise in Rust.

[6] Technically speaking, only **newlft** is special because it introduces a new bound variable in the type system (the new lifetime). For consistency, we treat **endlft** the same way.

call $f(\bar{x})$ **ret** k, where $\bar{x}$ is the list of parameters and k is the continuation that should be called when the function returns.

Local variables of λ_{Rust}—as represented by **let** bindings—are pure values. This is different from local variables in Rust (and MIR), which are mutable and addressable. Hence, to correctly model Rust's local variables, we allocate them on the heap and pass arguments in a "boxed" way, *i.e.*, as pointers to values in memory. Figure 9.1 shows how this looks in practice. Similar to prior work on low-level languages[7], we do not make a distinction between the stack and the heap.

```
fn option_as_mut<'a>
    (x: &'a mut Option<i32>) ->
    Option<&'a mut i32> {
  match *x {
    None => None,
    Some(ref mut t) => Some(t)
  }
}
```

$$\begin{array}{l}
\texttt{funrec}\ \mathit{option\_as\_mut}(x)\ \texttt{ret}\ \mathit{ret} := \\
\quad \texttt{let}\ r = \texttt{new}(2)\ \texttt{in} \\
\quad \texttt{letcont}\ k() := \texttt{delete}(1, x); \texttt{jump}\ \mathit{ret}(r)\ \texttt{in} \\
\quad \texttt{let}\ y = {}^{*}x\ \texttt{in case}\ {}^{*}y\ \texttt{of} \\
\quad -\ r \overset{\mathsf{inj}\ 0}{:=} (); \texttt{jump}\ k() \\
\quad -\ r \overset{\mathsf{inj}\ 1}{:=} y.1; \texttt{jump}\ k()
\end{array}$$

Figure 9.1: `option_as_mut` in Rust and λ_{Rust}.

We see that on the right, the function argument x is a pointer, which is dereferenced when used and deallocated before the function returns. In this case, since the Rust program takes a pointer, x actually is a pointer to a pointer. Similarly, a pointer r is allocated for the return value. Also note that the memory backing x is explicitly deallocated via **delete**, modeling the usually implicit deallocation of stack-allocated variables as they go out of scope.

Instructions. The λ_{Rust} language has instructions for the usual arithmetic operations (addition, subtraction, less-or-equal and equality), memory allocation, and deallocation, as well as loading from memory (${}^{*}p$) and storing a value into memory ($p_1 := p_2$). The `memcpy`-like instruction $p_1 :=_n {}^{*}p_2$ copies the contents of n memory locations from p_2 to p_1. All of these accesses are *non-atomic*, *i.e.*, they are not thread safe. We will come back to this point in §9.2.

The example above also demonstrates the handling of sums. Values of the `Option<i32>` type are represented by a sequence of two base values: an integer value that represents the tag (0 for `None` and 1 for `Some`) and, if the tag is 1, a value of type `i32` for the argument t of `Some`(t). If the tag is 0, the second value can be anything. The instructions $p_1 \overset{\mathsf{inj}\ i}{:=} p_2$ and $p_1 \overset{\mathsf{inj}\ i}{:=}_n {}^{*}p_2$ can be used to assign to a pointer p_1 of sum type, setting both the tag i and the value associated with this variant of the union, while $p_1 \overset{\mathsf{inj}\ i}{:=} ()$ is used for variants that have no data associated with them (like `None`). The **case** ${}^{*}p$ command is used to perform case-distinction on the tag of a sum stored in memory at location p, jumping to the n-th branch for tag n. If there is no branch corresponding to the tag, the program gets stuck. In contrast, **if** p is used for case-distinction on Boolean values.

[7] Leroy *et al.*, "The CompCert memory model, version 2", 2012 [Ler+12]; Krebbers, "The C standard formalized in Coq", 2015 [Kre15]

$$
\begin{aligned}
z &\in \mathbb{Z} \\
\mathit{Expr} \ni e &::= v \mid x \\
&\mid e_1.e_2 \mid e_1 + e_2 \mid e_1 - e_2 \\
&\mid e_1 \leq e_2 \mid e_1 == e_2 \\
&\mid e(\bar{e}) \\
&\mid {*^o}e \mid e_1 :=_o e_2 \\
&\mid \texttt{CAS}(e_0, e_1, e_2) \\
&\mid \texttt{alloc}(e) \mid \texttt{free}(e_1, e_2) \\
&\mid \texttt{case}\ e\ \texttt{of}\ \bar{e} \\
&\mid \texttt{fork}\ \{\ e\ \} \\
\mathit{Val} \ni v &::= \text{☣} \mid \ell \mid z \mid \texttt{rec}\ f(\bar{x}) := e \\
\mathit{Loc} \ni \ell &::= (i, n) \\
\mathit{Order} \ni o &::= \texttt{sc} \mid \texttt{na} \mid \texttt{na'} \\
\mathit{LockSt} \ni \pi &::= \texttt{writing} \mid \texttt{reading}\ n \\
h \in \mathit{Mem} &:= (\mathbb{N} \times \mathbb{N}) \xrightarrow{\text{fin}} \mathit{LockSt} \times \mathit{Val}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Ctx} \ni K &::= \bullet \\
&\mid K.e \mid v.K \\
&\mid K + e \mid v + K \mid K - e \mid v - K \\
&\mid K \leq e \mid v \leq K \mid K == e \mid v == K \\
&\mid K(\bar{e}) \mid v(\bar{v} \mathbin{+\!\!+} [K] \mathbin{+\!\!+} \bar{e}) \\
&\mid {*^o}K \mid K :=_o e \mid v :=_o K \\
&\mid \texttt{CAS}(K, e_1, e_2) \\
&\mid \texttt{CAS}(v_0, K, e_2) \\
&\mid \texttt{CAS}(v_0, v_1, K) \\
&\mid \texttt{alloc}(K) \\
&\mid \texttt{free}(K, e_2) \\
&\mid \texttt{free}(e_1, K) \\
&\mid \texttt{case}\ K\ \texttt{of}\ \bar{e}
\end{aligned}
$$

Figure 9.2: λ_{Rust} core language syntax.

9.2 Operational semantics

The operational semantics of λ_{Rust} is given by translation into a core language (aptly named λ_{Rust}-core), as defined in Figure 9.2. The core language is a lambda calculus equipped with primitive values, n-ary function calls, pointer arithmetic ($e_1.e_2$), and concurrency including two different "memory orders" to distinguish atomic and non-atomic accesses.[8] We define the semantics this way for three reasons. First of all, we can model some of the λ_{Rust} constructs (*e.g.*, $p_1 :=_n {*p_2}$) as sequences of simpler instructions in the core language, which makes it easier to reason about. Secondly, we can reduce both continuations and functions to plain lambda terms, using the same underlying reasoning principles for both. Finally, the core language supports a substitution-based semantics (§7.3), which makes reasoning more convenient, whereas the CPS grammar given above is not closed under substitution.

The operational semantics is formally defined in Figure 9.3. We use evaluation contexts K to define *where* in the current expression the next reduction will take place (O-ectx); the remaining rules define the possible head reductions. In the following, we discuss some key aspects of this language and how λ_{Rust} is translated down to the core language.

Pointer equality. In Rust, safe code can perform pointer equality tests, so we have to somehow model this feature in our core language as well. However, pointer equality is a tricky subject in non-garbage-collected languages: typically, you want to make sure that the program *cannot observe* whether a newly allocated object reuses some memory that was previously allocated to some other object. This ensures maximal freedom when implementing the language semantics in a compiler. We can achieve

[8] Some primitives of the core language are not even exposed in higher-level λ_{Rust}, *e.g.*, instructions to spawn threads (`fork` $\{\ e\ \}$) or perform atomic accesses ($e_1 :=_{\mathtt{sc}} e_2$), including `CAS` (compare-and-swap). Since these operations do not have typing rules, they can anyway only be used by `unsafe` code which has full access to the core language.

Equality. $\boxed{h \vdash v_1 = v_2}$

$$h \vdash z = z \qquad\qquad h \vdash \ell = \ell \qquad\qquad \frac{\ell_1 \notin \mathsf{dom}(h) \lor \ell_2 \notin \mathsf{dom}(h)}{h \vdash \ell_1 = \ell_2}$$

Inequality. $\boxed{h \vdash v_1 \neq v_2}$

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

Small-step operational semantics. $\boxed{h \mid e \to h' \mid e'_1, e'^{?}_2}$

O-ECTX
$$\frac{h \mid e \to h \mid e'_1, e'^{?}_2}{h \mid K[e] \to h \mid K[e'_1], e'^{?}_2}$$

O-PROJ
$$\frac{\ell = (i, n')}{h \mid \ell.n \to h \mid (i, n + n')}$$

O-ADD
$$\frac{z_1 + z_2 = z'}{h \mid z_1 + z_2 \to h \mid z'}$$

O-SUB
$$\frac{z_1 - z_2 = z'}{h \mid z_1 - z_2 \to h \mid z'}$$

O-LE-TRUE
$$\frac{z_1 \leq z_2}{h \mid z_1 \leq z_2 \to h \mid 1}$$

O-LE-FALSE
$$\frac{z_1 > z_2}{h \mid z_1 \leq z_2 \to h \mid 0}$$

O-EQ-TRUE
$$\frac{h \vdash v_1 = v_2}{h \mid v_1 == v_2 \to h \mid 1}$$

O-EQ-FALSE
$$\frac{h \vdash v_1 \neq v_2}{h \mid v_1 == v_2 \to h \mid 0}$$

O-ALLOC
$$\frac{n > 0 \qquad \ell = (i, n') \qquad \{i\} \times \mathbb{N} \mathrel{\#} \mathsf{dom}(h) \qquad h' = h[\ell + m \leftarrow (\texttt{reading}\,0, \text{☣}) \mid m \in [<n]]}{h \mid \texttt{alloc}(n) \to h' \mid \ell}$$

O-FREE
$$\frac{n > 0 \qquad \ell = (i, n') \qquad \mathsf{dom}(h) \cap \{i\} \times \mathbb{N} = \{i\} \times ([\geq n', < n' + n]) \qquad h' = h[\ell + m \leftarrow \bot \mid m \in [<n]]}{(h \mid \texttt{free}(n, \ell)) \to (h' \mid \text{☣})}$$

O-DEREF-SC
$$\frac{h(\ell) = (\texttt{reading}\,n, v)}{h \mid {*^{\texttt{sc}}}\ell \to h \mid v}$$

O-DEREF-NA
$$\frac{h(\ell) = (\texttt{reading}\,n, v)}{(h \mid {*^{\texttt{na}}}\ell) \to (h[\ell \leftarrow (\texttt{reading}\,n + 1, v)] \mid {*^{\texttt{na'}}}\ell)}$$

O-DEREF-NA'
$$\frac{h(\ell) = (\texttt{reading}\,n + 1, v)}{(h \mid {*^{\texttt{na'}}}\ell) \to (h[\ell \leftarrow (\texttt{reading}\,n, v)] \mid v)}$$

O-ASSIGN-SC
$$\frac{h(\ell) = (\texttt{reading}\,0, v')}{(h \mid \ell :=_{\texttt{sc}} v) \to (h[\ell \leftarrow (\texttt{reading}\,0, v)] \mid \text{☣})}$$

O-ASSIGN-NA
$$\frac{h(\ell) = (\texttt{reading}\,0, v')}{(h \mid \ell :=_{\texttt{na}} v) \to (h[\ell \leftarrow (\texttt{writing}, v')] \mid \ell :=_{\texttt{na'}} v)}$$

O-ASSIGN-NA'
$$\frac{h(\ell) = (\texttt{writing}, v')}{(h \mid \ell :=_{\texttt{na'}} v) \to (h[\ell \leftarrow (\texttt{reading}\,0, v)] \mid \text{☣})}$$

O-CAS-FAIL
$$\frac{h(\ell) = (\texttt{reading}\,n, v') \qquad h \vdash v' \neq v_1}{(h \mid \texttt{CAS}(\ell, v_1, v_2)) \to (h \mid 0)}$$

O-CAS-SUC
$$\frac{h(\ell) = (\texttt{reading}\,0, v') \qquad h \vdash v' = v_1}{(h \mid \texttt{CAS}(\ell, v_1, v_2)) \to (h[\ell \leftarrow (\texttt{reading}\,0, z_2)] \mid 1)}$$

O-CAS-STUCK
$$\frac{h(\ell) = (\texttt{reading}\,n, v') \qquad n > 0 \qquad h \vdash v' = v_1}{(h \mid \texttt{CAS}(\ell, v_1, v_2)) \to (h \mid 0())}$$

O-CASE
$$(h \mid \texttt{case}\; i \;\texttt{of}\; \bar{e}) \to (h \mid \bar{e}_i)$$

O-APP
$$(h \mid (\texttt{rec}\; f(\bar{x}) := e)(\bar{v})) \to (h \mid e[\texttt{rec}\; f(\bar{x}) := e/f, \bar{v}/\bar{x}])$$

O-FORK
$$h \mid \texttt{fork}\,\{\, e \,\} \to h \mid \text{☣}, e$$

Figure 9.3: λ_{Rust}-core operational semantics.

this, for example, by requiring that pointers must not be dangling when they are compared (*i.e.*, they must still point to allocated memory).

The issue is that safe code can compare *arbitrary* pointers, including dangling pointers. To still prevent the program from observing allocator details, we thus went with a different approach: when comparing two different locations $\ell_1 \neq \ell_2$ of which one is dangling, the result of the comparison is *non-deterministic*: it could be either **`true`** (represented as 1) or **`false`** (represented as 0). To this end, the semantics of equality is defined by O-eq-true and O-eq-false using two helper judgments, $h \vdash v_1 = v_2$ and $h \vdash v_1 \neq v_2$. These judgments define when two values can be considered equal and when they can be considered inequal depending on the current memory h—and these two cases are not mutually exclusive.

Pointer arithmetic. The memory model is inspired by CompCert[9] in terms of how it supports pointer arithmetic. As such, locations ℓ consist of a block index i and an offset n into that block. A memory h is a partial function mapping locations to values (and, as we will see in the next paragraph, a lock state).[10] Pointer arithmetic just acts on the offset (O-proj); this reflects the fact that it is impossible to perform arithmetic operations across memory blocks. Organizing memory in blocks also helps to ensure the correct deallocation of memory: **`free`** may only be used to deallocate an entire block at once (O-free).

Data races. On top of this, we want the memory model to detect and rule out data races. Following C++11,[11] we provide both *non-atomic* memory accesses, on which races are considered undefined behavior, and *atomic* accesses, which may be racy. However, for simplicity, we only provide sequentially consistent (SC) atomic operations, avoiding consideration of C++11's relaxed atomics in this dissertation.[12]

We consider a program to have a *data race* if there are ever two concurrent accesses to the same location, at least one of which is a write, and at least one of which is non-atomic. To verify the absence of data races in Iris, the operational semantics is carefully designed such that programs with a data race can reach a stuck state in at least one execution. This lets us prove the following theorem:

Theorem 3 (Data-race freedom)**.** *Given a program e with initial memory h, if that combination of program and memory is* safe *(*i.e.*, if no thread ever reaches a stuck state), then it is data-race-free.*

This theorem basically says that we can handle data races in the same way as other safety concerns such as use-after-free: we prove a Hoare triple for the code under consideration. Hoare triples imply that the program cannot get stuck, so this is sufficient to establish safety and thus data-race freedom.

To obtain Theorem 3, every memory location is equipped with some additional state $\pi \in \mathit{LockSt}$ resembling a reader-writer lock. The state **`reading`** 0 ("there are 0 readers") corresponds to the lock being unlocked. A non-atomic access consists of two separate steps: first we acquire the lock (a read lock for read accesses in O-deref-na, and a write lock for

[9] Leroy *et al.*, "The CompCert memory model, version 2", 2012 [Ler+12].

[10] We also experimented with representing the memory more like $\mathbb{N} \xrightarrow{\text{fin}} \mathbb{N} \xrightarrow{\text{fin}} \mathit{Val}$, but handling lookups in such nested partial maps turned out to be quite painful in Coq.

[11] ISO Working Group 21, "Programming languages – C++", 2011 [ISO11].

[12] Notice that atomicity is a property of the individual memory *access*, not of the memory location. The same location can be subject to both atomic and non-atomic accesses. This matches the approach of LLVM IR, and C++20 is also moving in this direction with the introduction of `atomic_ref`.

write accesses in O-ASSIGN-NA). In the executing expression, we replace the ordering parameter `na` by `na'`, which is not an actual memory ordering but merely indicates that we have acquired the lock and the next reduction step will finish the memory access. In the second step we then perform the actual access and release the lock (O-DEREF-NA' and O-ASSIGN-NA'). Crucially, these steps are separately subject to interference, so other threads could take turns between them. Moreover, if the first step cannot work because the lock is already held by another thread, the program is stuck.

Sequentially consistent (atomic) accesses, on the other hand, complete in a single step of execution and they do not modify the lock. However, they still check the lock to make sure that there is no conflicting non-atomic access going on (O-DEREF-SC, O-ASSIGN-SC). Again, in case of conflicting accesses, the program gets stuck.

Now imagine what happens when one thread does a non-atomic read while another thread does an atomic write to the same location. There exists an interleaving of these two threads where the first thread starts with O-DEREF-NA, changing the lock state of the location to `reading` 1. Then the thread gets scheduled away and the writing thread wants to take a turn with O-ASSIGN-SC—but it cannot, because that rule requires the lock state to be `reading` 0. The second thread is stuck. A similar situation arises for the other forms of data races (non-atomic read with non-atomic write, atomic read with non-atomic write, non-atomic write with atomic write, non-atomic write with non-atomic write).

Special care has to be taken for `CAS`(ℓ, v_1, v_2): this operation atomically compares the current value at ℓ with v_1, and if that comparison is successful, stores v_2 in ℓ. This means that, depending on the result of the comparison, a read or a write access happens, which affects the interaction with the lock (O-CAS-FAIL and O-CAS-SUC). Moreover, remember that our equality test is *non-deterministic*, and sometimes two values can be considered both equal and unequal. Now consider the situation where a CAS performs such a non-deterministic comparison and ℓ is currently read-locked: in case the comparison fails, the CAS only tries to acquire a read lock, which is fine. However, in case the comparison succeeds, the CAS tries to acquire a write lock, which would fail. Such a program *has* a data race (the CAS could write), but the program is *not* stuck as it can make progress via O-CAS-FAIL. To establish Theorem 3, we have to make sure that this program can still reach a stuck state. This is done by O-CAS-STUCK, which applies in exactly the cases where the comparison *could* succeed, but the lock is read-locked. In that case we reduce the program to $0()$; any stuck term would do.

Uninitialized memory. In our handling of uninitialized memory, we follow "Taming undefined behavior in LLVM":[13] upon allocation, memory holds a *poison* value ☣ that will cause the program to get stuck if it is ever used for a computation or a conditional branch. The only safe operations on ☣ are loading from and storing to memory. This ensures that in a safe (stuck-free) program, uninitialized memory cannot affect the program execution.

[13] Lee *et al.*, "Taming undefined behavior in LLVM", 2017 [Lee+17].

Basics, function bodies, values.

$$\begin{aligned}
\texttt{let}\,x = e\,\texttt{in}\,e' &:= (\texttt{rec}\,\_([x]) := e')(e) \\
e'; e &:= \texttt{let}\,\_ = e'\,\texttt{in}\,e \\
\texttt{skip} &:= ☣; ☣ \\
\texttt{newlft} &:= ☣ \\
\texttt{endlft} &:= \texttt{skip}
\end{aligned}$$

$$\begin{aligned}
\texttt{false} &:= 0 \\
\texttt{true} &:= 1 \\
\texttt{if}\,e_0\,\texttt{then}\,e_1\,\texttt{else}\,e_2 &:= \texttt{case}\,e_0\,\texttt{of}\,[e_1, e_2] \\
(\texttt{letcont}\,k(\bar{x}) := e\,\texttt{in}\,e') &:= (\texttt{let}\,k = (\texttt{rec}\,k(\bar{x}) := e)\,\texttt{in}\,e') \\
\texttt{jump}\,k(\bar{e}) &:= k(\bar{e}) \\
(\texttt{funrec}\,f(\bar{x})\,\texttt{ret}\,k := e) &:= (\texttt{rec}\,f([k] +\!\!+\, \bar{x}) := e) \\
\texttt{call}\,f(\bar{e})\,\texttt{ret}\,k &:= f([k] +\!\!+\, \bar{e})
\end{aligned}$$

Instructions.

$$\begin{aligned}
\texttt{new} := \texttt{rec}\,\mathit{new}(\mathit{size}) :=& \\
&\texttt{if}\,\mathit{size} == 0\,\texttt{then}\,(42, 1337)\,\texttt{else}\,\texttt{alloc}(\mathit{size}) \\
\texttt{delete} := \texttt{rec}\,\mathit{delete}(\mathit{size}, \mathit{ptr}) :=& \\
&\texttt{if}\,\mathit{size} == 0\,\texttt{then}\,☣\,\texttt{else}\,\texttt{free}(\mathit{size}, \mathit{ptr}) \\
\texttt{memcpy} := \texttt{rec}\,\mathit{memcpy}(\mathit{dst}, \mathit{len}, \mathit{src}) :=& \\
&\texttt{if}\,\mathit{len} \le 0\,\texttt{then}\,☣\,\texttt{else} \\
&\mathit{dst}.0 := \mathit{src}.0; \\
&\mathit{memcpy}(\mathit{dst}.1, \mathit{len} - 1, \mathit{src}.1)
\end{aligned}$$

$$\begin{aligned}
{}^{*}e &:= {}^{*\mathtt{na}}e \\
e_1 := e_2 &:= e_1 :=_{\mathtt{na}} e_2 \\
e_1 :=_n {}^{*}e_2 &:= \texttt{memcpy}(e_1, n, e_2) \\
e \mathrel{:}\overset{\mathsf{inj}\,i}{=\!=} () &:= e.0 := i \\
e_1 \mathrel{:}\overset{\mathsf{inj}\,i}{=\!=} e_2 &:= e_1.0 := i; e_1.1 := e_2 \\
e_1 \mathrel{:}\overset{\mathsf{inj}\,i}{=\!=}_n {}^{*}e_2 &:= e_1.0 := i; e_1.1 :=_n {}^{*}e_2
\end{aligned}$$

Figure 9.4: λ_{Rust} syntax defined in terms of the core language.

Translation from λ_{Rust}. Some λ_{Rust} constructs, such as basic arithmetic operations, are part of the core language and do not need any translation. The rest is treated as syntactic sugar as defined in Figure 9.4.

As usual, we define `let`-bound variables and sequencing in terms of immediately applied functions. We also define **`skip`** as an operation that takes a step (and then reduces to the useless value ☣). For technical reasons, the proof of **`endlft`** only works if there is some step of computation being performed by the program.[14]

We also define Booleans and conditionals based on integers and **`case`** in the straightforward way. Notice that O-le-true and all the other rules have been written specifically with this interpretation of integers as Booleans in mind.

The lowering to λ_{Rust}-core also explains the difference between a continuation and a function in λ_{Rust}. Continuations are just sugar for normal functions in the core language, so "jumping" to a continuation is just a function call. Proper λ_{Rust} functions, on the other hand, are modeled as λ_{Rust}-core functions whose first argument serves as return continuation: calling a function with a given return continuation boils down to passing that continuation as the first argument to the function.

Finally, we have to implement the memory operations of λ_{Rust}. Allocation and deallocation make direct use of the underlying core primitives, except that those primitives do not permit empty allocations (O-alloc,

[14] **`newlft`** does not do anything, and we could have left it out of λ_{Rust} and made lifetime creation just a "ghost step" in the type system. Instead, we decided to make it symmetric with **`endlft`**.

O-FREE), so we need a special case for size 0. In that case, **`new`** returns the “dummy” location $(42, 1337) \in \mathit{Loc}$. Any location would work, this address will never actually be accessed. We just need a value which syntactically is a location; that will later help to simplify our type system proofs.

$\lambda_{\textsf{Rust}}$ memory accesses are modeled as non-atomic accesses. To implement $e_1 :=_n {}^{*}e_2$, which copies a range of memory from wherever e_2 points to wherever e_1 points, we use a **`memcpy`** function which is implemented as a simple loop. The operations to initialize sum types write the given index to memory at offset 0 and then put the rest of the data at offset 1.

9.3 Type system: Overview

As we have seen in §8, the fundamental principle of the Rust type system is the idea that types represent ownership. As usual in ownership-based type systems, this is reflected in the *substructural* nature of the type context. This means that the context does not support the usual structural rule of duplicating assumptions.[15] But that is not all there is to the Rust type system: in order to avoid having to thread ownership and data through function calls, Rust uses references and borrowing (§8.2), granting temporary access to a data structure. Rust uses the notion of a lifetime (§8.5) to describe the duration for which access is granted. As such, it should come as no surprise that lifetimes will play a key role in the $\lambda_{\textsf{Rust}}$ type system.

Concretely, the types and contexts of $\lambda_{\textsf{Rust}}$ are as follows:

$$
\begin{array}{rlrl}
\mathit{Lft} \ni \kappa ::= \alpha \mid \textsf{static} & & \mathit{Mod} \ni \mu ::= \textsf{mut} \mid \textsf{shr} & \\
\mathit{Sort} \ni \sigma ::= \textsf{val} \mid \textsf{lft} \mid \textsf{type} & & \Gamma ::= \emptyset \mid \Gamma, X : \sigma & \\
\mathbf{E} ::= \emptyset \mid \mathbf{E}, \kappa \sqsubseteq_{\mathsf{e}} \kappa' & & \mathbf{T} ::= \emptyset \mid \mathbf{T}, p \lhd \tau \mid \mathbf{T}, p \lhd^{\dagger\kappa} \tau & \\
\mathbf{L} ::= \emptyset \mid \mathbf{L}, \kappa \sqsubseteq_{\mathsf{l}} \bar{\kappa} & & \mathbf{K} ::= \emptyset \mid \mathbf{K}, k \lhd \textsf{cont}(\mathbf{L}; \bar{x}.\, \mathbf{T}) & \\
\mathit{Type} \ni \tau ::= T \mid \textsf{bool} \mid \textsf{int} \mid \textsf{own}_n\, \tau \mid \&_{\mu}^{\kappa}\, \tau \mid \lightning_n \mid \Pi\bar{\tau} \mid \Sigma\bar{\tau} & & & \\
\quad \mid \forall\bar{\alpha}.\, \textsf{fn}(\digamma : \mathbf{E}; \bar{\tau}) \to \tau \mid \mu T.\, \tau & & &
\end{array}
$$

Selected rules of the most important auxiliary judgments are shown in Figure 9.5, and some of the rules for typing instructions and function bodies are given in Figure 9.6. The remaining rules have been moved to §9.4 to minimize clutter in this explanatory section. The sheer number of rules can be quite intimidating, so after discussing the types and contexts of the system, we will explain how to use these rules to typecheck two examples.

The types τ of $\lambda_{\textsf{Rust}}$. There are two kinds of pointer types: *owned pointers* $\textsf{own}_n\, \tau$ and *(borrowed) references* $\&_{\mu}^{\kappa}\, \tau$.

Owned pointers $\textsf{own}_n\, \tau$ are used to represent full ownership of (some part of) a heap allocation. Because we model the stack using heap allocations, owned pointers also represent Rust’s local, stack-allocated variables. As usual, τ is the type of the pointee. Furthermore, n tracks the size of the entire allocation. This can be different from the size of τ for *inner pointers* that point into a larger data structure.[16] Still, most of the time, n is the size of τ, in which case we omit the subscript.

[15] Reordering and removing assumptions from the context is still supported.

[16] Such pointers can be obtained using C-SPLIT-OWN, which provides a separate type assignment for each field of a struct. This is used to access a single field of a struct behind an owned pointer. We will come back to this rule in §9.4.

References $\&^{\kappa}_{\mu}\,\tau$ are qualified by a *modifier* μ, which is either **mut** (for mutable references, which are unique) or **shr** (for shared references), and a *lifetime* κ. References $\&^{\kappa}_{\mu}\,\tau$ are borrowed for lifetime κ and, as such, can only be used as long as the lifetime κ is alive, *i.e.*, still ongoing. Lifetimes begin and end at the **`newlft`** and **`endlft`** ghost instructions, following F-newlft and F-endlft. Furthermore, the special lifetime **static** lasts for the execution of the entire program.[17] The type system is able to abstract over lifetimes, so most of the time, we will work with lifetime variables α.

The type $\lightning_n$ describes arbitrary sequences of n base values. This type represents uninitialized memory. For example, when allocating an owned pointer (rule S-new), its type is **own** $\lightning_n$. Owned pointers permit strong updates, which means their type can change when the memory they point to is (re-)initialized. Similarly, the type changes back to **own** $\lightning_n$ when data is moved out of the owned pointer (rule Tread-own-move). Strong updates are sound because ownership of owned pointers is unique.

The types $\Pi\overline{\tau}$ and $\Sigma\overline{\tau}$ represent n-ary products and sums, respectively. In particular, this gives rise to a unit type **()** (the empty product $\Pi[]$) and the empty type **!** (the empty sum $\Sigma[]$). We use $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$ as notation for binary products ($\Pi[\tau_1, \tau_2]$) and sums ($\Sigma[\tau_1, \tau_2]$), respectively.

Function types $\forall\overline{\alpha}.\,\textsf{fn}(\digamma : \mathbf{E}; \overline{\tau}) \to \tau$ can be polymorphic over lifetimes $\overline{\alpha}$. The external lifetime context $\mathbf{E}$ can be used to demand that one lifetime parameter be included in another one. The lifetime $\digamma$ here is a binder than can be used in $\mathbf{E}$ to refer to the lifetime of this function. For example, the Rust type `for<'a> fn(&'a mut i32)` becomes $\forall\alpha.\,\textsf{fn}(\digamma : \digamma \sqsubseteq_{\mathsf{e}} \alpha; \&^{\alpha}_{\textsf{mut}}\,\textsf{int}) \to \textsf{()}$.[18] This is the type of a function that takes a mutable reference to an integer with any lifetime that includes this function call, and returns unit.[19] This lifetime inclusion is implicit in Rust; we chose to make all of these assumptions fully explicit in our model. As discussed before, both the parameters and the return value are transmitted via owned pointers; this calling convention is universally applied and hence does not show up in the function type.[20]

Finally, $\lambda_{\textsf{Rust}}$ supports *recursive types* $\mu T.\,\tau$, with the restriction (enforced by well-formedness as defined in §9.4) that T only appears in τ below a pointer type or within a function type. This matches the restrictions on recursive types in Rust. These restrictions arise because a type like $\mu T.\,\textsf{int} \times T$ would not even have a well-defined size:[21] in a typical functional language, such a type is implemented by adding a pointer indirection, but Rust does not implicitly add indirections like that. Instead, the user is required to acknowledge the indirection explicitly, by writing $\mu T.\,\textsf{int} \times \textsf{own}\,T$. This type is statically known to have size 2.

To keep the type system of $\lambda_{\textsf{Rust}}$ focused on our core objective (modeling borrowing and lifetimes), there is no support for type-polymorphic functions. Instead, we handle polymorphism on the meta-level: in our shallow embedding of the type system in Coq, we can quantify any definition and theorem over arbitrary semantic types (§10). We exploit this flexibility when verifying the safety of Rust libraries that use `unsafe` features (§13). These libraries are typically polymorphic, and by keeping

[17] This corresponds to `'static` in Rust, which plays the same role.

[18] `for<'a> fn(&'a mut i32)` is Rust notation for a type that universally quantifies over lifetime `'a`. Usually, we would just write `fn(&mut i32)`, which is a shorter way to implicitly denote the same type.

[19] As mentioned in §8.5, we use *lifetime inclusion* to model the dual of the "outlives" relation which the Rust compiler employs.

[20] Instead, it shows up in F-call and S-fn.

[21] The *size* of a type is defined in §9.4. It indicates how many memory locations it covers. Primitive types (integers, Booleans) have size 1; compound types (sums, products) are larger. This is a good enough model to make individual components of compound types separately addressable, but avoids going into the details of integer bitwidth, which are largely orthogonal.

Lifetimes. $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa_1 \sqsubseteq \kappa_2, \quad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \text{ }\mathsf{alive}}$

Lincl-static
$$\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \textbf{\textsf{static}}$$

Lincl-local
$$\frac{\kappa \sqsubseteq_{\mathsf{l}} \bar{\kappa} \in \mathbf{L} \qquad \kappa' \in \bar{\kappa}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}$$

Lincl-extern
$$\frac{\kappa \sqsubseteq_{\mathsf{e}} \kappa' \in \mathbf{E}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}$$

Lincl-refl
$$\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa$$

Lalive-static
$$\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \textbf{\textsf{static}} \text{ }\mathsf{alive}$$

Lalive-local
$$\frac{\kappa \sqsubseteq_{\mathsf{l}} \bar{\kappa} \in \mathbf{L} \qquad \forall i.\, \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \bar{\kappa}_i \text{ }\mathsf{alive}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \text{ }\mathsf{alive}}$$

Lalive-incl
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \text{ }\mathsf{alive} \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa' \text{ }\mathsf{alive}}$$

Subtyping. $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \Rightarrow \tau_2}$

T-bor-lft
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \&^{\kappa'}_{\mu}\, \tau \Rightarrow \&^{\kappa}_{\mu}\, \tau}$$

T-rec-unfold
$$\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mu T.\, \tau \Leftrightarrow \tau[\mu T.\, \tau / T]$$

T-own
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \Rightarrow \tau_2}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \textbf{\textsf{own}}_n\, \tau_1 \Rightarrow \textbf{\textsf{own}}_n\, \tau_2}$$

Type coercion. $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T}_1 \Rightarrow \mathbf{T}_2}$

C-subtype
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau \Rightarrow \tau'}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash p \triangleleft \tau \Rightarrow p \triangleleft \tau'}$$

C-copy
$$\frac{\Gamma \vdash \tau \text{ }\mathsf{copy}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash p \triangleleft \tau \Rightarrow p \triangleleft \tau, p \triangleleft \tau}$$

C-weaken
$$\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T}, \mathbf{T}' \Rightarrow \mathbf{T}$$

C-split-own
$$\frac{\bar{\tau} \neq [\,] \qquad \forall i.\, m_i = \sum_{j<i} \mathsf{size}(\bar{\tau}_j)}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash p \triangleleft \textbf{\textsf{own}}_n\, \Pi\bar{\tau} \Leftrightarrow \overline{p.m \triangleleft \textbf{\textsf{own}}_n\, \tau}}$$

C-share
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \text{ }\mathsf{alive}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash p \triangleleft \&^{\kappa}_{\textbf{\textsf{mut}}}\, \tau \Rightarrow p \triangleleft \&^{\kappa}_{\textbf{\textsf{shr}}}\, \tau}$$

C-borrow
$$\Gamma \mid \mathbf{E}; \mathbf{L} \vdash p \triangleleft \textbf{\textsf{own}}_n\, \tau \Rightarrow p \triangleleft \&^{\kappa}_{\textbf{\textsf{mut}}}\, \tau, p \triangleleft^{\dagger\kappa} \textbf{\textsf{own}}_n\, \tau$$

C-reborrow
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa' \sqsubseteq \kappa}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash p \triangleleft \&^{\kappa}_{\textbf{\textsf{mut}}}\, \tau \Rightarrow p \triangleleft \&^{\kappa'}_{\textbf{\textsf{mut}}}\, \tau, p \triangleleft^{\dagger\kappa'} \&^{\kappa}_{\textbf{\textsf{mut}}}\, \tau}$$

Reading and writing. $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimapinv^{\tau} \tau_2, \quad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_2}$

Tread-own-copy
$$\frac{\Gamma \vdash \tau \text{ }\mathsf{copy}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \textbf{\textsf{own}}_n\, \tau \multimapinv^{\tau} \textbf{\textsf{own}}_n\, \tau}$$

Tread-own-move
$$\frac{n = \mathsf{size}(\tau)}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \textbf{\textsf{own}}_m\, \tau \multimapinv^{\tau} \textbf{\textsf{own}}_m\, \lightning\, n}$$

Tread-bor
$$\frac{\Gamma \vdash \tau \text{ }\mathsf{copy} \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \text{ }\mathsf{alive}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \&^{\kappa}_{\mu}\, \tau \multimapinv^{\tau} \&^{\kappa}_{\mu}\, \tau}$$

Twrite-own
$$\frac{\mathsf{size}(\tau) = \mathsf{size}(\tau')}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \textbf{\textsf{own}}_n\, \tau' \multimap^{\tau} \textbf{\textsf{own}}_n\, \tau}$$

Twrite-bor
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \text{ }\mathsf{alive}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \&^{\kappa}_{\textbf{\textsf{mut}}}\, \tau \multimap^{\tau} \&^{\kappa}_{\textbf{\textsf{mut}}}\, \tau}$$

Type context unblocking. $\boxed{\Gamma \vdash \mathbf{T}_1 \Rightarrow^{\dagger\kappa} \mathbf{T}_2}$

Tunblock-hasty
$$\frac{\Gamma \mid \mathbf{T}_1 \Rightarrow^{\dagger\kappa} \mathbf{T}_2}{\Gamma \mid \mathbf{T}_1, p \triangleleft \tau \Rightarrow^{\dagger\kappa} \mathbf{T}_2, p \triangleleft \tau}$$

Tunblock-unblock
$$\frac{\Gamma \mid \mathbf{T}_1 \Rightarrow^{\dagger\kappa} \mathbf{T}_2}{\Gamma \mid \mathbf{T}_1, p \triangleleft^{\dagger\kappa} \tau \Rightarrow^{\dagger\kappa} \mathbf{T}_2, p \triangleleft \tau}$$

Tunblock-skip
$$\frac{\Gamma \mid \mathbf{T}_1 \Rightarrow^{\dagger\kappa} \mathbf{T}_2}{\Gamma \mid \mathbf{T}_1, p \triangleleft^{\dagger\kappa'} \tau \Rightarrow^{\dagger\kappa} \mathbf{T}_2, p \triangleleft^{\dagger\kappa'} \tau}$$

Figure 9.5: Selection of the typing rules of λ_{Rust} (auxiliary judgments).

Typing of instructions. $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{T} \vdash I \dashv x.\, \mathbf{T}_2}$

S-num

$$\Gamma \mid \mathbf{E}; \mathbf{L} \mid \emptyset \vdash z \dashv x.\, x \lhd \textsf{\textbf{int}}$$

S-nat-leq

$$\Gamma \mid \mathbf{E}; \mathbf{L} \mid p_1 \lhd \textsf{\textbf{int}}, p_2 \lhd \textsf{\textbf{int}} \vdash p_1 \leq p_2 \dashv x.\, x \lhd \textsf{\textbf{bool}}$$

S-fn

$$\frac{\begin{array}{c}\Gamma \vdash \bar{\tau}' \ \textsf{copy} \qquad \Gamma \vdash \bar{\tau}' \ \textsf{send} \\ \Gamma, \bar{\alpha}, \digamma : \textsf{\textbf{lft}}, f, \bar{x}, k : \textsf{\textbf{val}} \mid \mathbf{E}, \mathbf{E}'; \digamma \sqsubseteq_{\mathsf{l}} [] \mid k \lhd \textsf{\textbf{cont}}(\digamma \sqsubseteq_{\mathsf{l}} []; y.\, y \lhd \textsf{\textbf{own}}\, \tau); \\ \bar{p} \mathrel{\overline{\lhd}} \bar{\tau}', \bar{x} \mathrel{\overline{\lhd}} \textsf{\textbf{own}}\, \bar{\tau}, f \lhd \forall \bar{\alpha}.\, \textsf{\textbf{fn}}(\digamma : \mathbf{E}; \bar{\tau}) \to \tau \vdash F\end{array}}{\Gamma \mid \mathbf{E}'; \mathbf{L}' \mid \bar{p} \mathrel{\overline{\lhd}} \bar{\tau}' \vdash \texttt{funrec}\, f(\bar{x})\, \texttt{ret}\, k := F \dashv f.\, f \lhd \forall \bar{\alpha}.\, \textsf{\textbf{fn}}(\digamma : \mathbf{E}; \bar{\tau}) \to \tau}$$

S-path

$$\Gamma \mid \mathbf{E}; \mathbf{L} \mid p \lhd \tau \vdash p \dashv x.\, x \lhd \tau$$

S-new

$$\Gamma \mid \mathbf{E}; \mathbf{L} \mid \emptyset \vdash \texttt{new}(n) \dashv x.\, x \lhd \textsf{\textbf{own}}_n \lightning_n$$

S-delete

$$\frac{n = \textsf{size}(\tau)}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid p \lhd \textsf{\textbf{own}}_n\, \tau \vdash \texttt{delete}(n, p) \dashv \emptyset}$$

S-deref

$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimapinv^{\tau} \tau_1' \qquad \textsf{size}(\tau) = 1}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid p \lhd \tau_1 \vdash {}^*p \dashv x.\, p \lhd \tau_1', x \lhd \tau}$$

S-assgn

$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_1'}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau \vdash p_1 := p_2 \dashv p_1 \lhd \tau_1'}$$

S-sum-assgn

$$\frac{\bar{\tau}_i = \tau \qquad \Gamma \mid \tau_1 \multimap^{\Sigma \bar{\tau}} \tau_1'}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau \vdash p_1 \overset{\mathsf{inj}\ i}{:=} p_2 \dashv p_1 \lhd \tau_1'}$$

Typing of function bodies. $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F}$

F-consequence

$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T} \Rightarrow \mathbf{T}' \qquad \Gamma \mid \mathbf{E} \vdash \mathbf{K} \Rightarrow \mathbf{K}' \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}'; \mathbf{T}' \vdash F}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F}$$

F-let

$$\frac{\begin{array}{c}\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \vdash I \dashv x.\, \mathbf{T}_2 \\ \Gamma, x : \textsf{\textbf{val}} \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}_2, \mathbf{T} \vdash F\end{array}}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}_1, \mathbf{T} \vdash \texttt{let}\, x = I\, \texttt{in}\, F}$$

F-letcont

$$\frac{\begin{array}{c}\Gamma, k, \bar{x} : \textsf{\textbf{val}} \mid \mathbf{E}; \mathbf{L}' \mid \mathbf{K}, k \lhd \textsf{\textbf{cont}}(\mathbf{L}'; \bar{x}.\, \mathbf{T}'); \mathbf{T}' \vdash F' \\ \Gamma, k : \textsf{\textbf{val}} \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}, k \lhd \textsf{\textbf{cont}}(\mathbf{L}'; \bar{x}.\, \mathbf{T}'); \mathbf{T} \vdash F\end{array}}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash \texttt{letcont}\, k(\bar{x}) := F'\, \texttt{in}\, F}$$

F-jump

$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T} \Rightarrow \mathbf{T}'[\bar{y}/\bar{x}]}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid k \lhd \textsf{\textbf{cont}}(\mathbf{L}; \bar{x}.\, \mathbf{T}'); \mathbf{T} \vdash \texttt{jump}\, k(\bar{y})}$$

F-newlft

$$\frac{\Gamma, \alpha : \textsf{\textbf{lft}} \mid \mathbf{E}; \mathbf{L}, \alpha \sqsubseteq_{\mathsf{l}} \bar{\kappa} \mid \mathbf{K}; \mathbf{T} \vdash F}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash \texttt{newlft}; F}$$

F-endlft

$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}' \vdash F \qquad \mathbf{T} \Rightarrow^{\dagger\kappa} \mathbf{T}'}{\Gamma \mid \mathbf{E}; \mathbf{L}, \kappa \sqsubseteq_{\mathsf{l}} \bar{\kappa} \mid \mathbf{K}; \mathbf{T} \vdash \texttt{endlft}; F}$$

F-case-bor

$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \ \textsf{alive} \qquad \forall i.\, (\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p.1 \lhd \&^{\kappa}_{\mu}\, \tau_i \vdash F_i) \vee (\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p \lhd \&^{\kappa}_{\mu}\, \Sigma \bar{\tau} \vdash F_i)}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p \lhd \&^{\kappa}_{\mu}\, \Sigma \bar{\tau} \vdash \texttt{case}\, {}^*p\, \texttt{of}\, \overline{F}}$$

F-call

$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T} \Rightarrow \bar{x} \mathrel{\overline{\lhd}} \textsf{\textbf{own}}\, \bar{\tau}, \mathbf{T}' \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \bar{\kappa} \ \textsf{alive} \qquad \Gamma, \digamma : \textsf{\textbf{lft}} \mid \mathbf{E}, \digamma \sqsubseteq_{\mathsf{e}} \bar{\kappa}; \mathbf{L} \vdash \mathbf{E}'}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid k \lhd \textsf{\textbf{cont}}(\mathbf{L}; y.\, y \lhd \textsf{\textbf{own}}\, \tau, \mathbf{T}'); \mathbf{T}, f \lhd \textsf{\textbf{fn}}(\digamma : \mathbf{E}'; \bar{\tau}) \to \tau \vdash \texttt{call}\, f(\bar{x})\, \texttt{ret}\, k}$$

Figure 9.6: Selection of the typing rules of λ_{Rust} (instructions and function bodies).

the verification similarly polymorphic, we can prove that functions and libraries are safe to use at any instantiation of their type parameters.

Contexts. The typing judgments for function bodies F and instructions I have the shape $\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F$ and $\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \vdash I \dashv x.\, \mathbf{T}_2$. As we can see, there are five different kinds of contexts.

The first context, the *variable context* Γ, is the only binding context. It introduces all variables that are free in the judgment and keeps track of whether they are program variables ($x : \textsf{\textbf{val}}$; this also covers continuations), lifetime variables ($\alpha : \textsf{\textbf{lft}}$), or type variables[22] ($T : \textsf{\textbf{type}}$). All the remaining contexts state facts and assert ownership related to variables introduced here, but they do not introduce additional binders.

The *external lifetime context* $\mathbf{E}$ contains lifetime inclusion information ($\kappa \sqsubseteq_{\textsf{e}} \kappa'$) about lifetimes not under the current function's control (*i.e.*, the lifetimes a function is polymorphic over).

The *local lifetime context* $\mathbf{L}$ contains entries of the form $\kappa \sqsubseteq_{\textsf{l}} \bar{\kappa}$. It tracks which lifetime the current function has *full control* over—the lifetimes listed here on the left-hand side can be ended any time (F-endlft). On the right-hand side of the inclusion is a list of lifetimes that all outlive κ; ending any of them will implicitly also end κ (in the same `endlft`). When typing a function, initially only the function lifetime $\digamma \sqsubseteq_{\textsf{l}} []$ is listed here; more lifetimes can be added through F-newlft.

The *continuation context* $\mathbf{K}$ is used to type continuations: an item $k \lhd \textsf{\textbf{cont}}(\mathbf{L}; \bar{x}.\, \mathbf{T})$ says that continuation k can be called with arguments $\bar{x}$ assuming the type information in $\mathbf{T}$ and the lifetime information in $\mathbf{L}$ can be satisfied. When typechecking a function, the continuation context will typically contain at least the return continuation. As usual with CPS, since the return type is given by the return continuation, the function judgment does not have a notion of a return type itself. Instructions cannot invoke continuations, which is why their judgment does not have a continuation context.

And finally, the *typing context* $\mathbf{T}$ is in charge of describing ownership of local variables (in the type of a continuation, it expresses which ownership the continuation requires). It mostly contains type assignments $p \lhd \tau$.[23] Crucially, the typing context is substructural: type assignments can only be duplicated if the type satisfies $\Gamma \vdash \tau\ \textsf{copy}$ (C-copy), corresponding to Rust's `Copy` bound.[24] The typing judgment for instructions actually involves two typing contexts: one expressing the types *before* the execution of I, and one expressing the types *after* the execution. This is used to express the possibility of I "using up" variables or changing their type.

One curious aspect of $\mathbf{T}$ is that it assigns types to *paths* p, not variables x. Conceptually, paths denote an individual field of a compound data structure (in $\lambda_{\textsf{Rust}}$, this means paths can refer to (nested) products). In Rust, tracking of ownership and borrowing happens separately for each field of a `struct`; to model this adequately, our typing context also tracks fields separately using paths. Rules like C-split-own can be used to convert back and forth between considering a compound data structure as a whole, or in terms of its individual fields.

[22] Type variables can only occur in the definition of recursive types.

[23] We will come back to the other entries in $\mathbf{T}$, the *blocked* type assignments $p \lhd^{\dagger\kappa} \tau$, later.

[24] We formally define $\Gamma \vdash \tau\ \textsf{copy}$ in §9.4, but the upshot is that all types we have seen so far are duplicable except for owned pointers and mutable references.

9.3.1 Typechecking *option_as_mut*

To demonstrate how the contexts and typing rules of $\lambda_{\textsf{Rust}}$ are used, we will go through part of the typing derivation of the example from §9.1. Concretely, we aim to show the following:

$$\begin{aligned}&\emptyset \mid \emptyset; \emptyset \mid \emptyset; \emptyset \vdash \textit{option\_as\_mut} \dashv f.\\ &\qquad f \lhd \forall \alpha.\, \textsf{fn}(\digamma : \digamma \sqsubseteq_{\textsf{e}} \alpha; \&^{\alpha}_{\textsf{mut}}\,(() + \textsf{int})) \to (() + \&^{\alpha}_{\textsf{mut}}\,\textsf{int})\end{aligned}$$

option_as_mut is a value and values are valid instructions, so we use the instruction typing judgment here to say that, without any assumptions, the function has the equivalent of the following Rust type:[25]

```
for<'a> fn(&'a mut Option<i32>) -> Option<&'a mut i32>
```

To derive the desired judgment, we start by applying S-fn, the rule for typing values of function type. That rule is quite verbose, but all it really does is set up the contexts.[26] We have to derive a judgment for the body of *option_as_mut* in the following initial contexts:

$$\begin{aligned}\Gamma_1 &:= x : \textsf{val}, \textit{ret} : \textsf{val}, \alpha : \textsf{lft}, \digamma : \textsf{lft}\\ \mathbf{E}_1 &:= \digamma \sqsubseteq_{\textsf{e}} \alpha\\ \mathbf{L}_1 &:= \digamma \sqsubseteq_{\textsf{l}} []\\ \mathbf{K}_1 &:= \textit{ret} \lhd \textsf{cont}(\digamma \sqsubseteq_{\textsf{l}} []; r.\, r \lhd \textsf{own}\,(() + \&^{\alpha}_{\textsf{mut}}\,\textsf{int}))\\ \mathbf{T}_1 &:= x \lhd \textsf{own}\,\&^{\alpha}_{\textsf{mut}}\,(() + \textsf{int})\end{aligned}$$

The external lifetime context $\mathbf{E}$ reflects the assumption made in the function type, and the local lifetime context $\mathbf{L}$ declares our control over the function lifetime $\digamma$.[27] In principle, we could end $\digamma$ at any time, but $\mathbf{K}$ states that the return continuation *ret* requires as a precondition that $\digamma$ is still alive. If we ended that lifetime, we would not be able to return from the current function. The type of *ret* also says that it expects one argument r of our return type, `Option<&'a mut i32>`.

In the typing context, we have a single variable x (our argument), which is an owned pointer to $\&^{\alpha}_{\textsf{mut}}\,(() + \textsf{int})$, the $\lambda_{\textsf{Rust}}$ equivalent of the Rust type `&'a mut Option<i32>`. As already mentioned, the additional owned pointer indirection here models the fact that x on the Rust side is mutable and has an address in memory.

Under these initial contexts, the typing derivation is illustrated in Figure 9.7. We mostly use F-let to typecheck the function one instruction at a time.

The first instruction is **new**, so we use S-new. That extends our typing context with r being an uninitialized owned pointer:

$$x \lhd \textsf{own}\,\&^{\alpha}_{\textsf{mut}}\,(() + \textsf{int}), r \lhd \textsf{own}\,\lightning_2$$

Next, we declare a continuation (**letcont** $k() := \ldots$); it gets added to the continuation context $\mathbf{K}$ next to *ret*. The continuation k represents the merging control flow after the **case**. Following F-letcont, we have to pick $\mathbf{T}'$, the typing context at the call site of the continuation. It turns

[25] With lifetime elision, this can be written as `fn(&mut Option<i32>) -> Option<&mut i32>`.

[26] The typing rule also demands that all captured variables have to be `Copy` and `Send` (the respective judgments are formally defined in §9.4). This makes sure that functions can only make use of globally available, duplicable data. Notice that we do *not* use functions to model Rust's closures; at the level we work on (the MIR), closures have already been converted to explicit environments. The only reason we permit functions to capture anything at all is to permit one (global) function to call another.

[27] Rust does not have a direct equivalent of $\digamma$ in its surface syntax; instead it always implicitly assumes that lifetime parameters like `'a` outlive the current function.

```
funrec $option\_as\_mut(x)$ ret $ret$ :=
  $\{\mathbf{E} : \digamma \sqsubseteq_{\mathsf{e}} \alpha; \mathbf{L} : \digamma \sqsubseteq_{\mathsf{l}} [] \mid \mathbf{K} : ret \lhd \mathsf{cont}(\digamma \sqsubseteq_{\mathsf{l}} []; r.\ r \lhd \mathsf{own}\ (\mathbf{()} + \&^{\alpha}_{\mathsf{mut}}\ \mathsf{int})); \mathbf{T} : x \lhd \mathsf{own}\ \&^{\alpha}_{\mathsf{mut}}\ (\mathbf{()} + \mathsf{int})\}$
  let $r$ = new(2) in      (F-let, S-new)
  $\{\mathbf{T} : x \lhd \mathsf{own}\ \&^{\alpha}_{\mathsf{mut}}\ (\mathbf{()} + \mathsf{int}), r \lhd \mathsf{own}\ \lightning_2\}$   (we do not repeat unchanged contexts)
  letcont $k()$ := delete$(1, x)$; jump $ret(r)$ in      (F-letcont)
  $\{\mathbf{K} : ret \lhd \ldots, k \lhd \mathsf{cont}(\digamma \sqsubseteq_{\mathsf{l}} []; r \lhd \mathsf{own}\ (\mathbf{()} + \&^{\alpha}_{\mathsf{mut}}\ \mathsf{int}), x \lhd \mathsf{own}\ \lightning_1)\}$
  let $y$ = $^*x$ in      (F-let, S-deref, Tread-own-move)
  $\{\mathbf{T} : x \lhd \mathsf{own}\ \lightning_1, r \lhd \mathsf{own}\ \lightning_2, y \lhd \&^{\alpha}_{\mathsf{mut}}\ (\mathbf{()} + \mathsf{int})\}$
  case $^*y$ of      (F-case-bor)
  – $r :\overset{\mathsf{inj}\ 0}{=\!=} ()$; jump $k()$
  – $\{\mathbf{T} : x \lhd \mathsf{own}\ \lightning_1, r \lhd \mathsf{own}\ \lightning_2, y.1 \lhd \&^{\alpha}_{\mathsf{mut}}\ \mathsf{int}\}$
    $r :\overset{\mathsf{inj}\ 1}{=\!=} y.1$;      (F-let, S-sum-assgn, Twrite-own)
    $\{\mathbf{T} : x \lhd \mathsf{own}\ \lightning_1, r \lhd \mathsf{own}\ (\mathbf{()} + \&^{\alpha}_{\mathsf{mut}}\ \mathsf{int})\}$
    jump $k()$      (F-jump)
```

Figure 9.7: Example code with annotated type and continuation contexts.

out that the right choice is $r \lhd \mathsf{own}\ (\mathbf{()} + \&^{\alpha}_{\mathsf{mut}}\ \mathsf{int}), x \lhd \mathsf{own}\ \lightning_1$. We also have to pick the local lifetime context $\mathbf{L}'$; for that, $\digamma \sqsubseteq_{\mathsf{l}} []$ will do. Let us omit checking that the continuation actually has this type, and continue with the following new item in our continuation context:

$$k \lhd \mathsf{cont}(\digamma \sqsubseteq_{\mathsf{l}} []; r \lhd \mathsf{own}\ (\mathbf{()} + \&^{\alpha}_{\mathsf{mut}}\ \mathsf{int}), x \lhd \mathsf{own}\ \lightning_1)$$

Next, the code dereferences the argument ($\texttt{let}\ y = {}^*x$), which unwraps the additional owned pointer indirection that got inserted in the translation. Dereferencing is typechecked using S-deref. This rule uses a helper judgment: $\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_2$ means that we can read a value of type τ (the pointee) from a pointer of type τ_1, and doing so will change the type of the pointer to τ_2. In this case, we derive $\mathsf{own}\ \&^{\alpha}_{\mathsf{mut}}\ (\mathbf{()} + \mathsf{int}) \multimap^{\&^{\alpha}_{\mathsf{mut}}\ (\mathbf{()}+\mathsf{int})} \mathsf{own}\ \lightning_1$ from Tread-own-move. This is a destructive read: the type of the pointer changes because we *moved* the content out of the owned pointer. Effectively, x is now no longer initialized. After this instruction, our typing context becomes:

$$x \lhd \mathsf{own}\ \lightning_1, r \lhd \mathsf{own}\ \lightning_2, y \lhd \&^{\alpha}_{\mathsf{mut}}\ (\mathbf{()} + \mathsf{int})$$

Next, we have to typecheck the `case` using F-case-bor, which involves loading the tag from y.[28] Because we are dereferencing a reference (as opposed to an owned pointer) here, the type system requires us to show that the lifetime (α) is still alive. This is where the lifetime contexts $\mathbf{E}$ and $\mathbf{L}$ come in: We have to show $\mathbf{E}; \mathbf{L} \vdash \alpha\ \mathsf{alive}$.

To this end, we first make use of the *external* lifetime context $\mathbf{E}$, which contains $\digamma \sqsubseteq_{\mathsf{e}} \alpha$. We apply Lalive-incl, reducing the goal to $\mathbf{E}; \mathbf{L} \vdash \digamma\ \mathsf{alive}$: because $\digamma$ is shorter than α, it suffices to show that $\digamma$ is still alive. In the second step, we employ the presence of $\digamma \sqsubseteq_{\mathsf{l}} []$ in our *local* lifetime context $\mathbf{L}$. Using Lalive-local, this assumption says that $\digamma$ is alive as long as all its superlifetimes are alive. Because $\digamma$ has no superlifetimes, this finishes the proof that $\digamma$ is alive, and so is α.[29]

Having discharged the first premise of F-case-bor, let us now come to the second premise: showing that all the branches of the case distinction

[28] We assume that sums are always represented as a tag in the first location, followed by appropriately typed data. Rust actually exploits knowledge about the valid values of a type, like the fact that references are never null, to obtain more efficient representations of some sum types. This means that `Option<&mut T>` is actually represented as a single machine word, with null representing the `None` case and all other values representing `Some`. We make no attempt to model these layout optimizations in λ_{Rust}.

[29] One might expect that the presence of $\digamma$ in the local lifetime context alone is sufficient to show that it is alive. In such an alternative version of λ_{Rust}, Lalive-local would be simpler, but ending a lifetime would be more complicated as we would have to ensure that all its sublifetimes have been ended before. We chose our design because it maps most directly to the way we model lifetimes semantically (see §11).

are well-typed. The case distinction operates on a pointer to $\mathbf{()} + \mathsf{\textbf{int}}$, so in the branches, we can assume that $y.1$ (the data stored in the sum) is a pointer to $\mathbf{()}$ or $\mathsf{\textbf{int}}$, respectively.[30] The second case is the more interesting one, where we go on with the following typing context:

$$x \lhd \mathsf{\textbf{own}}\ \lightning_1, r \lhd \mathsf{\textbf{own}}\ \lightning_2, y.1 \lhd \&^{\alpha}_{\mathsf{\textbf{mut}}}\ \mathsf{\textbf{int}}$$

The next instruction is $r \overset{\mathsf{inj}\ 1}{:==} y.1$, which is typechecked using S-sum-assgn.[31] Again the main work of adjusting the types is offloaded to a helper judgment: $\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_2$ means that we can write a value of type τ to a pointer of type τ_1, changing the type of the pointer to τ_2. In this case, we derive $\mathsf{\textbf{own}}\ \lightning_2 \multimap^{()+\&^{\alpha}_{\mathsf{\textbf{mut}}}\ \mathsf{\textbf{int}}} \mathsf{\textbf{own}}\ (\mathbf{()} + \&^{\alpha}_{\mathsf{\textbf{mut}}}\ \mathsf{\textbf{int}})$ using Twrite-own. This is a strong update, which changes the type of r from uninitialized to the return type of our example function. Our context thus becomes:

$$x \lhd \mathsf{\textbf{own}}\ \lightning_1, r \lhd \mathsf{\textbf{own}}\ (\mathbf{()} + \&^{\alpha}_{\mathsf{\textbf{mut}}}\ \mathsf{\textbf{int}})$$

Notice that $y.1$ disappeared from the context; it was used up when we moved it into r.

Finally, we jump to the continuation k that we declared earlier. This is typechecked using F-jump, which verifies that our current typing context $\mathbf{T}$ and local lifetime context $\mathbf{L}$ match what is expected by the continuation.

9.3.2 *Typechecking borrowing*

As a second example, we will typecheck the function in Figure 9.8 which demonstrates how borrowing and ending a lifetime work.

```
fn bor_demo() -> i32 {
  let mut x = 0;
  let p = &mut x;
  *p = 42;
  x
}
```

```
funrec bor_demo() ret ret :=
  let x = new(1) in
  let v = 0 in x := v;
  newlft;
  let p = x in
  let v = 42 in p := v;
  endlft;
  jump ret(r)
```

Figure 9.8: bor_demo in Rust and λ_{Rust}.

The Rust function creates a local variable x, then creates a reference r pointing to x and writes through the reference. Finally, the value of x is returned. On the λ_{Rust} side, note again how x is a *pointer* to an integer, to model mutability of x. A naive translation would also add an extra pointer indirection to p, but since p is not mutated and does not have its address taken, we can simplify things to represent p without an extra indirection. This means that p and x are represented exactly the same way. The fact that one is a fully owned stack variable, and the other just a borrowed reference, will only be visible in the type system.

[30] Actually, the rule is more powerful than that: for each branch, we have the *choice* of either having $y.1$ in the context with a refined type, or keeping y with the original type. This is needed to model some tricky Rust programs.

[31] It may seem like F-let does not apply since here the program is built using sequencing instead of a let-binding, but remember that in Figure 9.4 we defined sequencing in terms of let-bindings.

The Rust function has type `fn() -> i32`, so we would like to derive the following λ_{Rust} typing judgment:

$$\emptyset \mid \emptyset; \emptyset \mid \emptyset; \emptyset \vdash \mathit{bor\_demo} \dashv f.\, f \triangleleft \mathbf{fn}(\digamma : \emptyset; \emptyset) \rightarrow \mathbf{int}$$

Again we start by using S-fn. The initial contexts for deriving a typing of the body look as follows:

$$\begin{aligned}
\Gamma_1 &:= \mathit{ret} : \mathbf{val}, \digamma : \mathbf{lft} \\
\mathbf{E}_1 &:= \emptyset \\
\mathbf{L}_1 &:= \digamma \sqsubseteq_{\mathsf{l}} [] \\
\mathbf{K}_1 &:= \mathit{ret} \triangleleft \mathbf{cont}(\digamma \sqsubseteq_{\mathsf{l}} []; r.\, r \triangleleft \mathbf{own\ int}) \\
\mathbf{T}_1 &:= \emptyset
\end{aligned}$$

We show the code annotated with typing contexts in Figure 9.9. As no continuations are defined in this example, the continuation context $\mathbf{K}$ stays the same throughout the derivation.

The first instruction is S-new, so we obtain ownership of some uninitialized memory ($x \triangleleft \mathbf{own}\ \lightning_1$).

The second instruction just gives 0 a name because the grammar does not permit storing values directly in locations. It is typed using S-num, adding $v \triangleleft \mathbf{int}$ to the context.

The next instruction is a simple assignment (this time, there are no sum types involved), which is typed using S-assgn. This rule makes use of the helper judgment for writing; we derive $\mathbf{own}\ \lightning_1 \multimap^{\mathbf{int}} \mathbf{own\ int}$ using Twrite-own as before, saying that we can write an $\mathbf{int}$ to a pointer (x) of type $\mathbf{own}\ \lightning_1$ and that changes the pointer's type to $\mathbf{own\ int}$. We could have duplicated v using C-copy if we wanted to keep it around, but we will not need it any more, so we are fine with that type assumption being removed from the context.

Now it gets interesting: we are starting a new lifetime using **`newlft`**, so we have to apply F-newlft. This rule lets us pick a list of "superlifetimes" of the new lifetime α, *i.e.*, a list of lifetimes that are going to outlive (last longer than) the new lifetime. In our case, we could pick the empty list, but for demonstration purposes we are going to make $\digamma$ a superlifetime. This means that $\alpha \sqsubseteq_{\mathsf{l}} [\digamma]$ is added to $\mathbf{L}$.

The next line looks like a simple re-binding of x to p, but just using S-path here would get us stuck later: this would remove x from the typing context, but we are using x again as our return value. Instead, remember that in the original Rust source program, the corresponding code is `let p = &mut x;`—we have to *borrow* x! Borrowing in λ_{Rust} is a "ghost step", it only affects the typing context but has no direct operational equivalent. The corresponding typing rule is C-borrow, and it says that we can take our local variable x of owned pointer type together with our newly created lifetime α, and replace x in the typing context by $x \triangleleft \&^{\alpha}_{\mathbf{mut}}\ \mathbf{int}$ and $x \triangleleft^{\dagger\alpha} \mathbf{own\ int}$. The two type assignments say that on the one hand, we can now consider x a reference borrowed for lifetime α (and that is the part we are going to use up in the assignment to p), but on the other hand we also remember that *when α ends*, we can get back full ownership of x. We say that this second type assignment is

```
funrec $\mathit{bor\_demo}()$ ret $\mathit{ret}$ :=
  $\{\mathbf{E} : \emptyset; \mathbf{L} : \digamma \sqsubseteq_{\mathsf{l}} [] \mid \mathbf{K} : \mathit{ret} \lhd \mathsf{cont}(\digamma \sqsubseteq_{\mathsf{l}} []; r.\, r \lhd \mathsf{own\ int}); \mathbf{T} : \emptyset\}$
  let $x$ = new(1) in     (F-let, S-new)
  $\{\mathbf{T} : x \lhd \mathsf{own}\ \lightning_1\}$  (we do not repeat unchanged contexts)
  let $v = 0$ in     (F-let, S-num)
  $\{\mathbf{T} : x \lhd \mathsf{own}\ \lightning_1, v \lhd \mathsf{int}\}$
  $x := v$;     (F-let, S-assgn, Twrite-own)
  $\{\mathbf{T} : x \lhd \mathsf{own\ int}\}$
  newlft;     (F-newlft)
  $\{\mathbf{L} : \alpha \sqsubseteq_{\mathsf{l}} [\digamma] \mid \mathbf{T} : x \lhd \mathsf{own\ int}\}$
  $\{\mathbf{T} : x \lhd \&^{\alpha}_{\mathsf{mut}}\ \mathsf{int}, x \lhd^{\dagger\alpha} \mathsf{own\ int}\}$  (F-consequence, C-borrow)
  let $p = x$ in     (F-let, S-path)
  $\{\mathbf{T} : p \lhd \&^{\alpha}_{\mathsf{mut}}\ \mathsf{int}, x \lhd^{\dagger\alpha} \mathsf{own\ int}\}$
  let $v = 42$ in     (F-let, S-num)
  $p := v$     (F-let, S-assgn, Twrite-bor)
  $\{\mathbf{T} : p \lhd \&^{\alpha}_{\mathsf{mut}}\ \mathsf{int}, x \lhd^{\dagger\alpha} \mathsf{own\ int}\}$
  endlft;     (F-endlft)
  $\{x \lhd \mathsf{own\ int}\}$
  jump $\mathit{ret}(x)$     (F-jump)
```

Figure 9.9: Second example code with annotated type contexts.

blocked by the lifetime α. These blocked assignments roughly correspond to the *loans* that the Rust type checker is tracking (see §8.5).

Now we are ready to use S-path, resulting in the following typing context:

$$p \lhd \&^{\alpha}_{\mathsf{mut}}\ \mathsf{int}, x \lhd^{\dagger\alpha} \mathsf{own\ int}$$

To recapitulate, the overall effect of **let** $p = x$ **in** on the typing context was that p became a reference with lifetime α and the type assignment of x became "blocked" for the same lifetime.

The next instruction is trivial, we are giving 42 the name v.

Then we use S-assgn again to write into the reference p, but this time we use a different rule for the writing helper judgment: we derive $\&^{\alpha}_{\mathsf{mut}}\ \mathsf{int} \multimap^{\mathsf{int}} \&^{\alpha}_{\mathsf{mut}}\ \mathsf{int}$ using Twrite-bor, which says that we can write an **int** to a poiner of type $\&^{\alpha}_{\mathsf{mut}}\ \mathsf{int}$ and that that write leaves the pointer's type unchanged. This derivation requires us to prove that α is alive, as is the case every time we use a reference. In our case, α is a local lifetime, so we can use Lalive-local, which demands that we show that all of α's superlifetimes are still alive. The goal thus becomes $\digamma$ alive, for which we use Lalive-local again, and since $\digamma$ has no superlifetimes we are done.[32]

With this, we arrive at the **endlft**, meaning we are going to use F-endlft. This rule lets us end *any* local lifetime. Most importantly, it lets us *unblock* blocked type assignments as defined by the unblocking judgment $\mathbf{T} \Rightarrow^{\dagger\kappa} \mathbf{T}'$, which replaces all $\lhd^{\dagger\kappa}$ by $\lhd$ and otherwise leaves $\mathbf{T}$ unchanged. In our case, this means we get full ownership of x back, just as one would expect when the borrow expires. (Technically, we could

[32] The check that $\digamma$ is still alive is necessary because F-endlft would let us end $\digamma$ any time, and that would also implicitly end α as it is a sublifetime of $\digamma$.

keep ownership of $\boldsymbol{p}$, but it would be useless since we can no longer show α alive. Hence we use C-weaken to simply forget about $\boldsymbol{p}$.)

The final instruction jumps to the return continuation, and since all types match up, this is trivial to typecheck with F-jump.

9.3.3 Further type system features

Now is a good time to go back to Figure 9.5 and Figure 9.6 and read all of the typing rules presented there; the following few paragraphs discuss the ones that were not needed for the examples.

External lifetime context satisfaction $\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{E}'$ is used on function calls to check the assumptions made by the callee (F-call). It uses the lifetime inclusion judgment to check that every inclusion in $\mathbf{E}'$ can be derived using the information in $\mathbf{E}$ and $\mathbf{L}$.

The $\overline{\triangleleft}$ in F-call indicates that we are requiring a list of type assignments in the context, matching a list of variables ($\bar{x}$) with an equal-length list of types ($\textsf{\textbf{own}}\ \bar{\tau}$).

Subtyping is described by $\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \Rightarrow \tau_2$. The main forms of subtyping supported in Rust are lifetime inclusion (T-bor-lft) and (un)folding recursive types (T-rec-unfold). Apart from that, we have the usual structural rules witnessing covariance and contravariance of type constructors (T-own is a representative example). On the type context level, $\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T}_1 \Rightarrow \mathbf{T}_2$ lifts subtyping (C-subtype, this is the "subsumption" rule) while also adding a few coercions that can only be applied at the top-level type. Most notably, a mutable reference can be coerced into a shared reference (C-share), an owned pointer can be borrowed (C-borrow) to create a mutable reference, and a mutable reference can be reborrowed (C-reborrow).[33]

9.4 Type system: Appendix

In Figure 9.5 and Figure 9.6, we only gave the typing rules that have been necessary to typecheck the examples, and we omitted some trivial definitions such as well-formedness or the size of a type. In this section, we give all the remaining definitions with brief explanations of anything unusual.

We begin with the well-formedness judgments, which are mostly concerned with defining the binding structure of the various type system components. The only interesting aspect here is well-formedness of recursive types, which requires the recursive occurrence to be below a pointer or function type. In the remainder of this dissertation, we implicitly work only with well-formed objects.

Well-formed paths. $\boxed{\Gamma \vdash_{\mathsf{wf}} p}$

$$\frac{x : \textsf{\textbf{val}} \in \Gamma}{\Gamma \vdash_{\mathsf{wf}} x} \qquad\qquad \frac{\Gamma \vdash_{\mathsf{wf}} p}{\Gamma \vdash_{\mathsf{wf}} p.n}$$

[33] There is no need to reborrow shared references because they are duplicable, and hence we can keep the original pointer while also using subtyping to obtain a shorter lifetime in a copy.

Well-formed lifetimes. $\boxed{\Gamma \vdash_{\mathsf{wf}} \kappa}$

$$\frac{\alpha : \textsf{\textbf{lft}} \in \Gamma}{\Gamma \vdash_{\mathsf{wf}} \alpha} \qquad\qquad \Gamma \vdash_{\mathsf{wf}} \textsf{\textbf{static}}$$

Well-formed external lifetime contexts. $\boxed{\Gamma \vdash_{\mathsf{wf}} \mathbf{E}}$

$$\Gamma \vdash_{\mathsf{wf}} \emptyset \qquad\qquad \frac{\Gamma \vdash_{\mathsf{wf}} \mathbf{L} \qquad \Gamma \vdash_{\mathsf{wf}} \kappa \qquad \Gamma \vdash_{\mathsf{wf}} \kappa'}{\Gamma \vdash_{\mathsf{wf}} \mathbf{L}, \kappa \sqsubseteq_{\mathsf{e}} \kappa'}$$

Well-formed local lifetime contexts. $\boxed{\Gamma \vdash_{\mathsf{wf}} \mathbf{L}}$

$$\Gamma \vdash_{\mathsf{wf}} \emptyset \qquad\qquad \frac{\Gamma \vdash_{\mathsf{wf}} \mathbf{L} \qquad \Gamma \vdash_{\mathsf{wf}} \kappa \qquad \forall \kappa' \in \bar{\kappa}.\, \Gamma \vdash_{\mathsf{wf}} \kappa'}{\Gamma \vdash_{\mathsf{wf}} \mathbf{L}, \kappa \sqsubseteq_{\mathsf{l}} \bar{\kappa}}$$

Well-formed types. $\boxed{\Gamma \vdash_{\mathsf{wf}} \tau}$

$$\frac{T : \textsf{\textbf{type}} \in \Gamma}{\Gamma \vdash_{\mathsf{wf}} T} \qquad \Gamma \vdash_{\mathsf{wf}} \textsf{\textbf{bool}} \qquad \Gamma \vdash_{\mathsf{wf}} \textsf{\textbf{int}} \qquad \Gamma \vdash_{\mathsf{wf}} \text{↯}_n \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \tau}{\Gamma \vdash_{\mathsf{wf}} \textsf{\textbf{own}}_n\, \tau} \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \kappa \qquad \Gamma \vdash_{\mathsf{wf}} \tau}{\Gamma \vdash_{\mathsf{wf}} \&^{\kappa}_{\mu}\, \tau}$$

$$\frac{\forall i.\, \Gamma \vdash_{\mathsf{wf}} \bar{\tau}_i}{\Gamma \vdash_{\mathsf{wf}} \Pi\bar{\tau}} \qquad \frac{\forall i.\, \Gamma \vdash_{\mathsf{wf}} \bar{\tau}_i}{\Gamma \vdash_{\mathsf{wf}} \Sigma\bar{\tau}} \qquad \frac{\Gamma, \bar{\alpha}, \digamma : \textsf{\textbf{lft}} \vdash_{\mathsf{wf}} \mathbf{E} \qquad \forall i.\, \Gamma, \bar{\alpha} : \textsf{\textbf{lft}} \vdash_{\mathsf{wf}} \bar{\tau}_i \qquad \Gamma, \bar{\alpha} : \textsf{\textbf{lft}} \vdash_{\mathsf{wf}} \tau}{\Gamma \vdash_{\mathsf{wf}} \forall \bar{\alpha}.\, \textsf{\textbf{fn}}(\digamma : \mathbf{E}; \bar{\tau}) \to \tau}$$

$$\frac{\Gamma, T : \textsf{\textbf{type}} \vdash_{\mathsf{wf}} \tau \qquad T \text{ only occurs below pointer or function types in } \tau}{\Gamma \vdash_{\mathsf{wf}} \mu T.\, \tau}$$

Well-formed type contexts. $\boxed{\Gamma \vdash_{\mathsf{wf}} \mathbf{T}}$

$$\Gamma \vdash_{\mathsf{wf}} \emptyset \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \mathbf{T} \qquad \Gamma \vdash_{\mathsf{wf}} p \qquad \Gamma \vdash_{\mathsf{wf}} \tau}{\Gamma \vdash_{\mathsf{wf}} \mathbf{T}, p \lhd \tau} \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \mathbf{T} \qquad \Gamma \vdash_{\mathsf{wf}} p \qquad \Gamma \vdash_{\mathsf{wf}} \kappa \qquad \Gamma \vdash_{\mathsf{wf}} \tau}{\Gamma \vdash_{\mathsf{wf}} \mathbf{T}, p \lhd^{\dagger\kappa} \tau}$$

As we have already mentioned in the previous section, all types have a *size* indicating how many memory locations are taken up by instances of this type. $\mathsf{size}(\tau)$ is a function computing that size for the given type τ.

Size. $\boxed{\mathsf{size}(\tau)}$

$$\begin{aligned}
\mathsf{size}(\textsf{\textbf{bool}}) &:= 1 & \mathsf{size}(\textsf{\textbf{own}}_n\, \tau) &:= 1 \\
\mathsf{size}(\textsf{\textbf{int}}) &:= 1 & \mathsf{size}(\&^{\kappa}_{\mu}\, \tau) &:= 1 \\
\mathsf{size}(\text{↯}_n) &:= n & \mathsf{size}(\Pi\bar{\tau}) &:= \sum_i \mathsf{size}(\bar{\tau}_i) \\
& & \mathsf{size}(\Sigma\bar{\tau}) &:= 1 + \max_i \mathsf{size}(\bar{\tau}_i) \\
\mathsf{size}(\mu T.\, \tau) &:= \mathsf{size}(\tau) & \mathsf{size}(\forall \bar{\alpha}.\, \textsf{\textbf{fn}}(\digamma : \mathbf{E}; \bar{\tau}) \to \tau) &:= 1
\end{aligned}$$

We need some simple judgments which correspond to the Rust traits `Copy`, `Send` and `Sync`.

$\Gamma \vdash \tau\ \mathsf{copy}$ reflects `Copy`, which indicates that a type can be freely duplicated. All base types except for owned pointers and mutable references are `Copy`, and structural types are `Copy` whenever all their components are. (In particular, recursive type variables can be assumed to be `Copy`.)

$\Gamma \vdash \tau\ \mathsf{send}$ and $\Gamma \vdash \tau\ \mathsf{sync}$, reflecting `Send` and `Sync`, describe types that can be safely sent to another thread or shared across thread boundaries. In the syntactic type system we have seen so far, all types are both `Send` and `Sync`, but for some of the unsafely implemented types we are going to consider in §13 that will not be the case. We will come back to these properties in §12.4.

Copy types. $\boxed{\Gamma \vdash \tau\ \mathsf{copy}}$

$$\Gamma \vdash \mathbf{bool}\ \mathsf{copy} \qquad \Gamma \vdash \mathbf{int}\ \mathsf{copy} \qquad \Gamma \vdash \text{↯}_n\ \mathsf{copy} \qquad \Gamma \vdash \&^{\kappa}_{\mathsf{shr}}\,\tau\ \mathsf{copy} \qquad \frac{\forall i.\ \Gamma \vdash \tau_i\ \mathsf{copy}}{\Gamma \vdash \Pi\bar{\tau}\ \mathsf{copy}} \qquad \frac{\forall i.\ \Gamma \vdash \tau_i\ \mathsf{copy}}{\Gamma \vdash \Sigma\bar{\tau}\ \mathsf{copy}}$$

$$\Gamma \vdash (\forall\bar{\alpha}.\ \mathbf{fn}(\mathsf{F} : \mathbf{E}; \bar{\tau}) \to \tau)\ \mathsf{copy} \qquad \Gamma \vdash T\ \mathsf{copy} \qquad \frac{\Gamma, T : \mathbf{type} \vdash \tau\ \mathsf{copy}}{\Gamma \vdash \mu T.\, \tau\ \mathsf{copy}}$$

Send types $\boxed{\Gamma \vdash \tau\ \mathsf{send}}$

$$\Gamma \vdash \mathbf{bool}\ \mathsf{send} \qquad \Gamma \vdash \mathbf{int}\ \mathsf{send} \qquad \Gamma \vdash \text{↯}_n\ \mathsf{send} \qquad \frac{\Gamma \vdash \tau\ \mathsf{send}}{\Gamma \vdash \mathbf{own}_n\,\tau\ \mathsf{send}} \qquad \frac{\tau\ \mathsf{send}}{\&^{\kappa}_{\mathsf{mut}}\,\tau\ \mathsf{send}} \qquad \frac{\Gamma \vdash \tau\ \mathsf{sync}}{\Gamma \vdash \&^{\kappa}_{\mathsf{shr}}\,\tau\ \mathsf{send}}$$

$$\frac{\forall i.\ \Gamma \vdash \tau_i\ \mathsf{send}}{\Gamma \vdash \Pi\bar{\tau}\ \mathsf{send}} \qquad \frac{\forall i.\ \Gamma \vdash \tau_i\ \mathsf{send}}{\Gamma \vdash \Sigma\bar{\tau}\ \mathsf{send}} \qquad \Gamma \vdash (\forall\bar{\alpha}.\ \mathbf{fn}(\mathsf{F} : \mathbf{E}; \bar{\tau}) \to \tau)\ \mathsf{send} \qquad \Gamma \vdash T\ \mathsf{send} \qquad \frac{\Gamma, T : \mathbf{type} \vdash \tau\ \mathsf{send}}{\Gamma \vdash \mu T.\, \tau\ \mathsf{send}}$$

Sync types $\boxed{\Gamma \vdash \tau\ \mathsf{sync}}$

$$\Gamma \vdash \mathbf{bool}\ \mathsf{sync} \qquad \Gamma \vdash \mathbf{int}\ \mathsf{sync} \qquad \Gamma \vdash \text{↯}_n\ \mathsf{sync} \qquad \frac{\Gamma \vdash \tau\ \mathsf{sync}}{\Gamma \vdash \mathbf{own}_n\,\tau\ \mathsf{sync}} \qquad \frac{\Gamma \vdash \tau\ \mathsf{sync}}{\Gamma \vdash \&^{\kappa}_{\mathsf{mut}}\,\tau\ \mathsf{sync}} \qquad \frac{\Gamma \vdash \tau\ \mathsf{sync}}{\Gamma \vdash \&^{\kappa}_{\mathsf{shr}}\,\tau\ \mathsf{sync}}$$

$$\frac{\forall i.\ \Gamma \vdash \tau_i\ \mathsf{sync}}{\Gamma \vdash \Pi\bar{\tau}\ \mathsf{sync}} \qquad \frac{\forall i.\ \Gamma \vdash \tau_i\ \mathsf{sync}}{\Gamma \vdash \Sigma\bar{\tau}\ \mathsf{sync}} \qquad \Gamma \vdash (\forall\bar{\alpha}.\ \mathbf{fn}(\mathsf{F} : \mathbf{E}; \bar{\tau}) \to \tau)\ \mathsf{sync} \qquad \Gamma \vdash T\ \mathsf{sync} \qquad \frac{\Gamma, T : \mathbf{type} \vdash \tau\ \mathsf{sync}}{\Gamma \vdash \mu T.\, \tau\ \mathsf{sync}}$$

For space reasons, we omitted the transitivity rule for lifetime inclusion and the rule for unblocking an empty context in Figure 9.5; they are given below. We also formally define the rather unremarkable external lifetime context satisfiability judgment.

Lifetime inclusion. $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa_1 \sqsubseteq \kappa_2}$

Lincl-trans

$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa' \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa' \sqsubseteq \kappa''}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa''}$$

External lifetime context satisfiability. $\boxed{\Gamma \mid \mathbf{E}_1; \mathbf{L}_1 \vdash \mathbf{E}_2}$

Esat-empty

$$\Gamma \mid \mathbf{E}_1; \mathbf{L}_1 \vdash \emptyset$$

Esat-incl

$$\frac{\Gamma \mid \mathbf{E}_1; \mathbf{L}_1 \vdash \kappa \sqsubseteq \kappa' \qquad \Gamma \mid \mathbf{E}_1; \mathbf{L}_1 \vdash \mathbf{E}_2}{\Gamma \mid \mathbf{E}_1; \mathbf{L}_1 \vdash \mathbf{E}_2, \kappa \sqsubseteq_{\mathsf{e}} \kappa'}$$

Type context unblocking. $\boxed{\Gamma \vdash \mathbf{T}_1 \Rrightarrow^{\dagger\kappa} \mathbf{T}_2}$

Tunblock-empty

$$\Gamma \mid \emptyset \Rrightarrow^{\dagger\kappa} \emptyset$$

The remaining subtyping rules are mostly structural: shared references, products and sums are covariant (just like owned pointers, as already defined before); mutable references are invariant; and functions as usual are covariant in their return type and contravariant in their argument type. Recursive types are covariant, but the rule for this makes use of meta-level implication, which usually has no place in a syntactic type system—the rule resembles what we have proven semantically, but properly expressing that syntactically would require introducing a context of "subtyping assumptions". We also have reflexivity and transitivity of subtyping.

Note that unlike typical unique pointers, mutable references have to be invariant because they are borrowed: when the lifetime ends, the lender will expect data of the original type τ, not some supertype τ'. The difference to unique pointers arises because mutable references are only *unique for their lifetime.*

And finally, T-uninit-prod declares the equivalence of a single large chunk of uninitialized memory with a product type where each component is uninitialized. (Note that the sum sign here is a sum of integers, not a sum type.) This rule is used when typechecking the initialization of a product: one large allocation is made, T-uninit-prod is used to give that allocation product type, C-split-own (from Figure 9.5) is used to view each field of the product as an individual path, and then they can all be initialized with the usual strong update via S-assgn and Twrite-own.

An equivalence ($\Leftrightarrow$) in a conclusion is sugar for two rules witnessing both directions. An equivalence in an assumption is sugar for requiring both directions to hold for the rule to apply.

Subtyping. $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \Rightarrow \tau_2}$

T-refl

$$\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau \Rightarrow \tau$$

T-trans

$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau \Rightarrow \tau' \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau' \Rightarrow \tau''}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau \Rightarrow \tau''}$$

T-uninit-prod

$$\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \lightning_{\Sigma\bar{n}} \Leftrightarrow \Pi\overline{\lightning_{n}}$$

T-rec

$$\frac{\forall \tau_1', \tau_2'.\, (\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1' \Rightarrow \tau_2') \Rightarrow (\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1[\tau_1'/T_1] \Rightarrow \tau_2[\tau_2'/T_2])}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mu T_1.\, \tau_1 \Rightarrow \mu T_2.\, \tau_2}$$

T-bor-shr

$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \Rightarrow \tau_2}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \&^{\kappa}_{\mathsf{shr}}\, \tau_1 \Rightarrow \&^{\kappa}_{\mathsf{shr}}\, \tau_2}$$

$$\textsc{T-bor-mut}\quad \frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \Leftrightarrow \tau_2}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \&^{\kappa}_{\textsf{mut}}\, \tau_1 \Leftrightarrow \&^{\kappa}_{\textsf{mut}}\, \tau_2}$$

$$\textsc{T-prod}\quad \frac{\forall i.\, \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \bar{\tau}_i \Rightarrow \overline{\tau'}_i}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \Pi\bar{\tau} \Rightarrow \Pi\overline{\tau'}}$$

$$\textsc{T-sum}\quad \frac{\forall i.\, \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \bar{\tau}_i \Rightarrow \overline{\tau'}_i}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \Sigma\bar{\tau} \Rightarrow \Sigma\overline{\tau'}}$$

$$\textsc{T-fn}\quad \frac{\begin{array}{c}\Gamma, \bar{\alpha}', \digamma : \textsf{lft} \mid \mathbf{E}', \mathbf{E}_0; \mathbf{L}_0 \vdash \mathbf{E}[\bar{\kappa}/\bar{\alpha}] \\ \forall i.\, \Gamma, \bar{\alpha}', \digamma : \textsf{lft} \mid \mathbf{E}', \mathbf{E}_0; \mathbf{L}_0 \vdash \bar{\tau}'_i \Rightarrow \bar{\tau}_i \qquad \Gamma, \bar{\alpha}', \digamma : \textsf{lft} \mid \mathbf{E}', \mathbf{E}_0; \mathbf{L}_0 \vdash \tau \Rightarrow \tau'\end{array}}{\Gamma \mid \mathbf{E}_0; \mathbf{L}_0 \vdash \forall\bar{\alpha}.\, \textsf{fn}(\digamma : \mathbf{E}; \bar{\tau}) \to \tau \Rightarrow \forall\bar{\alpha}'.\, \textsf{fn}(\digamma : \mathbf{E}'; \bar{\tau}') \to \tau'}$$

The only interesting type coercion rule we did not discuss yet is C-split-bor, which is the equivalent of C-split-own for references. (Again the sum sign here refers to a sum of integers.) The rule requires the list of types $\bar{\tau}$ in the product to be non-empty, because otherwise the right-hand side of this bidirectional type coercion would be empty—and we cannot conjure references (or owned pointers) to empty products out of nothing, they still have to be syntactically a pointer *at least*. The other side-condition handles the offsets: m_i is the offset of the i-th field, so it has to equal the sum of the sizes of all fields before it.

Continuations do not have any interesting coercions. The judgment expresses that the context can be reordered, weakened, and that the assumptions the coercion is making about the type context behave contravariantly.

Type coercion. $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T}_1 \Rightarrow \mathbf{T}_2}$

$$\textsc{C-perm}\quad \frac{\mathbf{T}' \text{ is a permutation of } \mathbf{T}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T} \Rightarrow \mathbf{T}'}$$

$$\textsc{C-frame}\quad \frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T}_1 \Rightarrow \mathbf{T}_2}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T}', \mathbf{T}_1 \Rightarrow \mathbf{T}', \mathbf{T}_2}$$

$$\textsc{C-split-bor}\quad \frac{\bar{\tau} \neq [\,] \qquad \forall i.\, m_i = \sum_{j<i} \textsf{size}(\bar{\tau}_j)}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash p \lhd \&^{\kappa}_{\mu}\, \Pi\bar{\tau} \Leftrightarrow \overline{p.m \lhd \&^{\kappa}_{\mu}\, \tau}}$$

Continuation coercion. $\boxed{\Gamma \mid \mathbf{E} \vdash \mathbf{K}_1 \Rightarrow \mathbf{K}_2}$

$$\frac{\mathbf{K}' \text{ is a permutation of } \mathbf{K}}{\Gamma \mid \mathbf{E} \vdash \mathbf{K} \Rightarrow \mathbf{K}'} \qquad \Gamma \mid \mathbf{E} \vdash \mathbf{K}, \mathbf{K}' \Rightarrow \mathbf{K} \qquad \frac{\Gamma \mid \mathbf{E} \vdash \mathbf{K} \Rightarrow \mathbf{K}' \qquad \Gamma, \bar{x} : \textsf{val} \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T}' \Rightarrow \mathbf{T}}{\Gamma \mid \mathbf{E} \vdash \mathbf{K}, k \lhd \textsf{cont}(\mathbf{L}; \bar{x}.\mathbf{T}) \Rightarrow \mathbf{K}', k \lhd \textsf{cont}(\mathbf{L}; \bar{x}.\mathbf{T}')}$$

Most of the remaining typing rules for instructions do not really introduce anything new. We can type Boolean values and integer arithmetic, and we have various variants of assignment that all make use of the helper judgment for writing which expresses how the types change when memory gets written to. The *memcpy* rules (S-memcpy and S-sum-memcpy) perform both read and write accesses, so they also make use of the reading helper judgment that we have seen before with S-deref.

The only interesting rules we have not discussed yet are S-deref-bor-own and S-deref-bor-mut. The point of these rules is to overcome a limitation of the dereferencing rule S-deref that we used in our examples: it creates a copy of the data behind the pointer, so this rule only lets us dereference pointers to `Copy` types. That makes the rule insufficient to typecheck Rust programs that perform (re)borrowing of nested pointers: given a reference `x: &'a &'b mut T` (*i.e.*, a shared reference to a mutable reference), we can obtain a *shared* reference to the inner `T` via dereferencing (`*x: &'a T`). This is not a plain copy of the inner reference, because its type changes: the new reference is shared, and it will have the *outer* of the two lifetimes (`'a`) which Rust enforces to always be shorter than the inner lifetime. In other words, a mutable reference "below" a shared reference basically becomes itself shared, and we can get shared access to all data it points to, but subject to the shorter outer lifetime `'a`. If the outer reference is mutable, then we can even get *mutable* access to the inner data (the `T`), but still at the shorter outer lifetime.[34] And if the inner reference is instead an owned pointer, then it does not even have a lifetime to worry about. S-deref-bor-mut and S-deref-bor-own are responsible for typing the dereferencing of such nested pointers.

The remaining cases of nested pointers can be typechecked as well: if the outer pointer is owned (for example, given a pointer of type $\textsf{\textbf{own}}\ \textsf{\textbf{own}}\ \tau$), we can use C-borrow to turn it into a reference ($\&^{\kappa}_{\textsf{\textbf{mut}}}\ \textsf{\textbf{own}}\ \tau$), and then S-deref-bor-own or S-deref-bor-mut applies. If the inner pointer is a shared reference, we can use S-deref, and in that case we actually obtain a proper copy of the inner reference at its full lifetime.

Well-typed instructions. $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \vdash I \dashv x.\, \mathbf{T}_2}$

S-true

$$\Gamma \mid \mathbf{E}; \mathbf{L} \mid \emptyset \vdash \texttt{true} \dashv x.\, x \lhd \textsf{\textbf{bool}}$$

S-false

$$\Gamma \mid \mathbf{E}; \mathbf{L} \mid \emptyset \vdash \texttt{false} \dashv x.\, x \lhd \textsf{\textbf{bool}}$$

S-nat-op

$$\Gamma \mid \mathbf{E}; \mathbf{L} \mid p_1 \lhd \textsf{\textbf{int}}, p_2 \lhd \textsf{\textbf{int}} \vdash p_1 \{+,-\}\, p_2 \dashv x.\, x \lhd \textsf{\textbf{int}}$$

S-deref-bor-own

$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa\ \textsf{alive}}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid p \lhd \&^{\kappa}_{\mu}\ \textsf{\textbf{own}}_n\ \tau \vdash {}^{*}p \dashv x.\, x \lhd \&^{\kappa}_{\mu}\ \tau}$$

S-deref-bor-mut

$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa\ \textsf{alive} \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid p \lhd \&^{\kappa}_{\mu}\ \&^{\kappa'}_{\textsf{\textbf{mut}}}\ \tau \vdash {}^{*}p \dashv x.\, x \lhd \&^{\kappa}_{\mu}\ \tau}$$

S-sum-assgn-unit

$$\frac{\bar{\tau}_i = \Pi[\,] \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\Sigma\bar{\tau}} \tau_1'}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid p \lhd \tau_1 \vdash p \overset{\textsf{inj}\ i}{:==} () \dashv p \lhd \tau_1'}$$

S-memcpy

$$\frac{\textsf{size}(\tau) = n \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_1' \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_2 \multimapinv^{\tau} \tau_2'}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau_2 \vdash p_1 :=_n {}^{*}p_2 \dashv p_1 \lhd \tau_1', p_2 \lhd \tau_2'}$$

S-sum-memcpy

$$\frac{\textsf{size}(\tau) = n \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\Sigma\bar{\tau}} \tau_1' \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_2 \multimapinv^{\tau} \tau_2' \qquad \bar{\tau}_i = \tau}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau_2 \vdash p_1 \overset{\textsf{inj}\ i}{:==}_n {}^{*}p_2 \dashv p_1 \lhd \tau_1', p_2 \lhd \tau_2'}$$

Finally, in terms of function bodies, F-if is rather unremarkable and F-case-own is very similar to F-case-bor. F-equalize, on the other hand, is *interesting*: it says that whenever we have a local lifetime α that is a sublifetime of *exactly one* other lifetime κ, then we may "equalize" (or "equate") the two lifetimes: we can basically declare that α will last as long as possible, which—since it is a sublifetime of κ—means it lasts exactly as long as κ. Thus we obtain mutual lifetime inclusion between α and κ, and we also lose α from the local lifetime context as we are no longer in control of when this lifetime ends. This obscure rule is needed to typecheck the "problem case #3" from an early blog post on non-lexical lifetimes.[35] A variant of non-lexical lifetimes has been implemented in

[34] After all, we only exclusively borrowed that inner reference for the short lifetime, so after the lifetime is over, other parties may have access to it again. We thus cannot claim exclusive access to `T` for any longer than the outer lifetime.

[35] Matsakis, "Non-lexical lifetimes: Introduction", 2016 [Mat16b].

Rust since the publication of that blog post, but problem case #3 still gets rejected by current compilers. However, work is underway on a third-generation borrow checker[36] dubbed "Polonius",[37] and supporting such code is one key design goal of Polonius.

Well-typed functions. $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F}$

F-equalize

$$\frac{\Gamma \mid \mathbf{E}, \alpha \sqsubseteq_{\mathsf{e}} \kappa, \kappa \sqsubseteq_{\mathsf{e}} \alpha; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F}{\Gamma \mid \mathbf{E}; \mathbf{L}, \alpha \sqsubseteq_{\mathsf{l}} [\kappa] \mid \mathbf{K}; \mathbf{T} \vdash F}$$

F-if

$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F_1 \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F_2}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p \lhd \textsf{\textbf{bool}} \vdash \texttt{if}\ p\ \texttt{then}\ F_1\ \texttt{else}\ F_2}$$

F-case-own

$$\frac{\begin{array}{c}\forall i.\ (\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p.0 \lhd \textsf{\textbf{own}}_n\ \lightning, p.1 \lhd \textsf{\textbf{own}}_n\ \bar\tau_i, p.(1 + \mathsf{size}(\bar\tau_i)) \lhd \textsf{\textbf{own}}_n\ \lightning_{(\max_j \mathsf{size}(\bar\tau_j)) - \mathsf{size}(\bar\tau_i)} \vdash F_i) \lor \\ (\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p \lhd \textsf{\textbf{own}}_n\ \Sigma\bar\tau \vdash F_i)\end{array}}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p \lhd \textsf{\textbf{own}}_n\ \Sigma\bar\tau \vdash \texttt{case}\ {}^*p\ \texttt{of}\ \overline{F}}$$

[36] The first two generations were the naive scope-based checker that came with Rust 1.0, and NLL (non-lexical lifetimes).

[37] Matsakis, "An alias-based formulation of the borrow checker", 2018 [Mat18].

Chapter 10

A semantic model of $\lambda_{\textsf{Rust}}$ types in Iris

Our proof of soundness of the $\lambda_{\textsf{Rust}}$ type system proceeds by defining a *logical relation*, which interprets the types and typing judgments of $\lambda_{\textsf{Rust}}$ as logical predicates in an appropriate semantic domain. We focus here on the interpretation of types, leaving the interpretation of typing judgments and the statements of our main soundness theorem to §12. First, we give a simplified version of the semantic domain of types (§10.1). To reason about $\lambda_{\textsf{Rust}}$ programs, we need an appropriate program logic; its specificities are introduced in §10.2. Based on that, we give the semantic interpretation of some representative $\lambda_{\textsf{Rust}}$ types (§10.3). Finally, in §10.4, we focus on the interpretation of shared reference types. It turns out that to account for them, we have to generalize the semantic domain of types.

10.1 A simplified semantic domain of types

The semantic domain of types answers the question "What *is* a type?". Usually, the answer is that *a type denotes a set of values*—or, equivalently, a predicate over values. Fundamentally, this is also the case for $\lambda_{\textsf{Rust}}$, but the details are somewhat more complicated.

First of all, our model of the type system of $\lambda_{\textsf{Rust}}$ expresses types not as predicates in "plain mathematics" (*e.g.*, the usual higher-order logic), but as predicates in *Iris*. Using Iris to express types has the advantage that concepts like ownership are already built into the underlying framework, so the model itself does not have to take care of them. We assume familiarity with the features of Iris covered in Part I. Extra elements we need will be introduced *en passant*, as needed.

Another source of complexity in our semantic model is that types are not mere predicates over *values*: our interpretation of types associates with every type τ an Iris predicate $[\![\tau]\!].\textsc{own} \in \mathit{TId} \times \mathit{List}(\mathit{Val}) \to \mathit{iProp}$. This Iris-level predicate (as indicated by the *iProp*) takes two parameters: the second parameter is the *list* of values we are considering. As discussed in §9, types describe data that is laid out in memory and spans multiple locations. However, we have to impose some restrictions on the lists of values accepted by a type: we require that every type has a fixed *size* $[\![\tau]\!].\textsc{size}$. This size is used to compute the layout of compound data structures, *e.g.*, for product types. We require that a type only accepts lists whose length matches the size:

$$[\![\tau]\!].\textsc{own}(t, \bar{v}) \twoheadrightarrow |\bar{v}| = [\![\tau]\!].\textsc{size} \qquad (\textsc{ty-size})$$

Furthermore, for `Copy` types, we require that $[\![\tau]\!].\textsc{own}(t, \bar{v})$ be *persistent.*[1]

The first parameter of the predicate (of type *TId*) permits types to moreover depend on the *thread identifier* of the thread that claims ownership. This is used for types like `&Cell` that cannot be sent to another thread. In other words, *ownership is (in general) thread-relative.* It turns out that this enables a very natural way of modeling `Send`: semantically speaking, a type τ is `Send` if $[\![\tau]\!].\textsc{own}$ does *not* depend on the thread id. This will be defined formally in §12.4, and in §13.1 we will see how `Cell` makes crucial use of the thread identifier.

To summarize, the *preliminary* form of a semantic type $[\![\tau]\!] \in \mathit{SemType}$ is described by the following record:

$$\mathit{PreSemType} := \left\{ \begin{array}{l} \textsc{size} : \mathbb{N}, \\ \textsc{own} : \mathit{TId} \times \mathit{List}(\mathit{Val}) \to \mathit{iProp} \end{array} \right\}$$

$$\mathit{SemType} := \{ T \in \mathit{PreSemType} \mid\, \vdash \forall t, \bar{v}.\, T.\textsc{own}(t, \bar{v}) \mathrel{-\!\!*} |\bar{v}| = T.\textsc{size} \}$$

10.2 Program logic

In order to use Iris for our semantic model of types, we need to instantiate Iris for λ_{Rust}. All of the proof rules in Figure 5.6 (page 75) can be used without any issue, but λ_{Rust} has slightly different reasoning principles than HeapLang. The proof rules for arithmetic and other pure operations in λ_{Rust} are straightforward. However, λ_{Rust} has recursive functions,[2] for which we need a suitable reasoning principle. Moreover, the memory of λ_{Rust} is somewhat different from the one in HeapLang, and thus also needs dedicated proof rules.

Recursive functions. The proof rule hoare-rec for reasoning about recursive functions is shown at the top of Figure 10.1. This rule says that to verify $\texttt{rec}\ f(\bar{x}) := e$ applied to some arguments, it is sufficient to verify the body of the function (with suitable substitutions applied) *under the assumption* that all recursive calls (which will, after substitution, be calls to $\texttt{rec}\ f(\bar{x}) := e$) are already verified. Note that this recursive assumption carries its own universal quantification over $\bar{v}$, which is crucial to permit recursive calls to use different arguments than the original call. The precondition P can depend on $\bar{v}$, and thus serves as a loop invariant.[3] The entire assumption is wrapped in the persistence modality $\square$ to ensure that the result of this verification can be applied as often as necessary.

hoare-rec can be derived using *Löb induction*,[4] using the O-app reduction step to strip off the later that arises as part of that induction.

Memory operations. The two key assertions that we will use to reason about memory (taking the place of the standard *points-to* connective) are $\ell \overset{q}{\mapsto} \bar{v}$ and $\mathsf{Dealloc}(\ell, n, m)$, both of which are timeless.[5] Their rules are defined in Figure 10.1.

$\ell \overset{q}{\mapsto} \bar{v}$ is a variant of the fractional points-to assertion that works with *lists* of values. It states that, starting at location ℓ, the next $|\bar{v}|$ many locations in memory contain the values given by $\bar{v}$, and asserts *ownership* of fraction q of this memory region. We will use $\ell \overset{q}{\mapsto} v$ as sugar for a

[1] Remember that an Iris proposition is considered persistent if it does not describe ownership of any exclusive right or resource, and can therefore be freely copied and shared among several parties (§3.3).

[2] Full HeapLang, as formalized in Coq, also supports recursive functions, but the simplified version we used in Part I does not.

[3] A weakest precondition version of this rule would still require such a loop invariant, making hoare-rec one of the few proof rules that does not actually become simpler when using weakest preconditions instead of Hoare triples.

[4] Jung *et al.*, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b], §5.6.

[5] As explained in §3.2, timeless propositions can have a $\triangleright$ removed in front of them with vs-timeless.

Proof rule for recursive functions.

HOARE-REC

$$\frac{\Box\Big(\big(\forall \bar{v}.\, \{P\}\, (\mathtt{rec}\, f(\bar{x}) := e)(\bar{v})\, \{w.\, Q\}\big) \mathrel{-\!\!*} \forall \bar{v}.\, \{P\}\, e[\mathtt{rec}\, f(\bar{x}) := e/f, \bar{v}/\bar{x}]\, \{w.\, Q\}\Big)}{\forall \bar{v}.\, \{P\}\, (\mathtt{rec}\, f(\bar{x}) := e)(\bar{v})\, \{w.\, Q\}}$$

Primitive proof rules for memory operations.

LRUST-HEAP-TIMELESS

$$\mathsf{timeless}(\ell \xmapsto{q} \bar{v})$$

LRUST-DEALLOC-TIMELESS

$$\mathsf{timeless}(\mathsf{Dealloc}(\ell, n, m))$$

LRUST-HEAP-NIL

$$\mathsf{True} \mathrel{-\!\!*} \ell \mapsto \epsilon$$

LRUST-HEAP-ADD

$$\frac{|\bar{v}| = |\bar{v}'|}{\ell \xmapsto{q} \bar{v} * \ell \xmapsto{q'} \bar{v}' \mathrel{*\!\!*} \ell \xmapsto{q+q'} \bar{v} * \bar{v} = \bar{v}'}$$

LRUST-HEAP-APP

$$\ell \xmapsto{q} \bar{v} * \ell.|\bar{v}| \xmapsto{q} \bar{v}' \mathrel{*\!\!*} \ell \xmapsto{q} \bar{v} +\!\!+ \bar{v}'$$

LRUST-DEALLOC-NULL

$$\mathsf{True} \mathrel{-\!\!*} \mathsf{Dealloc}(\ell, 0, 0)$$

LRUST-DEALLOC-SPLIT

$$\mathsf{Dealloc}(\ell, n, m) * \mathsf{Dealloc}(\ell.n, n', m) \mathrel{*\!\!*} \mathsf{Dealloc}(\ell, n + n', m)$$

LRUST-ALLOC

$$\frac{n > 0}{\{\mathsf{True}\}\, \mathtt{alloc}(n)\, \{\ell.\, \exists \bar{v}.\, \ell \mapsto \bar{v} * |\bar{v}| = n * \mathsf{Dealloc}(\ell, n, n)\}}$$

LRUST-FREE

$$\frac{|\bar{v}| > 0}{\{\ell \mapsto \bar{v} * \mathsf{Dealloc}(\ell, |\bar{v}|, |\bar{v}|)\}\, \mathtt{free}(|\bar{v}|, \ell)\, \{\mathsf{True}\}}$$

LRUST-DEREF-AC

$$\{\ell \xmapsto{q} v\}\, {}^{*\mathtt{sc}}\ell\, \{v'.\, v' = v * \ell \xmapsto{q} v\}$$

LRUST-DEREF-NA

$$\{\ell \xmapsto{q} v\}\, {}^{*\mathtt{na}}\ell\, \{v'.\, v' = v * \ell \xmapsto{q} v\}$$

LRUST-ASSIGN-SC

$$\{\ell \mapsto v\}\, \ell :=_{\mathtt{sc}} w\, \{v'.\, v' = () * \ell \mapsto w\}$$

LRUST-ASSIGN-NA

$$\{\ell \mapsto v\}\, \ell :=_{\mathtt{na}} w\, \{v'.\, v' = () * \ell \mapsto w\}$$

LRUST-CAS-INT

$$\{\ell \mapsto n\}\, \mathtt{CAS}(\ell, m_1, m_2)\, \{b.\, (b = \mathtt{true} * n = m_1 * \ell \mapsto m_2) \lor (b = \mathtt{false} * n \neq m_1 * \ell \mapsto n)\}$$

Derived proof rules for memory operations.

LRUST-NEW

$$\{\mathsf{True}\}\, \mathtt{new}(n)\, \{\ell.\, \exists \bar{v}.\, \ell \mapsto \bar{v} * |\bar{v}| = n * \mathsf{Dealloc}(\ell, n, n)\}$$

LRUST-DELETE

$$\{\ell \mapsto \bar{v} * \mathsf{Dealloc}(\ell, |\bar{v}|, |\bar{v}|)\}\, \mathtt{delete}(|\bar{v}|, \ell)\, \{\mathsf{True}\}$$

LRUST-MEMCPY

$$\frac{|\bar{v}_s| = n \qquad |\bar{v}_d| = n}{\{\ell_d \mapsto |\bar{v}_d| * \ell_s \xmapsto{q} |\bar{v}_s|\}\, \mathtt{memcpy}(\ell_d, n, \ell_s)\, \{\ell_d \mapsto |\bar{v}_s| * \ell_s \xmapsto{q} |\bar{v}_s|\}}$$

Figure 10.1: Program logic proof rules for λ_{Rust}.

singleton list, and omit q when it is 1. This points-to assertion can be split along two dimensions: we can reduce the fraction of ownership of the entire region (lrust-heap-add),[6] and we can split the block into two pieces that we each own at the original fraction (lrust-heap-app)[7]. Ownership of the empty list at some location is trivial (lrust-heap-nil).

Owning some memory asserts that all the owned locations are currently *unlocked*, *i.e.*, their lock state is **reading** 0. This lets us give entirely standard rules for both the atomic and non-atomic load and store operations (lrust-deref-na, lrust-assign-sc *et cetera*). It may be surprising that we, in fact, have *the same* rules for atomic (-sc) and non-atomic (-na) accesses. In terms of reasoning, the only difference between the atomic and the non-atomic operation is that the atomic ones are, well, *physically atomic*, which lets us use rules like wp-fup-atomic and hoare-inv. We can open invariants around atomic accesses, but not around non-atomic accesses. This reflects that the difference between the two kinds of accesses is not in how they behave when we have exclusive local ownership of this memory; the difference is in how they behave when ownership is *shared* and multiple threads are accessing the same location governed by some protocol that is enforced by an invariant, and it is the invariant that owns the memory. With non-atomic accesses, such coordination is simply not possible as we cannot access locations that are owned by an invariant!

We show only the CAS rule for integers (note that lrust-cas-int is restricted to integers n, m and does not work with arbitrary values v). CAS is also possible for locations, but due to the non-deterministic nature of location equality and how it depends on which locations are still allocated in the heap (see Figure 9.3 on page 109), the rule gets more complicated. For the examples we are considering in this dissertation, integer CAS is sufficient.[8]

The assertion $\mathsf{Dealloc}(\ell, n, m)$ is only relevant for allocation and deallocation of memory. It says that we own part of the right to deallocate a block; namely, we own the right to deallocate the part of the block that starts at ℓ and has size n. The entire size of the block is recorded in m. lrust-dealloc-split can be used to take apart and assemble this permission (similar to lrust-heap-app). Allocation initially provides the permission for the entire block (lrust-alloc), and deallocation demands that $n = m$ to make sure that we are indeed deallocating an entire block at once, and not just some part of it (lrust-free).

From these rules, we can easily derive rules for our derived operations **new** (lrust-new), **delete** (lrust-delete), and **memcpy** (lrust-memcpy). The former have almost the same specification as **alloc** and **free**, respectively, except that the side-condition $n > 0$ is gone. For **memcpy**, we require ownership of two regions of memory of equal size, but for the source region (that is only read), owning any fraction q is sufficient. In the post-condition, both regions contain the values $\bar{v}_s$ that were originally in the source region.

[6] Here, $\ast\!\ast$ is sugar for a two-way magic wand:

$$P \ast\!\ast\, Q := (P \mathrel{-\!\!*} Q) \wedge (Q \mathrel{-\!\!*} P)$$

This matches our Coq formalization: we use implication only when necessary, and magic wand otherwise (similar to how we use conjunction only when necessary and separating conjunction otherwise). So far, it turns out that implication is never necessary.

[7] Notice how the second points-to on the left-hand side is offset by exactly the size of the first.

[8] The CAS rules for locations can be found in the Coq development in `theories/lang/lifting.v`.

10.3 Interpreting types

Now that we have a semantic domain of types and a program logic for λ_{Rust}, we can define the *semantic interpretation* as a function from syntactic types τ into the semantic domain.[9] In this chapter, we focus on the most representative types. The full interpretation can be found in §12.2.

Booleans. To get started, let us consider a very simple type: **bool**. It should not come as a surprise that $[\![\textsf{bool}]\!].\textsc{size} := 1$. The semantic predicate of a Boolean is defined as follows:

$$[\![\textsf{bool}]\!].\textsc{own}(t, \bar{v}) := \bar{v} = [\texttt{true}] \vee \bar{v} = [\texttt{false}] \qquad (\textsc{bool-own})$$

In other words, a Boolean can only be a singleton list (which is already expressed by its size), and that list has to contain either **`true`** or **`false`**.

Unsurprisingly, the semantic interpretation of integers is similar and equally straightforward.

Products. Given two types τ_1 and τ_2, we define the semantics of their binary product $\tau_1 \times \tau_2$ as that of the two types laid out one after the other in memory. This definition can be iterated to yield the interpretation of n-ary products.

For the size, we have $[\![\tau_1 \times \tau_2]\!].\textsc{size} := [\![\tau_1]\!].\textsc{size} + [\![\tau_2]\!].\textsc{size}$. The semantic predicate associated with $\tau_1 \times \tau_2$ uses *separating conjunction* ($P * Q$) to join the semantic predicates of both types. The separating conjunction ensures that they describe ownership of disjoint resources that can be used independently of each other. (Here, $+\!\!+$ is list concatenation.)

$$\begin{aligned}[\![\tau_1 \times \tau_2]\!].\textsc{own}(t, \bar{v}) := \exists \bar{v}_1, \bar{v}_2.\ \bar{v} = \bar{v}_1 +\!\!+ \bar{v}_2 * \\ [\![\tau_1]\!].\textsc{own}(t, \bar{v}_1) * [\![\tau_2]\!].\textsc{own}(t, \bar{v}_2) \\ (\textsc{pair-own})\end{aligned}$$

Owned pointers. In order to give a semantic interpretation to the type $\textsf{own}_n\ \tau$ of owned pointers, we use the points-to predicate introduced in §10.2.[10] With this ingredient, the interpretation is given by:

$$\begin{aligned}[\![\textsf{own}_n\ \tau]\!].\textsc{size} &:= 1 \\ [\![\textsf{own}_n\ \tau]\!].\textsc{own}(t, \bar{v}) &:= \exists \ell.\ \bar{v} = [\ell] * \triangleright \exists \overline{w}.\ \ell \mapsto \overline{w} * \\ &\qquad [\![\tau]\!].\textsc{own}(t, \overline{w}) * \mathsf{Dealloc}(\ell, [\![\tau]\!].\textsc{size}, n) \\ &\qquad (\textsc{own-ptr-own})\end{aligned}$$

In other words, an owned pointer to τ is a single location ℓ which points to some data $\overline{w}$ which has type τ.[11] So far, so simple.

What is less simple is that we wrap this entire definition in a *later* modality ($\triangleright$, see §5.1). The reason we are doing this has to do with recursive types: remember that in λ_{Rust}, a recursive type is well-formed if the recursive occurrence is below a pointer indirection (or function type). This indicates that there is something special that the pointer types (and function type) are doing that makes recursion "work". The special thing they do is that they only use the type variable τ *below a later modality*. This lets us make use of a mechanism Iris provides to define predicates

[9] With a more conventional semantic domain such as *Val* → *Prop*, this function would have a type like *Type* → (*Val* → *Prop*). In other words, the semantic interpretation can often be equivalently viewed as a relation between types and values—hence the term "logical relation". With our more complex domain though, the term makes less sense. That is why we will just speak of a semantic interpretation here.

[10] Remember that the n in $\textsf{own}_n\ \tau$ is the size of the allocated block, which may be larger than the size of τ.

[11] This is where we exploit the fact that **`new`**, the wrapper around **`alloc`** that also works for size 0, always returns a location. If it returned an integer instead (like 0), we would have to account for that case in own-ptr-own.

by *guarded recursion*: guarded recursion means that a predicate can refer to itself recursively, but only below a $\triangleright$ modality. The use of a guard ensures that the circular definition can be solved—regardless of whether the recursive reference occurs positively, negatively, or both—using the technique of "step-indexing".[12] By making sure that all pointer types (and the function type) "guard" their type arguments, we can use guarded recursion to define recursive types in λ_{Rust}.

Of course, this comes with a price: in the type system proof, we now have to get rid of this $\triangleright$ in order to actually use the ownership behind the pointer indirection. But that is fine, because actually dereferencing the pointer performs a physical step of computation, which is exactly what we need to strip a $\triangleright$.

The other oddity is the proposition $\mathsf{Dealloc}(\ell, [\![\tau]\!].\textsc{size}, n)$ in the semantic predicate. As explained in §10.2, $\mathsf{Dealloc}$ expresses the right to deallocate some block of memory: if $n = [\![\tau]\!].\textsc{size}$, we can satisfy the preconditions of LRUST-FREE, which is exactly what we will need for the proof of S-DELETE. But in general, n can be bigger than $[\![\tau]\!].\textsc{size}$, which indicates that we own only a part of the block of memory that this pointer points to.

Mutable references. Mutable references, like owned pointers, are unique pointers to something of type τ. The key difference is that mutable references are *borrowed*, not owned, and hence they come with a lifetime indicating when they expire. In standard separation logic, an assertion always represents ownership of some part of the heap, for an unlimited duration (or until the owner actively decides to give it to another party). Instead, a mutable reference in Rust represents ownership *for a limited period of time*. When this lifetime of the reference is over, a mutable reference becomes useless, because the original owner gets back the full ownership.

To handle this new notion of "ownership with an expiry date", we developed a custom logic for reasoning about lifetimes and borrowing. It is called the *lifetime logic*. This logic is embedded and proven correct in Iris, and we describe it in §11. Most importantly, for an Iris assertion P and a lifetime κ, the lifetime logic defines an assertion $\&^{\kappa}_{\mathsf{full}} P$, called a *full borrow*, representing ownership of P for the duration of lifetime κ. Using full borrows, the interpretation of the type of mutable references is as follows:

$$\begin{aligned} [\![\&^{\kappa}_{\mathsf{mut}}\,\tau]\!].\textsc{size} &:= 1 \\ [\![\&^{\kappa}_{\mathsf{mut}}\,\tau]\!].\textsc{own}(t, \bar{v}) &:= \exists \ell.\, \bar{v} = [\ell] * \&^{[\![\kappa]\!]}_{\mathsf{full}} \big(\exists \bar{w}.\, \ell \mapsto \bar{w} * [\![\tau]\!].\textsc{own}(t, \bar{w})\big) \end{aligned} \qquad \text{(MUT-REF-OWN)}$$

This is very similar to the interpretation of $\mathbf{own}_n\,\tau$, except that the assertion describing ownership of the contents of the reference ($\exists \bar{w}.\, \ell \mapsto \bar{w} * [\![\tau]\!].\textsc{own}(t, \bar{w})$) is wrapped in a full borrow (at lifetime κ) instead of a later modality $\triangleright$.[13] The $\&^{[\![\kappa]\!]}_{\mathsf{full}}$ already functions as a guard for the purpose of recursively defined predicates, so there is no need for us to add any extra $\triangleright$.

[12] Appel and McAllester, "An indexed model of recursive types for foundational proof-carrying code", 2001 [AM01].

[13] Also, there is no Dealloc because mutable references cannot be used to deallocate memory.

10.4 Interpreting shared references

The interpretation of shared references $\&^{\kappa}_{\textsf{shr}}\,\tau$ requires more work than the types we considered so far. Usually, we would proceed as we did above: Define $[\![\&^{\kappa}_{\textsf{shr}}\,\tau]\!].\textsc{own}$ based on $[\![\tau]\!].\textsc{own}$ such that all the typing rules for $\&^{\kappa}_{\textsf{shr}}\,\tau$ work out. Most of the time, this does not leave much room for choice; the primitive operations available for the type almost define it uniquely. This is decidedly not the case for shared references, for it turns out that, in Rust, there are hardly any primitive operations on `&T`. The only properties that hold for `&T` *in general* is that it can be created from a `&mut T` (C-share), it is `Copy`, it has size 1, and its values have to be memory locations. Moreover, if `T` is itself `Copy`, it becomes possible to copy data out of an `&T` (Tread-bor).

A possible model that would explain all of this behavior is to say that a shared reference points to borrowed *read-only* memory storing some list of values $\overline{w}$, and also $\overline{w}$ must satisfy $[\![\tau]\!].\textsc{own}$.[14] This naive model matches the intuition that Rust does not permit mutation of shared data. However, as we have seen in §8.6, types like `Cell` or `Mutex` *do* provide some mutating operations on shared references. If we adapt the read-only interpretation of shared references, we will later be unable to verify that these operations are safely encapsulated in unsafe code—under this model, any mutation of shared references is violating fundamental type system assumptions.

To be able to define a generally appropriate interpretation of shared references, we permit every type to pick its own *sharing predicate*. We then use the sharing predicate of τ to define $[\![\&^{\kappa}_{\textsf{shr}}\,\tau]\!].\textsc{own}$. This permits, for every type, a different set of operations on its shared references. For example, the sharing predicate for basic types like **bool** allows read-only access for anyone, while the sharing predicate for `Mutex<T>` allows read and write accesses to the underlying object of type `T` once the lock has been acquired.

More formally, we extend the semantic domain of types and associate with each of them another predicate $[\![\tau]\!].\textsc{shr} \in \mathit{Lft} \times \mathit{TId} \times \mathit{Loc} \to \mathit{iProp}$, which we use to model shared references:

$$[\![\&^{\kappa}_{\textsf{shr}}\,\tau]\!].\textsc{size} := 1$$

$$[\![\&^{\kappa}_{\textsf{shr}}\,\tau]\!].\textsc{own}(t,\bar{v}) := \exists \ell.\ \bar{v} = [\ell] * [\![\tau]\!].\textsc{shr}([\![\kappa]\!], t, \ell) \qquad (\textsc{shr-ref-own})$$

The $[\![\tau]\!].\textsc{shr}$ predicate takes three parameters: the lifetime κ of the shared reference, the thread identifier t, and the location ℓ constituting the shared reference itself. To support the aforementioned primitive operations on `&T`, the sharing predicate has to satisfy the following properties:

$$\mathsf{persistent}([\![\tau]\!].\textsc{shr}(\kappa, t, \ell)) \qquad (\textsc{ty-shr-persist})$$

$$\&^{\kappa}_{\textsf{full}}\big(\exists \overline{w}.\ \ell \mapsto \overline{w} * [\![\tau]\!].\textsc{own}(t, \overline{w})\big) * [\kappa]_q \Rrightarrow\!\!\ast_{\mathcal{N}_{\textsf{lft}}} [\![\tau]\!].\textsc{shr}(\kappa, t, \ell) * [\kappa]_q \qquad (\textsc{ty-share})$$

$$\kappa' \sqsubseteq \kappa * [\![\tau]\!].\textsc{shr}(\kappa, t, \ell) \mathrel{-\!\!*} [\![\tau]\!].\textsc{shr}(\kappa', t, \ell) \qquad (\textsc{ty-shr-mono})$$

[14] As we will see later in this section, this is the model we use for "simple types".

First, TY-SHR-PERSIST requires that $[\![\tau]\!].\textsc{shr}$ be persistent, which implies that $[\![\&^{\kappa}_{\textsf{shr}}\,\tau]\!].\textsc{own}(t, \bar{v})$ is persistent. This corresponds to the fact that, in Rust, shared references are always `Copy`.

Second, TY-SHARE asserts that shared references can be created from mutable references: This is the main ingredient for proving the rule C-SHARE of the type system. Looking at this rule more closely, its first premise is a full borrow of an owned pointer to τ. This is exactly $[\![\&^{\kappa}_{\textsf{mut}}\,\tau]\!].\textsc{own}(t, [\ell])$. Its second premise is a *lifetime token* $[\kappa]_q$, which, as we will explain in §11, witnesses that the lifetime is alive and permits accessing borrows. Given these premises, TY-SHARE states that we can perform a view shift, denoted by the Iris connective $\Rrightarrow\!\!\ast$.[15] This view shift will safely transform the resources described by the premises into those described by the conclusion, namely τ's sharing predicate along with the same lifetime token that was passed in. The mask $\mathcal{N}_{\textsf{lft}}$ (see §5) grants access to the lifetime logic.

Third, TY-SHR-MONO requires that $[\![\tau]\!].\textsc{shr}$ be monotone with respect to the lifetime parameter. This is important for proving the subtyping rule T-BOR-LFT.

The addition of the sharing predicate completes the semantic domain of types, which now looks as follows:

$$\begin{aligned} \mathit{PreSemType} &:= \left\{ \begin{array}{r} \textsc{size} : \mathbb{N}, \\ \textsc{own} : \mathit{TId} \times \mathit{List}(\mathit{Val}) \to \mathit{iProp}, \\ \textsc{shr} : \mathit{Lft} \times \mathit{TId} \times \mathit{Loc} \to \mathit{iProp} \end{array} \right\} \\ \mathit{SemType} &:= \left\{ T \in \mathit{PreSemType} \;\middle|\; \begin{array}{l} \text{TY-SIZE, TY-SHR-PERSIST,} \\ \text{TY-SHARE, TY-SHR-MONO} \end{array} \right\} \end{aligned}$$

Let us now go back to the types we already considered above and define their sharing predicates.

Sharing predicate for products. The sharing predicate for products is simply the separating conjunction of the sharing predicates of the two components:

$$[\![\tau_1 \times \tau_2]\!].\textsc{shr}(\kappa, t, \ell) := [\![\tau_1]\!].\textsc{shr}(\kappa, t, \ell) * [\![\tau_2]\!].\textsc{shr}(\kappa, t, \ell + [\![\tau_1]\!].\textsc{size})$$

The location used for the second component is shifted by $[\![\tau_1]\!].\textsc{size}$, reflecting the memory layout.

Sharing predicate for simple types. While the read-only definition of shared references is not suitable *in general*, there are still many types for which this is the right notion of sharing—types like integers and Booleans. We thus introduce a notion of *simple types* that can be used as a sharing predicate for any `Copy` type τ of size 1. All of these types have an ownership predicate that can be written as follows:

$$[\![\tau]\!].\textsc{own}(t, \bar{v}) = \exists v.\, \bar{v} = [v] * \Phi_\tau(t, v)$$

where Φ_τ is a persistent predicate specific to the individual type. For example, $\Phi_{\textsf{bool}}(t, v) := v = \texttt{true} \lor v = \texttt{false}$. With this, we can define

[15] The connective $P \Rrightarrow\!\!\ast\; Q$ is just a shorthand for $P \mathrel{-\!\!*} {\Rrightarrow} Q$ in Iris (§5.5). We use the term *view shift* both for $\Rrightarrow$ and $\Rrightarrow\!\!\ast$, but will not actually use the former connective in this part of the dissertation, so there should be no ambiguity.

the sharing predicate uniformly as follows:

$$[\![\tau]\!].\textsc{shr}(\kappa, t, \ell) := \exists v.\ \&^{\kappa}_{\mathsf{frac}}(\lambda q.\ \ell \overset{q}{\mapsto} v) * \triangleright \Phi_\tau(t, v)$$

This definition says that there exists a fixed value v (the current value the reference points to, which cannot change) such that Φ_τ holds under the later modality $\triangleright$ (recall that shared references are pointers, and hence occurrences of τ need to be guarded to enable construction of recursive types), and that we have a *fractured borrow* $\&^{\kappa}_{\mathsf{frac}}(\lambda q.\ \ell \overset{q}{\mapsto} v)$ of the ownership of the memory cell.

Fractured borrows are another notion provided by the lifetime logic: similar to full borrows, they represent temporary ownership of some resource, limited by a given lifetime. The difference is that they are persistent, but only grant *some fraction* of the content. Fortunately, that is all that is needed in order to support a read of the shared reference.

The case of the missing $\triangleright$*.* It might come as a surprise that the ownership predicate shr-ref-own of $\&^{\kappa}_{\mathsf{shr}}\,\tau$ does not use a $\triangleright$ to guard the use of the sharing predicate of τ. Given what we explained above about how all pointer types must be guarded to define the semantic interpretation of recursive types, how can this be? The full answer is complicated, but here is the high-level summary: it is okay for the *ownership* predicate of a pointer-to-τ to use the *sharing* predicate of τ in an unguarded way, because we are making sure that no sharing predicate of *any* type constructor[16] (pointer or not) uses the ownership predicate of its type parameters in an unguarded way. This makes sure that there is never an unguarded "cycle" where ownership of $\&^{\kappa}_{\mathsf{shr}}\,\tau$ is defined in terms of sharing of τ, which itself is (recursively) defined in terms of ownership of $\&^{\kappa}_{\mathsf{shr}}\,\tau$—that last step would always add a guard.

A useful intuition here is to think of this as mutual recursion. When justifying the termination of mutual recursion with some notion of size, we can "sort" the functions involved in the recursion with some well-formed order such that "bigger" functions are permitted to call *strictly* "smaller" functions with arguments that are *of the same size*—they just must not become bigger. This is okay because we know that the "smaller" function cannot recursively call the "bigger" one again without making the size strictly smaller. As a consequence, the size decreases on every possible cycle. If we carry this analogy to our domain of semantic types, then the ownership predicate is "bigger" than the sharing predicate, so we permit the ownership predicate to be defined in terms of the sharing predicate, but not vice versa. This ensures (as we have proven in Coq) that shared references do properly guard τ, and recursive types are well-defined.

[16] Type constructors are type formers that take a type as parameter, like $\&^{\kappa}_{\mathsf{mut}}\,\_$ or $\_ \times \_$.

Chapter 11

Lifetime logic

In §10, we gave a semantic model for λ_{Rust} types, but we left some important notions undefined. In particular, we used the notion of a *full borrow* $\&^{\kappa}_{\mathsf{full}} P$ in the interpretation of mutable references to reflect that this kind of ownership is temporary and will “expire” when lifetime κ ends; we mentioned *lifetime tokens* $[\kappa]_q$ as a resource used to witness that a lifetime is ongoing; and we employed *fractured borrows* $\&^{\kappa}_{\mathsf{frac}} \Phi$ in the sharing predicate of simple types.

In this chapter, we describe the *lifetime logic*, a library we have developed in Iris to support these notions. We start by presenting the two core notions of *lifetimes* and *full borrows* in §11.1. We then continue in §11.2, explaining how lifetimes can be *compared* and *intersected*. In §11.3 and §11.4, we present *fractured borrows* and *atomic borrows*—two forms of persistent borrowing that are useful for defining sharing predicates. In §11.5, we explain that all forms of borrowing that we have seen so far can, under the hood, be expressed in terms of the lower-level mechanism of *indexed borrows*. Finally, in §11.6 and §11.7, we roughly sketch in two stages how the lifetime logic and indexed borrows can be implemented in Iris.

11.1 Full borrows and lifetime tokens

Figure 11.1 shows the main rules of the lifetime logic. We explain them by referring to the following Rust example, similar to the one in §8.5:

```
1  let mut v = vec![10, 11]; v.push(12);
2  let vptr = v.index_mut(1);
3  *vptr = 23;
4  println!("{:?}", v);
```

Here, we desugared `&mut v[1]` to show the underlying method call `v.index_mut(1)` in line 2. Similar to what we saw with `push` before, that call in turn is sugar for `Vec::index_mut(&mut v, 1)`. The type of `index_mut` looks as follows:

```
for<'a> fn(&'a mut Vec<i32>, usize) -> &'a mut i32
```

To call this function, we thus need a borrow at some lifetime κ (which we will use to instantiate `'a`). To get started, we need to create this lifetime. This is the role of LftL-begin: it lets us perform an Iris view shift to create a fresh lifetime κ and gives us the full *lifetime token* $[\kappa]_1$

Lifetime tokens (all primitive).

LftL-begin
$$\mathsf{True} \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \exists\kappa.\, [\kappa]_1 * \big([\kappa]_1 \mathrel{\Rrightarrow\!\!\!\triangleright\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} [\dagger\kappa]\big)$$

LftL-tok-timeless
$$\mathsf{timeless}([\kappa]_q)$$

LftL-end-persist
$$\mathsf{persistent}([\dagger\kappa])$$

LftL-tok-fract
$$[\kappa]_{q+q'} \mathrel{*\!\!*} [\kappa]_q * [\kappa]_{q'}$$

LftL-tok-inter
$$[\kappa \sqcap \kappa']_q \mathrel{*\!\!*} [\kappa]_q * [\kappa']_q$$

LftL-end-inter
$$[\dagger\, \kappa \sqcap \kappa'] \mathrel{*\!\!*} [\dagger\kappa] \vee [\dagger\kappa']$$

LftL-tok-unit
$$\mathsf{True} \mathrel{-\!\!*} [\varepsilon]_q$$

LftL-end-unit
$$[\dagger\varepsilon] \mathrel{-\!\!*} \mathsf{False}$$

LftL-not-own-end
$$[\kappa]_q * [\dagger\kappa] \mathrel{-\!\!*} \mathsf{False}$$

Lifetimes κ form a cancellable commutative monoid with operation $\sqcap$ and unit ε.

Lifetime inclusion (all derived from the definition).

LftL-incl
$$\kappa \sqsubseteq \kappa' \;:=\; \Box\Big(\big(\forall q.\, ([\kappa]_q \;\; \propto_{\mathcal{N}_{\mathsf{lft}}} \;\; q'.\, [\kappa']_{q'})\big) \;*\; \big([\dagger\kappa'] \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} [\dagger\kappa]\big)\Big)$$

LftL-incl-refl
$$\kappa \sqsubseteq \kappa$$

LftL-incl-trans
$$\kappa_1 \sqsubseteq \kappa_2 * \kappa_2 \sqsubseteq \kappa_3 \mathrel{-\!\!*} \kappa_1 \sqsubseteq \kappa_3$$

LftL-incl-isect
$$\kappa \sqcap \kappa' \sqsubseteq \kappa$$

LftL-incl-glb
$$\kappa \sqsubseteq \kappa' * \kappa \sqsubseteq \kappa'' \mathrel{-\!\!*} \kappa \sqsubseteq \kappa' \sqcap \kappa''$$

Full borrows (primitive; see Figure 11.8 for more primitive rules).

LftL-borrow
$$\triangleright P \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa}_{\mathsf{full}}\, P * \big([\dagger\kappa] \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \triangleright P\big)$$

LftL-bor-split
$$\&^{\kappa}_{\mathsf{full}}(P * Q) \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa}_{\mathsf{full}}\, P * \&^{\kappa}_{\mathsf{full}}\, Q$$

Full borrows (derived).

LftL-bor-acc
$$\&^{\kappa}_{\mathsf{full}}\, P * [\kappa]_q \;\; \propto_{\mathcal{N}_{\mathsf{lft}}} \;\; \triangleright P$$

LftL-reborrow
$$\kappa' \sqsubseteq \kappa * \&^{\kappa}_{\mathsf{full}}\, P \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa'}_{\mathsf{full}}\, P * \big([\dagger\kappa'] \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa}_{\mathsf{full}}\, P\big)$$

LftL-bor-exists
$$\frac{\text{domain of } x \text{ is inhabited}}{\&^{\kappa}_{\mathsf{full}}(\exists x.\, P) \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \exists x.\, \&^{\kappa}_{\mathsf{full}}\, P}$$

LftL-bor-shorten
$$\kappa' \sqsubseteq \kappa \mathrel{-\!\!*} \&^{\kappa}_{\mathsf{full}}\, P \mathrel{-\!\!*} \&^{\kappa'}_{\mathsf{full}}\, P$$

Figure 11.1: Selected primitive and derived rules of the lifetime logic.

witnessing that this lifetime is ongoing. (This token can then be split into fractional lifetime tokens $[\kappa]_q$—see below.) It also provides the view shift $[\kappa]_1 \mathrel{\Rrightarrow\!\!\!\triangleright\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} [\dagger\kappa]$: we will use this view shift later to end κ by exchanging the full lifetime token $[\kappa]_1$ for a *dead token*, written $[\dagger\kappa]$, indicating that κ has ended.[1]

Once the lifetime has been created, we can borrow the vector `v` at the lifetime κ in order to pass a borrowed reference to `index_mut`. This is allowed by LftL-borrow, really the core rule of the lifetime logic:

$$\triangleright P \quad \mathrel{\Rrightarrow\!\!\!*} \quad \underbrace{\&^{\kappa}_{\mathsf{full}}\, P}_{\text{ownership } \textit{during } \kappa} \quad * \underbrace{\big([\dagger\kappa] \mathrel{\Rrightarrow\!\!\!*} \triangleright P\big)}_{\text{ownership } \textit{after } \kappa}$$

This rule splits ownership of a resource $\triangleright P$ (in our example, the vector `v`) into the separating conjunction of a *full borrow* $\&^{\kappa}_{\mathsf{full}}\, P$ and an *inheritance* $[\dagger\kappa] \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \triangleright P$. The borrow grants access to P *during* the

[1] Note that the ending view shift is a "view shift that takes a step" $\mathrel{\Rrightarrow\!\!\!\triangleright\!\!\!*}$ rather than a normal view shift $\mathrel{\Rrightarrow\!\!\!*}$. This is syntactic sugar for $\mathrel{-\!\!*} \mathrel{\vDash\!\!\!\Rightarrow} \triangleright \mathrel{\vDash\!\!\!\Rightarrow}$. As we will see in §11.6, for technical reasons related to step-indexing, we have to restrict the view shift to be only used when performing a physical step of computation.

lifetime κ, while the inheritance allows us to retrieve ownership of $\triangleright P$ *after* κ has ended.[2] In other words, LftL-borrow *splits ownership in time*. The separating conjunction indicates that the two operands are "disjoint", which means we can safely transfer ownership of the borrow to `index_mut` and keep ownership of the inheritance for ourselves to use later. The unusual aspect of this is that we do not have disjointness *in space* (*e.g.*, in the memory), since both the borrow and the inheritance grant access to the same resource. Instead, we have disjointness *in time*: the lifetime κ is *either* ongoing *or* ended, so the borrow and the inheritance are never useful at the same time.

It is not worth going into the actual implementation of `index_mut` here. The relevant part is what `index_mut` does with respect to ownership. First, the ownership of the memory used by the vector ("inside" the full borrow) is split into two parts: (1) the ownership of the accessed vector position, and (2) the ownership of the rest of the vector. Then, the rule LftL-bor-split is used to split the full borrow into two full borrows dedicated to each of these parts. The full borrow of part (1) is returned to the caller; this matches the return type of `index_mut`. On the other hand, the full borrow of part (2) is dropped.[3] This means that the ownership of the rest of the vector is effectively lost until the lifetime ends, at which point it can be recovered using the inheritance. Supporting this pattern of forgetting about some borrowed resources and recovering them using the inheritance is a key feature of the lifetime logic.

The next step of our program is the write to `*vptr` in line 3. Recall that the type of `vptr` is `&mut i32`, which represents ownership of a full borrow of a single memory location. In order to perform this write, we need to *access* this full borrow and get the resource it contains (in particular, the points-to predicate $\ell \mapsto \bar{v}$). This is what LftL-bor-acc does.[4] If we give it a full borrow $\&^{\kappa}_{\mathsf{full}}\, P$ and a lifetime token $[\kappa]_q$, witnessing that κ is alive, then we get the resource P (later). Moreover, we also obtain the view shift $\triangleright P \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa}_{\mathsf{full}}\, P * [\kappa]_q$: this can be used when we are done with P, in order to reconstitute the full borrow and get back the lifetime token. Notice that the view shifts we are using here are *non-mask-changing*, which means we can keep the borrow open for as long as we want, not just for an atomic instant. But if we ever want to get the lifetime token back, we need to reconstitute (or "close") the borrow eventually—and in our proofs, we will always be forced to give back all the lifetime tokens that we obtained. This can be seen, for example, in ty-share: a token is provided as a premise to this view shift, but the same token must also be returned again in the conclusion.

Finally, at the end of line 3 of our example, `vptr` is not needed any more and it is time to end κ. To this end, we apply the view shift $[\kappa]_1 \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} [\dagger\kappa]$ that we obtained when κ was created.[5] In doing so, we have to give up the lifetime token $[\kappa]_1$ (ensuring that all borrows are closed again), but we get back the dead token $[\dagger\kappa]$, which can be used to prove that κ has indeed ended. Now that κ has ended, we can use our inheritance $[\dagger\kappa] \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \triangleright P$ to get back the ownership of `v` before printing it in line 4. Note that the dead token $[\dagger\kappa]$ is persistent (LftL-end-persist), so it can be used multiple times—this expresses that once the lifetime has ended, it can

[2] The *later* modality ($\triangleright$) is needed here for the same reason as in §5.1: to avoid unsoundness due to impredicativity. And as before, whenever P is *timeless*, we can ignore the modality ($\models\!\!\Rrightarrow$-timeless).

[3] We can do this because, as discussed in §6.2, Iris is an *affine* logic, which means it is possible to give up ownership of resources at any time.

[4] This rule uses syntactic sugar for symmetric accessors that we defined in §5.6.

[5] This involves stripping a later, which we will do using wp-step (page 75).

never be "un-ended". We may have to use the token many times because there could be many borrows (and thus many inheritances we wish to use) at the same lifetime. Each *inheritance*, however, may of course be used only once.

One important feature of the lifetime logic that this example does not demonstrate is the parameter q of a lifetime token $[\kappa]_q$, which is a fraction. Lifetime tokens can always be split into smaller parts, in a reversible fashion (LftL-tok-fract). This is needed when we want to access several full borrows with the same lifetime at the same time, or to witness that a lifetime is ongoing in several threads simultaneously. We also use lifetime tokens to express that a lifetime cannot be both dead and alive at the same time (LftL-not-own-end); we will see the use of this principle in §11.3.

Besides LftL-bor-split, borrows also enjoy the structural rule LftL-bor-exists, which lets us commute an existential quantifier "out" of a borrow (assuming the domain of quantification is inhabited). This basically "freezes" the witness x: in $\&^{\kappa}_{\mathsf{full}}(\exists x.\, P)$, we get a fresh x each time we access the borrow and we can pick an arbitrary x as part of closing the borrow; after using LftL-bor-split the witness x is fixed and cannot be changed any more. (LftL-bor-split looks like it could be invertible, and indeed it is; we will discuss its inverse LftL-bor-merge in §11.5.)

11.2 Lifetime inclusion

In §8.5 and §9, we have seen that Rust relates lifetimes by *lifetime inclusion*. This is used for subtyping (T-bor-lft) and reborrowing (C-reborrow).

With the lifetime logic, we need to define what it *means* for a lifetime κ to be "included" in another κ'. The key property of lifetime inclusion is that when the shorter κ is still alive, then so is the longer κ'. From the perspective of lifetime tokens, this means that, given a token for κ, we should be able to obtain a token for κ'. Conversely, given a dead token for κ', we should be able to obtain a dead token for κ, as well. This is expressed in the definition of lifetime inclusion as follows:

$$\kappa \sqsubseteq \kappa' \quad := \quad \Box\Big(\Big(\forall q.\, \big([\kappa]_q \;\; \propto_{\mathcal{N}_{\mathsf{lft}}} \;\; q'.\, [\kappa']_{q'}\big)\Big) \;*\; \big([\dagger\kappa'] \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} [\dagger\kappa]\big)\Big)$$

We use a variant of symmetric accessors with a binder, which are defined as follows:

$$Q \quad {}^{\mathcal{E}_1}{\propto}^{\mathcal{E}_2} \;\; x.\, P(x) := Q \mathrel{-\!\!*} {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} \big(\exists x.\, P(x) * (P(x) \mathrel{-\!\!*} {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_1} Q)\big)$$

In other words, when applying the accessor, the client learns about some witness x that is used in P. When using the closing view shift, x must not have changed.

The first part of lifetime inclusion says that we can trade a fraction of the token of κ for a potentially different fraction of the token of κ'.[6] Since the accessor is symmetric, it also provides a way to revert this trading to recover the original token of κ, so that no token is permanently lost. Crucially, the entire fraction q' has to be "given up" when trading back; there is no way to keep some part of the token for κ'. The second

[6] Note that the accessor is non-mask-changing, so there is no atomicity restriction to using it.

part of this definition is the analogue for dead tokens. Note that since dead tokens are persistent, it is not necessary to provide a way to recover the dead token that is passed in. The entire definition is wrapped in Iris's persistence modality $\square$ (§5.4) to make lifetime inclusion a persistent assertion that can be reused as often as needed.

It is easy to show that lifetime inclusion is a preorder (LftL-incl-refl, LftL-incl-trans). Inclusion can be used to *shorten* a full borrow (LftL-bor-shorten): If a full borrow is valid for a long lifetime, then it should also be valid for the shorter one. This rule justifies subtyping based on lifetimes in λ_{Rust}.

An even stronger use of lifetime inclusion is *reborrowing*, expressed by LftL-reborrow. This rule is used to prove the reborrowing rule in the type system, C-reborrow. Unlike shortening, reborrowing provides an inheritance to regain the initial full borrow after the shorter lifetime has ended.[7] This may sound intuitively plausible, but turns out to be extremely subtle. In fact, most of the complexity in the model of the lifetime logic arises from reborrowing.

Lifetime intersection. Beyond having a preorder, lifetimes also have a greatest lower bound: given two lifetimes κ and κ', their *intersection* $\kappa \sqcap \kappa'$ is the lifetime that ends whenever either of the operands ends (LftL-incl-glb).[8]

Lifetime intersection is particularly useful to create a fresh lifetime that is a sublifetime of some existing κ. We invoke the rule LftL-begin to create an auxiliary lifetime α_0, and then we use the intersection $\alpha := \kappa \sqcap \alpha_0$ as our new lifetime. By the rules of greatest lower bounds (LftL-incl-isect), it follows that $\alpha \sqsubseteq \kappa$. In the type system, we use this in the proof of F-newlft to create a new lifetime α that is shorter than all the lifetimes in $\bar{\kappa}$.

As we will see later, lifetime intersection also comes up when dealing with "nested" borrows, *i.e.*, borrows of borrows.

Intersection of lifetimes interacts well with lifetime tokens: a token of the intersection is composed of tokens of both operands, at the same fraction (LftL-tok-inter).[9] In other words, the intersection is alive if and only if both operands are alive. Similarly, the intersection has ended if and only if either operand has ended (LftL-end-inter). These laws let us do the token trading required by lifetime inclusion, showing that intersection indeed is the greatest lower bound for $\sqsubseteq$ (LftL-incl-isect, LftL-incl-glb).

Furthermore, intersection is commutative, associative and cancellative. It also has a unit ε.[10] This lifetime never ends (LftL-end-unit) and we can freely obtain any fraction of the token for it (LftL-tok-unit).[11] We use ε to model the **static** lifetime of λ_{Rust}, corresponding to `'static` in Rust.

11.3 Fractured borrows

Full borrows and lifetimes are powerful tools for modeling temporary ownership in Iris, and we use them to define the semantic interpretation

[7] Shortening can *almost* be derived from reborrowing by dropping the inheritance. However, shortening is just a plain magic wand, whereas reborrowing involves a view shift—and some uses of shortening would not work with that extra update modality of the view shift in the way. That is why we have both rules.

[8] The concrete definition of lifetime intersection is an implementation detail of the lifetime logic; we will give it in §11.6. For now, it suffices to say that lifetimes are treated symbolically, and intersection just tracks which symbols have been intersected. In particular, intersection is *not* idempotent.

[9] One somewhat surprising consequence of this property is that $\kappa \sqcap \kappa \neq \kappa$. Making intersection idempotent is incompatible with LftL-tok-inter.

[10] ε is also a "top" element w.r.t. inclusion.

[11] Note that $q := 1$ is a possible choice. We can even get more than fraction 1 of the token. This is why we have no rule that $[\kappa]_q$ implies $q \leq 1$.

Lftl-bor-fracture

$$\&^{\kappa}_{\mathsf{full}}\, \Phi(1) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa}_{\mathsf{frac}}\, \Phi$$

Lftl-fract-shorten

$$\frac{\kappa' \sqsubseteq \kappa}{\&^{\kappa}_{\mathsf{frac}}\, \Phi \mathrel{-\!\!*} \&^{\kappa'}_{\mathsf{frac}}\, \Phi}$$

Lftl-fract-persist

$$\mathsf{persistent}(\&^{\kappa}_{\mathsf{frac}}\, \Phi)$$

Lftl-fract-acc

$$\frac{\Box\, \forall q_1, q_2.\, \Phi(q_1 + q_2) \mathrel{*\!*} \Phi(q_1) * \Phi(q_2)}{\&^{\kappa}_{\mathsf{frac}}\, \Phi \mathrel{-\!\!*} \left([\kappa]_q \;\; \propto_{\mathcal{N}_{\mathsf{lft}}} \;\; q'.\, \triangleright \Phi(q')\right)}$$

Lftl-fract-acc-atomic

$$\&^{\kappa}_{\mathsf{frac}}\, \Phi \mathrel{-\!\!*} \left(\mathsf{True} \;\; {}^{\mathcal{N}_{\mathsf{lft}}}\!\propto^{\emptyset} \;\; b, q.\, \mathsf{ifthenelse}(b, \triangleright \Phi(q), [\dagger\kappa])\right)$$

Figure 11.2: Proof rules for fractured borrows (all derived).

of mutable references in λ_{Rust}. However, they cannot be used as-is for modeling Rust's shared references. In §10.4, we used the notion of *fractured borrows* as a key ingredient for defining the default read-only sharing predicate. Figure 11.2 gives the main reasoning rules for fractured borrows.[12]

The key point of fractured borrows is that they are *persistent*, and thus can be shared between many parties. This ability comes at the cost of a weaker accessor: while full borrows grant access to their *entire* content (Lftl-bor-acc), a fractured borrow only provides access to *some fraction* of its content (Lftl-fract-acc). To express this, fractured borrows $\&^{\kappa}_{\mathsf{frac}}\, \Phi$ work on a *predicate* Φ over fractions that has to be compatible with addition: $\Phi(q_1 + q_2) \mathrel{*\!*} \Phi(q_1) * \Phi(q_2)$.[13] The rule consumes $\&^{\kappa}_{\mathsf{frac}}\, \Phi$ to create the accessor, so that resource is not given back by the accessor—but that would be unnecessary since $\&^{\kappa}_{\mathsf{frac}}\, \Phi$ is persistent, so it can be used any number of times. When using Lftl-fract-acc to access the content of the fractured borrow, we get $\Phi(q')$ for some unknown fraction q'. This works because no matter how many threads access the same fractured borrow at the same time, it is always possible to give out *some* tiny fraction of Φ and keep some remainder available for the next thread. Similarly to full borrows, Lftl-fract-acc requires a lifetime token for witnessing that the lifetime is alive, and gives back the lifetime token only when the resource is returned. And again the accessor is non-mask-changing, so the borrow can be kept open for an extended, non-atomic period of time.

The persistence of fractured borrows is required for their use as a sharing predicate (ty-shr-persist). We also need a lifetime shortening rule (Lftl-fract-shorten) for ty-shr-mono. To create fractured borrows in the first place, Lftl-bor-fracture can turn a full borrow into a fractured one.[14] This rule is used to prove ty-share.

Dynamic lifetime inclusion. Fractured borrows have an interesting interaction with lifetime inclusion. Assume we have a fractured borrow with lifetime κ of a lifetime token for another lifetime κ'. That is, assume $\Phi(1) = [\kappa']_q$. For Φ to be suitably compatible with addition, we define $\Phi(q') = [\kappa']_{q' \cdot q}$: owning the *full* fraction of this borrow means owning fraction q of the token; smaller fractions of the borrow are obtained via multiplication. Then we can derive $\kappa \sqsubseteq \kappa'$:

Lftl-incl-fract

$$\left(\&^{\kappa}_{\mathsf{frac}}\, \lambda q'.\, [\kappa']_{q' \cdot q}\right) \mathrel{-\!\!*} \kappa \sqsubseteq \kappa'$$

[12] Inference rules with Iris propositions above and below the bar are to be interpreted as Iris entailment, or (equivalently) a magic wand.

[13] The compatibility of addition is wrapped in the persistence modality □ (§5.4) to make sure we can apply that property multiple times.

[14] That rule looks pretty unidirectional, but remember that we can reborrow: by using Lftl-reborrow before applying Lftl-bor-fracture, we can fracture a full borrow for some lifetime and then re-gain full access once that lifetime has ended.

The reason this principle intuitively makes sense is that with the token of lifetime κ' being borrowed at κ, it is impossible to end κ' while κ is still ongoing: ending κ' needs the full token, but part of that token is stuck in κ and can only be recovered through an inheritance once κ has ended.

To prove LftL-incl-fract, we first consider the left conjunct of lifetime inclusion:

$$\left(\&^{\kappa}_{\mathsf{frac}}\, \lambda q'.\, [\kappa']_{q'\cdot q}\right) \mathrel{-\!\!*} \forall q.\, \left([\kappa]_q \;\propto_{\mathcal{N}_{\mathsf{lft}}}\; q''.\, [\kappa']_{q''}\right)$$

This unfolds to:

$$\left(\&^{\kappa}_{\mathsf{frac}}\, \lambda q'.\, [\kappa']_{q'\cdot q}\right) \mathrel{-\!\!*} \forall q.\, [\kappa]_q \mathrel{\Rrightarrow\!\!\ast}_{\mathcal{N}_{\mathsf{lft}}} \exists q''.\, [\kappa']_{q''} * \left([\kappa']_{q''} \mathrel{\Rrightarrow\!\!\ast}_{\mathcal{N}_{\mathsf{lft}}} [\kappa]_q\right)$$

This is almost equivalent to LftL-fract-acc. Starting with fraction q of the token for κ, we obtain some q' such that $\triangleright [\kappa']_{q'\cdot q} * \left(\triangleright [\kappa']_{q'\cdot q} \mathrel{\Rrightarrow\!\!\ast}_{\mathcal{N}_{\mathsf{lft}}} [\kappa]_q\right)$. With $q'' := q' \cdot q$, that is exactly what we want to prove, modulo some $\triangleright$. And since lifetime tokens are timeless, the $\triangleright$ make no difference ($\mathrel{|\!\!\Rrightarrow}$-timeless, $\triangleright$-intro) and we are done.

This completes one half of the proof for LftL-incl-fract. What is missing is a way to turn a dead token for κ' into a dead token for κ. This mechanism is provided by LftL-fract-acc-atomic. The key differences to LftL-fract-acc are that (a) it does not require a lifetime token, and (b) it is mask-changing, and thus can only be used atomically. Conceptually, this rule performs a case distinction: at the atomic instant where the rule is used, either the lifetime is still ongoing (and thus the accessor can provide atomic access to some fraction of the borrow), or else we have a proof that the lifetime has ended (a dead token). The Boolean $b \in \mathbb{B}$ records which of these cases applies.[15] Here, $\mathsf{ifthenelse}$ is defined as follows:

$$\mathsf{ifthenelse}(b, x, y) := \begin{cases} x & \text{if } b = 1 \ (\textit{i.e.},\ \ b \text{ is “true”}) \\ y & \text{otherwise} \end{cases}$$

Now we can complete the proof of LftL-incl-fract by showing:

$$\left(\&^{\kappa}_{\mathsf{frac}}\, \lambda q'.\, [\kappa']_{q'\cdot q}\right) \mathrel{-\!\!*} [\dagger\kappa'] \mathrel{\Rrightarrow\!\!\ast}_{\mathcal{N}_{\mathsf{lft}}} [\dagger\kappa]$$

Using LftL-fract-acc-atomic, we can distinguish two cases: either we obtain a dead token for κ, so we are done.[16] Or else we obtain $\triangleright \Phi(q'')$, which in our case means we obtain some fraction $(q'' \cdot q)$ of the token for κ'. But we also have a dead token for κ', so by LftL-not-own-end, we have a contradiction.[17]

Deriving lifetime inclusions from fractured borrows significantly expands the power of lifetime inclusion. So far, we have seen that we can use lifetime intersection to make a fresh α a sublifetime of some existing κ; however, for this to work out, we have to decide *in advance* which other lifetimes α is going to be a sublifetime of. Using fractured borrows, we can establish additional lifetime inclusions *dynamically* by borrowing one lifetime's token at another lifetime, when the involved lifetimes are already ongoing and in active use. Some interior mutable types like `RefCell<T>` or `RwLock<T>` allow sharing data structures for a lifetime that cannot be established in advance, and we thus found this scheme for proving lifetime inclusion dynamically crucial in proving the safety of such types.

[15] It may seem like we could equivalently use a disjunction, but that is not the case: remember that the right-hand side of an accessor describes both the resources that the accessor provides to its client, and the resources that the client gives back (in §5.6, these are P_1 and P_2, respectively). A disjunction would permit the client to use a *different* disjunct when closing the accessor than what was used when opening it. The use of an explicit Boolean here prevents the client from opening the accessor with the left disjunct ($\triangleright \Phi(q)$) and later closing it with the right one ($[\dagger\kappa]$), or vice versa.

[16] The token is persistent, so we can keep it when closing the borrow.

[17] This exploits that lifetime tokens are timeless, so we can get rid of the later.

LftL-bor-at
$$\mathcal{N} \mathrel{\#} \mathcal{N}_{\mathsf{lft}} * \&^{\kappa}_{\mathsf{full}}\, P \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa/\mathcal{N}}_{\mathsf{at}}\, P$$

LftL-at-shorten
$$\frac{\kappa' \sqsubseteq \kappa}{\&^{\kappa/\mathcal{N}}_{\mathsf{at}}\, P \mathrel{-\!\!*} \&^{\kappa'/\mathcal{N}}_{\mathsf{at}}\, P}$$

LftL-at-persist
$$\mathsf{persistent}(\&^{\kappa/\mathcal{N}}_{\mathsf{at}}\, P)$$

LftL-at-acc-tok
$$\&^{\kappa/\mathcal{N}}_{\mathsf{at}}\, P \mathrel{-\!\!*} \left([\kappa]_q \;\; {}^{\mathcal{N}_{\mathsf{lft}},\mathcal{N}}\!\!\propto^{\mathcal{N}_{\mathsf{lft}}} \;\; \triangleright P\right)$$

Figure 11.3: Proof rules for atomic borrows (all derived).

11.4 Atomic borrows

In the previous section, we have seen a form of borrowing that is persistent, but only provides access to some fraction of the borrowed resources. Next we will consider *atomic borrows*, which combine the persistence of fractured borrows with full access to the borrowed resources like full borrows. But of course, there is a catch: atomic borrows can only be accessed for a single atomic instant; it is not possible to keep them open for an extended duration. In that sense, atomic borrows are very similar to invariants, except that they are integrated into the lifetime logic.[18] And indeed, the main use-case for atomic borrows is in the verification of Rust types like `Mutex<T>` or `RwLock<T>`: concurrent synchronization primitives that would usually be verified with an invariant governing the shared state. As we will see in §13.2, the invariant gets replaced by an atomic borrow.

The basic rules for atomic borrows are given in Figure 11.3. Similar to invariants $\boxed{P}^{\mathcal{N}}$, atomic borrows $\&^{\kappa/\mathcal{N}}_{\mathsf{at}}\, P$ come with a namespace $\mathcal{N}$ that is used to ensure that an atomic borrow cannot be opened twice (*i.e.*, this avoids reentrancy). They can be created from full borrows via LftL-bor-at, and are subject to lifetime inclusion as usual (LftL-at-shorten). The accessor LftL-at-acc-tok says that given some fraction of the lifetime token, we have access to the full content $\triangleright P$ of the borrow, but in so doing we have to remove $\mathcal{N}$ from our mask—this ensures that the accessor has to be closed again within the same atomic instant.[19]

11.5 Indexed borrows: Unifying persistent borrowing

We have discussed two different forms of persistent borrows (fractured and atomic borrows). In the next chapter, we will even introduce a third form of persistent borrows called *non-atomic borrows* (§12.3). It would be quite unsatisfying if all of these would be "primitive" in the sense that their correctness is justified by direct appeal to the underlying model of borrows in Iris. Instead, following the usual Iris approach, we would much rather have a minimal "(base) lifetime logic" from which more advanced forms of borrowing can be derived. This leads to several layers of abstraction built on top of each other, as shown in Figure 11.4. The core lifetime logic consists of lifetime tokens, lifetime inclusion, full borrows and *indexed borrows* which we introduce in this section. As we will see next, the other

[18] In fact, they are even more similar to *cancellable* invariants (§5.2), but with cancellation being controlled by a lifetime instead of a token per invariant. The use-cases for cancellable invariants and the various forms of borrowing in the lifetime logic are very similar: in both cases, some resources are temporarily shared, but eventually full ownership of the resources needs to be reclaimed. The lifetime logic is like cancellable invariants "on steroids".

[19] The mask $\mathcal{N}_{\mathsf{lft}}, \mathcal{N}$ indicates the disjoint union of $\mathcal{N}_{\mathsf{lft}}$ and $\mathcal{N}$.

forms of borrowing are built on top of that core. Furthermore, in §11.6 we will see that the lifetime logic itself rests on yet another abstraction dubbed “boxes”.

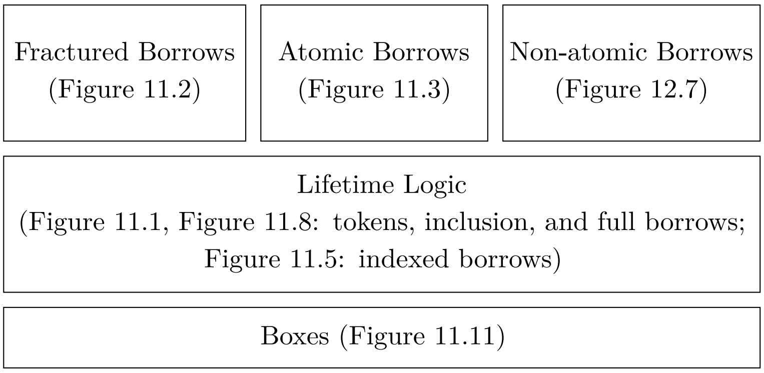

Figure 11.4: Lifetime logic abstractions.

The key to establishing derived borrowing principles such as fractured borrows and atomic borrows is to put the borrow into an Iris invariant. This lets us share one borrow amongst everyone with access to the fractured/atomic borrow, and make sure they all follow a common protocol. The problem with this approach, as we saw in §5.1, is that putting a borrow into an invariant means we have to add a $\triangleright$ modality in front of the borrow. Borrows are not timeless,[20] so this extra modality prevents us from using any of the lifetime logic proof rules.

Thus, the plan is to introduce a mechanism that lets us put borrows into invariants *without* having to pay tribute to step-indexing by adding a $\triangleright$ modality. The lifetime logic achieves this using *indexed borrows.*[21]

An indexed borrow can be used to split a full borrow $\&^{\kappa}_{\mathsf{full}}\, P$ into a timeless token $[\mathsf{Bor} : i]_1$ and a persistent proposition $\&^{\kappa}_{i}\, P$ (LftL-bor-idx). The index i is used to tie the two pieces together. By putting the timeless part into an invariant and keeping the persistent part “next to” the invariant (it is persistent, so can be shared amongst all relevant parties just like the invariant itself), we can effectively share access to a full borrow without a $\triangleright$ modality being added.[22]

Intuitively, $\&^{\kappa}_{i}\, P$ says that P is *or was* a borrowed proposition at lifetime κ with index i. The token $[\mathsf{Bor} : i]_1$ expresses ownership of said borrow, which also ensures that the borrow still exists (and has not been, for example, split using LftL-bor-split). The token comes with a fraction, but the only rule that lets us do anything with *partial* ownership of the token is LftL-idx-acc-atomic (which is needed to derive LftL-fract-acc-atomic). Indexed borrows are subject to shortening as usual (LftL-idx-shorten), and they can be accessed non-atomically like full borrows (LftL-idx-acc). Furthermore, if P and Q are equivalent, then a borrow of P is equivalent to a borrow of Q (LftL-idx-iff).[23]

To explain the more obscure proof rules for indexed borrows, we will look at how to define atomic and fractured borrows in terms of lifetime logic primitives (indexed borrows and lifetime tokens). We will also see how LftL-reborrow follows from LftL-idx-bor-unnest.

[20] Higher-order ghost state (§6.1) is needed internally to define full borrows. This makes them non-timeless.

[21] It turns out that same mechanism can also be used to derive LftL-reborrow from lower-level reasoning principles—this is because reborrowing is closely related to dealing with “nested” borrows, and putting a borrow into a borrow incurs the same $\triangleright$-related problems as putting a borrow into an invariant.

[22] Or rather, timelessness allows us to use $\Rrightarrow$-timeless to remove the modality again.

[23] We demand the equivalence under the persistence modality to ensure that the equivalence proof itself does not consume any resources. Also, it is sufficient to prove them “later” equivalent, which will be crucial for subtyping in λ_{Rust}.

LftL-bor-idx
$$\&^{\kappa}_{\mathsf{full}}\, P \mathrel{*\!*} \exists i.\ \&^{\kappa}_{i}\, P * [\mathsf{Bor} : i]_1$$

LftL-idx-persist
$$\mathsf{persistent}(\&^{\kappa}_{i}\, P)$$

LftL-idx-timeless
$$\mathsf{timeless}([\mathsf{Bor} : i]_q)$$

LftL-idx-iff
$$\frac{\triangleright \Box (P \mathrel{*\!*} Q)}{\&^{\kappa}_{i}\, P \mathrel{-\!\!*} \&^{\kappa}_{i}\, Q}$$

LftL-idx-shorten
$$\frac{\kappa' \sqsubseteq \kappa}{\&^{\kappa}_{i}\, P \mathrel{-\!\!*} \&^{\kappa'}_{i}\, P}$$

LftL-idx-fract
$$[\mathsf{Bor} : i]_{q+q'} \mathrel{*\!*} [\mathsf{Bor} : i]_q * [\mathsf{Bor} : i]_{q'}$$

LftL-idx-acc
$$\&^{\kappa}_{i}\, P \mathrel{-\!\!*} \left([\mathsf{Bor} : i]_1 * [\kappa]_q \ \propto_{\mathcal{N}_{\mathsf{lft}}}\ \triangleright P\right)$$

LftL-idx-acc-atomic
$$\&^{\kappa}_{i}\, P \mathrel{-\!\!*} \left([\mathsf{Bor} : i]_q \ {}^{\mathcal{N}_{\mathsf{lft}}}\!\!\propto^{\emptyset}\ b.\, \mathsf{ifthenelse}(b, \triangleright P, [\dagger\kappa])\right)$$

LftL-idx-bor-unnest
$$\&^{\kappa}_{i}\, P * \&^{\kappa'}_{\mathsf{full}}([\mathsf{Bor} : i]_1) \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa \sqcap \kappa'}_{\mathsf{full}}\, P$$

Figure 11.5: Proof rules for indexed borrows (all primitive).

11.5.1 Atomic borrows

In order to derive the proof rules in Figure 11.3, we define atomic borrows as follows:

$$\&^{\kappa/\mathcal{N}}_{\mathsf{at}}\, P := \exists i.\, (\mathcal{N} \mathrel{\#} \mathcal{N}_{\mathsf{lft}}) * \&^{\kappa}_{i}\, P * \boxed{[\mathsf{Bor} : i]_1}^{\mathcal{N}} \qquad (\text{at-bor})$$

In other words, an atomic borrow simply puts the full borrow token into an invariant. The indexed borrow is kept separately, together with a proof that $\mathcal{N}$ and $\mathcal{N}_{\mathsf{lft}}$ are disjoint. With this, LftL-at-persist is immediate. Turning a full borrow into an atomic borrow (LftL-bor-at) follows directly from LftL-bor-idx and creating an invariant to put the borrow token into. LftL-at-shorten is a consequence of LftL-idx-shorten.

The only really interesting rule is LftL-at-acc-tok, and it is not very hard to derive either. We show a proof outline in Figure 11.6.[24] We unfold the symmetric accessor, revealing our full goal:

$$\&^{\kappa/\mathcal{N}}_{\mathsf{at}}\, P * [\kappa]_q \ {}^{\mathcal{N}_{\mathsf{lft}},\mathcal{N}}\!\!\Rrightarrow\!\!\!*^{\mathcal{N}_{\mathsf{lft}}}\ \triangleright P * \left(\triangleright P \ {}^{\mathcal{N}_{\mathsf{lft}}}\!\!\Rrightarrow\!\!\!*^{\mathcal{N}_{\mathsf{lft}},\mathcal{N}}\ [\kappa]_q * \&^{\kappa/\mathcal{N}}_{\mathsf{at}}\, P\right)$$

After introducing our assumptions and unfolding the atomic borrow, we open the invariant in $\mathcal{N}$. That gives us full access to the (timeless) borrow token. Next, we apply LftL-idx-acc.[25] Here we exploit that $\mathcal{N} \mathrel{\#} \mathcal{N}_{\mathsf{lft}}$: we can still use proof rules of the lifetime logic even though we have the invariant in $\mathcal{N}$ already open. With this, we obtain $\triangleright P$, so we can prove the first separating conjunct after the view shift.

We can use all our remaining resources for the second conjunct, so our context consists of the two closing view shifts (for the invariant and the borrow), and the goal is:

$$\triangleright P \ {}^{\mathcal{N}_{\mathsf{lft}}}\!\!\Rrightarrow\!\!\!*^{\mathcal{N}_{\mathsf{lft}},\mathcal{N}}\ [\kappa]_q * \&^{\kappa/\mathcal{N}}_{\mathsf{at}}\, P$$

We start by adding $\triangleright P$ to our context (bringing us back to the same context as before), after which we first close the borrow and then close the invariant. This completes the proof.

[24] The way to read these proof outlines is as follows: between curly braces and in green, we show the current context. The orange resources are persistent and remain implicitly available throughout the rest of the outline without being repeated everywhere. We also use orange text when referring back to them.

Most steps only change the context, in which case we only show how the context keeps evolving—until the goal becomes relevant, at which point we explicitly state it. This is the current goal for all contexts shown above, so to determine the goal at some point in the proof, look down below the current context to the next **Goal** statement.

We use numbered abbreviations to avoid repeating the same terms over and over; those abbreviations are introduced by the bold numbers and refer to the formula immediately following that number.

[25] Note that we could not use LftL-idx-acc-atomic as that would remove $\mathcal{N}_{\mathsf{lft}}$ from our mask, but we have to keep those invariants enabled.

$\left\{[\kappa]_q * \&_{\mathsf{at}}^{\kappa/\mathcal{N}}\, P\right\}_{\mathcal{N}_{\mathsf{lft}},\mathcal{N}}$

$\left\{[\kappa]_q * \mathcal{N} \mathrel{\#} \mathcal{N}_{\mathsf{lft}} * \&_i^{\kappa}\, P * \boxed{[\mathsf{Bor} : i]_1}^{\mathcal{N}}\right\}_{\mathcal{N}_{\mathsf{lft}},\mathcal{N}}$

Open invariant (inv-acc) in $\mathcal{N}$, get closing view shift: $(\mathbf{1})\ \triangleright [\mathsf{Bor} : i]_1 \ {}^{\mathcal{N}_{\mathsf{lft}}}{\Rrightarrow\!\!\ast}^{\mathcal{N}_{\mathsf{lft}},\mathcal{N}}\ \mathsf{True}$.

Due to $\mathcal{N} \mathrel{\#} \mathcal{N}_{\mathsf{lft}}$, we know that $\mathcal{N}_{\mathsf{lft}}$ remains in the mask.

$\left\{[\kappa]_q * [\mathsf{Bor} : i]_1 * (1)\right\}_{\mathcal{N}_{\mathsf{lft}}}$

Use LftL-idx-acc with $\&_i^{\kappa}\, P$, get closing view shift: $(\mathbf{2})\ \triangleright P \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} [\mathsf{Bor} : i]_1 * [\kappa]_q$.

$\{\triangleright P * (1) * (2)\}_{\mathcal{N}_{\mathsf{lft}}}$

Goal: ${}^{\mathcal{N}_{\mathsf{lft}},\mathcal{N}}{\models\!\!\!\Rrightarrow}^{\mathcal{N}_{\mathsf{lft}}} \left(\triangleright P * \left(\triangleright P \ {}^{\mathcal{N}_{\mathsf{lft}}}{\Rrightarrow\!\!\ast}^{\mathcal{N}_{\mathsf{lft}},\mathcal{N}}\ [\kappa]_q\right)\right)$

Introduce update modality, and discharge first separating conjunct.

$\{(1) * (2)\}$ (No mask because our goal does not start with a fancy update modality.)

Goal: $\triangleright P \ {}^{\mathcal{N}_{\mathsf{lft}}}{\Rrightarrow\!\!\ast}^{\mathcal{N}_{\mathsf{lft}},\mathcal{N}}\ [\kappa]_q$

Introduce assumption of magic wand.

$\{(1) * (2) * \triangleright P\}_{\mathcal{N}_{\mathsf{lft}}}$

Use (2).

$\left\{(1) * [\mathsf{Bor} : i]_1 * [\kappa]_q\right\}_{\mathcal{N}_{\mathsf{lft}}}$

Use (1).

$\left\{[\kappa]_q\right\}_{\mathcal{N}_{\mathsf{lft}},\mathcal{N}}$

Goal: ${}^{\mathcal{N}_{\mathsf{lft}}}{\models\!\!\!\Rrightarrow}^{\mathcal{N}_{\mathsf{lft}},\mathcal{N}}\ [\kappa]_q$

Figure 11.6: Proof outline for LftL-at-acc-tok.

11.5.2 *Fractured borrows*

The goal is to derive the proof rules in Figure 11.2 (page 146). Ownership of a fractured borrow is defined roughly as follows:[26]

$$\begin{aligned}
\&_{\mathsf{frac}}^{\kappa}\, \Phi &:= \exists \kappa', \gamma, i.\ \kappa \sqsubseteq \kappa' * \boxed{\exists q_i.\ [\mathsf{Bor} : i]_{q_i}}^{\mathcal{N}_{\mathsf{lft}}} * \&_i^{\kappa'}\, I_\Phi(\kappa') \\
I_\Phi(\kappa') &:= \exists q.\ \Phi(q) * \boxed{q}^{\gamma} * (q = 1 \lor [\kappa']_{1-q})
\end{aligned}$$

First of all, the actual lifetime of the borrow is some κ' that includes κ. This is used to justify LftL-fract-shorten.[27] Beyond this, we quantify over the index i of the underlying borrow, and the ghost name γ of a fractional token. This token named γ is not to be confused with the borrow token; in fact, fractured borrows have to handle *three* kinds of tokens: borrow tokens $[\mathsf{Bor} : i]_q$, lifetime tokens $[\kappa]_q$, and their own custom tokens $\boxed{q}^{\gamma}$. This custom token is needed to avoid having to require $\Phi(q) \mathrel{-\!\!*} q \leq 1$ in LftL-fract-acc.

The two key ingredients of a fractured borrow are:

- an invariant that always contains *some* fraction of the token for our borrow. We can thus always open the invariant, take out some part of what is in there, and put back the rest. This gives rise to the following lemma:

$$\Box \models\!\!\!\Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} \exists q_i.\ [\mathsf{Bor} : i]_{q_i} \qquad \text{(frac-bor-tok)}$$

Intuitively, the invariant serves as a "dispenser" where we can always get some fraction of the token.[28] This also means that it will be impossible to ever recover full ownership of that token again, but that is okay—we will never need more than a fraction of it.

[26] The actual definition is slightly different: fractured borrows are really defined in terms of atomic borrows. However, after inlining the definition of atomic borrows, the result is equivalent to what we have given here. Also note that $[\kappa']_{1-q}$ implicitly assumes that $1 - q$ is positive and thus $q < 1$.

[27] Such a down-closure is a standard technique for achieving monotonicity. It was not needed for atomic borrows as they do not use κ in an invariant position (the lifetime token).

[28] In other words, $\exists q_i.\ [\mathsf{Bor} : i]_{q_i}$ is *almost* persistent, except that we have to eliminate an update modality every time we need some more of the token.

- a borrow of some fraction q of Φ together with the same fraction of our custom token. On top of that, the borrow also contains fraction $1 - q$ of the lifetime token.[29] This setup implies that the *sum* of the fraction of the lifetime token that we own ($[\kappa']_{1-q}$), and of the fraction that we own of our own custom token and borrowed proposition ($\boxed{q}^{\gamma} * \Phi(q)$), is always 1. This is crucial when accessing the borrow to take out some fraction of Φ, as it means we can "trade" these different tokens against each other. Assuming $\Box\, \forall q_1, q_2.\, \Phi(q_1 + q_2) \ast\!\!\ast\, \Phi(q_1) * \Phi(q_2)$, we have:

$$I_\Phi(\kappa') * \boxed{q}^{\gamma} * \Phi(q) \twoheadrightarrow\!\!\!\ast\; I_\Phi(\kappa') * [\kappa']_q \qquad (\text{FRAC-BOR-TRADE}1)$$

$$I_\Phi(\kappa') * [\kappa']_q \twoheadrightarrow\!\!\!\ast\; I_\Phi(\kappa') * \exists q_0.\, [\kappa']_{q-q_0} * \boxed{q_0}^{\gamma} * \Phi(q_0) \qquad (\text{FRAC-BOR-TRADE}2)$$

  Notice how I stays unchanged, and otherwise we swap fraction q of one resource against the same fraction of something else. The second rule is a bit more complicated because we have to ensure that there is always *some* fraction of Φ left in I, so we only convert some unknown fraction q_0 of the lifetime token, and keep the rest.[30]

Both of these pieces (and lifetime inclusion) are persistent, giving rise to LftL-fract-persist.

Let us now look at how one would prove LftL-fract-acc. The proof outline is shown in Figure 11.7. After unfolding the accessor notation our goal is:[31]

$$\&^{\kappa}_{\mathsf{frac}}\, \Phi * [\kappa]_{q_\kappa} \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \exists q_0.\, \triangleright \Phi(q_0) * \left(\triangleright \Phi(q_0) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} [\kappa]_{q_\kappa}\right) \qquad (\text{FRAC-ACC})$$

Unfolding the fractured borrow reveals the actual lifetime κ' of the borrow. We also obtain $\kappa \sqsubseteq \kappa'$, which (by the definition of lifetime inclusion) means we can trade our κ token against *some* fraction $q_{\kappa'}$ of κ'. The first main step of the proof is now described by the following lemma:

$$[\kappa']_{q_{\kappa'}} * \boxed{\exists q_i.\, [\mathsf{Bor} : i]_{q_i}}^{\mathcal{N}_{\mathsf{lft}}} * \&^{\kappa}_{i}\, I_\Phi(\kappa') \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}}$$
$$\exists q_i, q_0.\, [\kappa']_{q_{\kappa'}-q_0} * [\mathsf{Bor} : i]_{q_i} * \triangleright \Phi(q_0) * \boxed{q_0}^{\gamma}$$

Using the invariant (via frac-bor-tok), we can get hold of some fraction q_i of the borrow token. Then we open the indexed borrow. We cannot use LftL-idx-acc as that would require *full* ownership of the borrow token where we just own fraction q_i, so instead we use LftL-idx-acc-atomic.[32] This rule works without a lifetime token and either grants access to the borrowed resources, or else provides a dead token for the lifetime (similar to LftL-fract-acc-atomic). Since we actually own a lifetime token, we can exclude the second case using LftL-not-own-end to obtain a contradiction. Thus we obtain the content of the borrow (and a view shift to close the accessor again, not shown below):

$$[\kappa']_{q_{\kappa'}} * [\mathsf{Bor} : i]_{q_i} * \triangleright I_\Phi(\kappa')$$

Now we perform token trading using frac-bor-trade2: some fraction q_0 is subtracted from the lifetime token, and we obtain an equal fraction of

[29] Since 0 is not a valid fraction for tokens, we have to handle $q = 1$, where we do not own any part of the lifetime token, separately.

[30] Specifically, the proof of this lemma uses $q_0 := \mathsf{min}(q, q')/2$ where q' is the existentially quantified fraction in I. This is definitely smaller than q, so $q - q_0$ is positive, and definitely smaller than q', so some fraction of Φ is left.

[31] We freely use (un)currying when needed. In this case the goal literally unfolds to $\&^{\kappa}_{\mathsf{frac}}\, \Phi \twoheadrightarrow\!\!\!\ast\; [\kappa]_{q_\kappa} \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \ldots$, and we uncurried it to the form shown on the left.

[32] We anyway have no intention of keeping the borrow open for more than an instant.

$$\Big\{\Big((\mathbf{1})\ \Box\,\forall q_1, q_2.\, \Phi(q_1+q_2) \ast\!\!\ast\, \Phi(q_1) \ast \Phi(q_2)\Big) \ast [\kappa]_{q_\kappa} \ast \&^{\kappa}_{\mathsf{frac}}\, \Phi\Big\}_{\mathcal{N}_{\mathsf{lft}}}$$

$$\Big\{[\kappa]_{q_\kappa} \ast \kappa \sqsubseteq \kappa' \ast \boxed{\exists q_i.\, [\mathsf{Bor} : i]_{q_i}}^{\mathcal{N}_{\mathsf{lft}}} \ast \&^{\kappa}_{i}\, I_\Phi(\kappa')\Big\}_{\mathcal{N}_{\mathsf{lft}}}$$

Use frac-bor-tok. Use LftL-incl with $\kappa \sqsubseteq \kappa'$, get closing view shift: $(\mathbf{2})\ [\kappa']_{q_{\kappa'}} \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} [\kappa]_{q_\kappa}$.

$$\Big\{[\kappa']_{q_{\kappa'}} \ast [\mathsf{Bor} : i]_{q_i} \ast (2)\Big\}_{\mathcal{N}_{\mathsf{lft}}}$$

Use LftL-idx-acc-atomic on $\&^{\kappa}_{i}\, I_\Phi(\kappa')$; with $[\kappa']_{q_{\kappa'}}$ we can exclude the dead case. The tokens are timeless. Get closing view shift: $(\mathbf{3})\ \triangleright I_\Phi(\kappa')\ {}^{\emptyset}\!\Rrightarrow\!\!\ast^{\mathcal{N}_{\mathsf{lft}}} [\mathsf{Bor} : i]_{q_i}$.

$$\Big\{[\kappa']_{q_{\kappa'}} \ast \triangleright I_\Phi(\kappa') \ast (2) \ast (3)\Big\}_{\emptyset}$$

Use frac-bor-trade2 under $\triangleright$ modality ($\triangleright$-mono) with $q := q_{\kappa'}$ (needs (1)).

$$\Big\{[\kappa']_{q_{\kappa'}-q_0} \ast \triangleright I_\Phi(\kappa') \ast \triangleright \Phi(q_0) \ast \boxed{q_0}^{\gamma} \ast (2) \ast (3)\Big\}_{\emptyset}$$

Use (3).

$$\Big\{[\kappa']_{q_{\kappa'}-q_0} \ast [\mathsf{Bor} : i]_{q_i} \ast \triangleright \Phi(q_0) \ast \boxed{q_0}^{\gamma} \ast (2)\Big\}_{\mathcal{N}_{\mathsf{lft}}}$$

Goal: $\mathrel{|}\!\!\!\Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} \Big(\exists q_0.\, \triangleright \Phi(q_0) \ast \big(\triangleright \Phi(q_0) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} [\kappa]_{q_\kappa}\big)\Big)$

Introduce update modality, and discharge first separating conjunct.

$$\Big\{[\kappa']_{q_{\kappa'}-q_0} \ast [\mathsf{Bor} : i]_{q_i} \ast \boxed{q_0}^{\gamma} \ast (2)\Big\}$$

Goal: $\triangleright \Phi(q_0) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} [\kappa]_{q_\kappa}$

Introduce assumption of magic wand.

$$\Big\{[\kappa']_{q_{\kappa'}-q_0} \ast [\mathsf{Bor} : i]_{q_i} \ast \boxed{q_0}^{\gamma} \ast (2) \ast \triangleright \Phi(q_0)\Big\}_{\mathcal{N}_{\mathsf{lft}}}$$

Use LftL-idx-acc-atomic on $\&^{\kappa}_{i}\, I_\Phi(\kappa')$ (excluding dead case), get closing view shift: $(\mathbf{4})\ \triangleright I_\Phi(\kappa')\ {}^{\emptyset}\!\Rrightarrow\!\!\ast^{\mathcal{N}_{\mathsf{lft}}} [\mathsf{Bor} : i]_{q_i}$.

$$\Big\{[\kappa']_{q_{\kappa'}-q_0} \ast \boxed{q_0}^{\gamma} \ast (2) \ast \triangleright \Phi(q_0) \ast \triangleright I_\Phi(\kappa') \ast (4)\Big\}_{\emptyset}$$

Use frac-bor-trade1 under $\triangleright$ modality ($\triangleright$-mono) with $q := q_0$ (needs (1)).

$$\Big\{[\kappa']_{q_{\kappa'}-q_0} \ast \triangleright I_\Phi(\kappa') \ast [\kappa']_{q_0} \ast (2) \ast (4)\Big\}_{\emptyset}$$

Use (4).

$$\Big\{[\kappa']_{q_{\kappa'}} \ast (2) \ast [\mathsf{Bor} : i]_{q_i}\Big\}_{\mathcal{N}_{\mathsf{lft}}}$$

Use (2).

$$\Big\{[\kappa]_{q_\kappa} \ast [\mathsf{Bor} : i]_{q_i}\Big\}_{\mathcal{N}_{\mathsf{lft}}}$$

Goal: $\mathrel{|}\!\!\!\Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} [\kappa]_{q_\kappa}$

Figure 11.7: LftL-fract-acc proof outline.

Φ and the custom token. ($\triangleright$-mono says we can apply frac-bor-trade2 under a $\triangleright$, but then we get the result under a $\triangleright$ as well. All the tokens are timeless so we can strip the $\triangleright$ in front of them immediately.) Then we can close the borrow again, which completes the proof of the lemma.

This completes the opening phase of LftL-fract-acc: we have successfully obtained ownership of some fraction (q_0) of $\triangleright \Phi$, so we can discharge the first separating conjunct of our goal in frac-acc.

The second conjunct, the view shift, is basically the inverse of the first. We have to prove that we can close everything up again:

$$[\kappa']_{q_{\kappa'}-q_0} \ast [\mathsf{Bor} : i]_{q_i} \ast \&^{\kappa}_{i}\, I_\Phi(\kappa') \ast \triangleright \Phi(q_0) \ast \boxed{q_0}^{\gamma} \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} [\kappa']_{q_{\kappa'}}$$

This suffices because $[\kappa']_{q_{\kappa'}}$ is what we got by using the lifetime inclusion $\kappa \sqsubseteq \kappa'$, so the closing part of the lifetime inclusion accessor can be used to turn that back into $[\kappa']_{q_\kappa}$, which is all we need for frac-acc.

To prove this second main step, we use LftL-idx-acc-atomic again. Since we still own some fraction of the lifetime token for κ', we can again exclude the case where the lifetime has already ended. We also own fraction q_0 of both $\triangleright \Phi$ and our custom token, hence we can put those into the borrow, obtaining $[\kappa']_{q_0}$ in the trade (frac-bor-trade1). Together with the fraction $q_{\kappa'} - q_0$ of that token that we still own, this completes the proof of LftL-fract-acc.[33]

The proof of LftL-fract-acc-atomic is simpler; it just requires a single application of LftL-idx-acc-atomic—no need to trade back and forth with the tokens inside the borrow. The fractured borrow always contains *some* fraction of Φ, so we can always give the client direct access to that.

Finally, we consider LftL-bor-fracture:

$$\&^{\kappa}_{\mathsf{full}}\, \Phi(1) \mathrel{\Rrightarrow\!\!\ast}_{\mathcal{N}_{\mathsf{lft}}} \exists \kappa', \gamma, i.\, \kappa \sqsubseteq \kappa' * \boxed{\exists q_i.\, [\mathsf{Bor} : i]_{q_i}}^{\mathcal{N}_{\mathsf{lft}}} * \&^{\kappa'}_{i}\, I_{\Phi}(\kappa')$$

For this rule, we will need a stronger variant of LftL-bor-acc, which we show in Figure 11.8: we need to (a) *change* the content of the borrow, not just access it, and (b) do so without even owning any part of the lifetime token.

The former is achieved by LftL-bor-acc-strong. Note that this is an accessor, but it is not symmetric, so we have no convenient syntactic sugar. The key feature of LftL-bor-acc-strong is that it lets us *change* the borrowed proposition from P to Q, if we can also prove a view shift that can convert Q back into P. This proof can itself consume arbitrary resources (it does not have to be persistent), making this much stronger than just saying that full borrows are closed under equivalence. Moreover, this proof may to assume that the lifetime of the borrow has already ended, which will be crucial. Note that we do *not* learn that the lifetime κ that we ascribe to the borrow is dead, but the "true" lifetime κ' of the borrow without any shortening.[34]

For LftL-bor-fracture, we need a variant of LftL-bor-acc-strong which additionally does not require a lifetime token. This is achieved by LftL-bor-acc-atomic-strong, which is a lot like LftL-idx-acc-atomic that we already saw. The rule says that either we have atomic access to the borrow in the same way as LftL-bor-acc-strong, or else we get a proof that the lifetime has already ended.[35]

To show LftL-bor-fracture, we start by allocating some ghost state of type *Frac* (as defined in §4.1) for our custom token. We pick an initial value of 1, so afterwards we own $\boxed{1}^{\gamma}$.

The next step is to adjust the borrow of $\Phi(1)$, as described by the following helper lemma:

$$\&^{\kappa}_{\mathsf{full}}\, \Phi(1) * \boxed{1}^{\gamma} \mathrel{\Rrightarrow\!\!\ast}_{\mathcal{N}_{\mathsf{lft}}} \exists \kappa'.\, \kappa \sqsubseteq \kappa' * \&^{\kappa'}_{\mathsf{full}}\, I_{\Phi}(\kappa')$$

After applying that lemma, completing LftL-bor-fracture is just a matter of using LftL-bor-idx, followed by putting the borrow token into an invariant.

To prove the helper lemma, we apply LftL-bor-acc-atomic-strong. In case the lifetime is already over, we get a dead token, so we can pick

[33] At the end, we still own $[\mathsf{Bor} : i]_{q_i}$, which we throw away. This is no problem as we can always get more of that token via frac-bor-tok.

[34] Unfortunately, we cannot turn $[\dagger\kappa']$ into $[\dagger\kappa]$ because that view shift requires a mask of $\mathcal{N}_{\mathsf{lft}}$, and the strong accessor requires the mask to be empty.

[35] The point of ${}^{\emptyset}{\models\!\!\!\Rrightarrow}^{\mathcal{N}_{\mathsf{lft}}} \mathsf{True}$ is to let the client change the mask back to what it was before.

Primitive rules.

LftL-bor-merge
$$\&^{\kappa}_{\mathsf{full}}\, P * \&^{\kappa}_{\mathsf{full}}\, Q \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa}_{\mathsf{full}}(P * Q)$$

LftL-bor-fake
$$[\dagger\kappa] \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa}_{\mathsf{full}}\, P$$

LftL-bor-acc-strong
$$\&^{\kappa}_{\mathsf{full}}\, P * [\kappa]_q \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \exists \kappa'.\, \kappa \sqsubseteq \kappa' * \triangleright P * \Big(\forall Q.\, \triangleright(\triangleright Q * [\dagger\kappa'] \Rrightarrow\!\!\!\ast_{\emptyset} \triangleright P) * \triangleright Q \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa'}_{\mathsf{full}}\, Q * [\kappa]_q\Big)$$

LftL-bor-acc-atomic-strong
$$\&^{\kappa}_{\mathsf{full}}\, P \;{}^{\mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow\!\!\!\ast^{\emptyset} \Big(\exists \kappa'.\, \kappa \sqsubseteq \kappa' * \triangleright P * \Big(\forall Q.\, \triangleright(\triangleright Q * [\dagger\kappa'] \Rrightarrow\!\!\!\ast_{\emptyset} \triangleright P) * \triangleright Q \;{}^{\emptyset}\!\Rrightarrow\!\!\!\ast^{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa'}_{\mathsf{full}}\, Q\Big)\Big) \vee$$
$$\Big([\dagger\kappa] * {}^{\emptyset}\!\!\Rrightarrow^{\mathcal{N}_{\mathsf{lft}}} \mathsf{True}\Big)$$

Derived rules.

LftL-bor-unnest
$$\&^{\kappa'}_{\mathsf{full}}\, \&^{\kappa}_{\mathsf{full}}\, P \;\triangleright\!\!\Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa' \sqcap \kappa}_{\mathsf{full}}\, P$$

Figure 11.8: Advanced proof rules for full borrows.

$\kappa' := \kappa$ and "fake" the result using LftL-bor-fake—creating a borrow at an already dead lifetime is a trivial operation, as such a borrow is useless. In case the lifetime is still alive, we learn about the underlying lifetime κ' and $\kappa \sqsubseteq \kappa'$, so we can already discharge those parts of the goal. As the new borrow content we pick $I_{\Phi}(\kappa')$. This means we need to show that we have $\triangleright I_{\Phi}(\kappa')$:

$$\triangleright \Phi(1) * \boxed{1}^{\gamma} \Rrightarrow\!\!\!\ast_{\emptyset} \triangleright I_{\Phi}(\kappa')$$

and that it can be turned back into the original content of the borrow:

$$\triangleright I_{\Phi}(\kappa') * [\dagger\kappa'] \Rrightarrow\!\!\!\ast_{\emptyset} \triangleright \Phi(1)$$

In both case we can remove the later modality in both assumption and conclusion (▷-mono). The former is now trivial (pick $q := 1$ for the q hidden inside I). For the latter, LftL-not-own-end implies that $q = 1$, and then we are done. (Ownership of $\boxed{1}^{\gamma}$ gets thrown away in the process.)

These rules in Figure 11.8 are strong enough to derive LftL-bor-exists which we saw before: we can open the borrow using LftL-bor-acc-atomic-strong, and if κ is still ongoing we learn the current value of x, we pick the new borrow content Q to "freeze" that x, and we are done. In case the lifetime is already over, we obtain $[\dagger\kappa]$ which we use with LftL-bor-fake and the fact that the domain of x is inhabited to complete the proof.

The only remaining proof rule not mentioned yet in Figure 11.8 is LftL-bor-merge, which shows that LftL-bor-split is invertible. This rule will be shown in §11.7.

11.5.3 Reborrowing and other derived rules

Some of the proof rules in Figure 11.1 (page 142) can be derived from more primitive rules: LftL-incl-isect and LftL-incl-glb are fairly

$\{\kappa' \sqsubseteq \kappa * \&_{\mathsf{full}}^{\kappa} P\}_{\mathcal{N}_{\mathsf{lft}}}$

Use LftL-bor-idx, then borrow token at κ' (LftL-borrow).

$\left\{\&_{i}^{\kappa} P * \&_{\mathsf{full}}^{\kappa'} [\mathsf{Bor} : i]_1 * \left([\dagger\kappa'] \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}} \triangleright [\mathsf{Bor} : i]_1\right)\right\}_{\mathcal{N}_{\mathsf{lft}}}$

Apply LftL-idx-bor-unnest on $\&_{i}^{\kappa} P$.

$\left\{\&_{\mathsf{full}}^{\kappa \sqcap \kappa'} P * \left([\dagger\kappa'] \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \triangleright [\mathsf{Bor} : i]_1\right)\right\}_{\mathcal{N}_{\mathsf{lft}}}$

By LftL-incl-glb, $\kappa' \sqsubseteq \kappa$ and reflexivity of lifetime inclusion, we have $\kappa' \sqsubseteq \kappa \sqcap \kappa'$. Use that with LftL-bor-shorten.

$\left\{\&_{\mathsf{full}}^{\kappa'} P * \left([\dagger\kappa'] \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \triangleright [\mathsf{Bor} : i]_1\right)\right\}_{\mathcal{N}_{\mathsf{lft}}}$

Goal: $\mathrel{|\!\!\Rrightarrow}_{\mathcal{N}_{\mathsf{lft}}} \left(\&_{\mathsf{full}}^{\kappa'} P * \left([\dagger\kappa'] \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \&_{\mathsf{full}}^{\kappa} P\right)\right)$

Introduce update modality, and discharge first separating conjunct.

$\left\{\left([\dagger\kappa'] \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \triangleright [\mathsf{Bor} : i]_1\right)\right\}$

Goal: $[\dagger\kappa'] \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \&_{\mathsf{full}}^{\kappa} P$

Introduce assumption of magic wand.

$\left\{[\dagger\kappa'] * \left([\dagger\kappa'] \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \triangleright [\mathsf{Bor} : i]_1\right)\right\}_{\mathcal{N}_{\mathsf{lft}}}$

Use the view shift. Borrow tokens are timeless so we can use $\mathrel{|\!\!\Rrightarrow}$-timeless.

$\{[\mathsf{Bor} : i]_1\}_{\mathcal{N}_{\mathsf{lft}}}$

Use LftL-bor-idx with the borrow token and $\&_{i}^{\kappa} P$.

$\{\&_{\mathsf{full}}^{\kappa} P\}_{\mathcal{N}_{\mathsf{lft}}}$

Goal: $\mathrel{|\!\!\Rrightarrow}_{\mathcal{N}_{\mathsf{lft}}} \&_{\mathsf{full}}^{\kappa} P$

Figure 11.9: Proof outline for LftL-reborrow.

simple consequences of the definition of lifetime inclusion (LftL-incl). LftL-bor-acc can be derived from LftL-bor-acc-strong by picking $Q := P$. LftL-bor-shorten follows from LftL-idx-shorten.

But the most interesting derived rule is LftL-reborrow, which as we will see can be derived from LftL-idx-bor-unnest:

$$\kappa' \sqsubseteq \kappa * \&_{\mathsf{full}}^{\kappa} P \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \&_{\mathsf{full}}^{\kappa'} P * \left([\dagger\kappa'] \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \&_{\mathsf{full}}^{\kappa} P\right)$$

An outline for this proof is given in Figure 11.9.

Naively, we might want to just borrow $\&_{\mathsf{full}}^{\kappa} P$ at lifetime κ', but that would not work because borrowing requires a leading $\triangleright$ similar to invariants. Instead, we do what we did for atomic and fractured borrows: we break the borrow into its persistent and timeless parts, and only borrow the timeless part $[\mathsf{Bor} : i]_1$ at lifetime κ'. Adding a later there poses no problems, thanks to $\mathrel{|\!\!\Rrightarrow}$-timeless. At that point, we have:

$$\&_{i}^{\kappa} P * \&_{\mathsf{full}}^{\kappa'} [\mathsf{Bor} : i]_1 * \left([\dagger\kappa'] \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}} \triangleright [\mathsf{Bor} : i]_1\right)$$

The first two conjuncts exactly match LftL-idx-bor-unnest, so we apply that rule. Now we can see why it is called "unnesting": in some sense, $\&_{\mathsf{full}}^{\kappa'} [\mathsf{Bor} : i]_1$ is a nested borrow (a "borrow of a borrow"),[36] and the rule lets us remove the nesting and obtain a borrow at the intersection of the two lifetimes: $\&_{\mathsf{full}}^{\kappa \sqcap \kappa'} P$. To obtain the first separating conjunct of our goal, all we have to do is shorten the lifetime from $\kappa \sqcap \kappa'$ to κ', which LftL-incl-glb says is possible.[37] For the second part, we make use of the inheritance that we obtained when borrowing the borrow token: $[\dagger\kappa'] \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathsf{lft}}} \triangleright [\mathsf{Bor} : i]_1$. That is already almost what we want, we only need to turn $[\mathsf{Bor} : i]_1$ back into a full borrow, which we can do because

[36] Only the token of the inner borrow is borrowed, really, but since the rest is persistent that does not make a difference.

[37] With $\kappa' \sqsubseteq \kappa$, we can show that $\kappa \sqcap \kappa'$ and κ' mutually outlive each other—they are "equal", at least as far as lifetime inclusion is concerned.

the token is timeless (so we can remove the $\triangleright$) and the indexed borrow itself ($\&^{\kappa}_{i}\, P$) is persistent, and thus has not been "used up" by the unnesting. This concludes the proof.

We can also derive a variant of unnesting for full borrows, LftL-bor-unnest. Notice, however, that this uses a view shift that "takes a step", *i.e.*, this rule is only useful when the program also performs a step of computation. To derive LftL-bor-unnest, we start by turning $\&^{\kappa'}_{\mathsf{full}}\, \&^{\kappa}_{\mathsf{full}}\, P$ into the equivalent $\&^{\kappa'}_{\mathsf{full}}\, \exists i.\ \&^{\kappa}_{i}\, P * [\mathsf{Bor} : i]_1$ (via LftL-idx-iff). By LftL-bor-exists and LftL-bor-split, we have that there exists some i such that $\&^{\kappa'}_{\mathsf{full}}\, \&^{\kappa}_{i}\, P * \&^{\kappa'}_{\mathsf{full}}\, [\mathsf{Bor} : i]_1$. Indexed borrows are persistent, so we can open the first borrow with LftL-bor-acc-atomic-strong, make a copy and close it again.[38] This way, we obtain $\triangleright \&^{\kappa}_{i}\, P$. Now we can take the step that the rule permits us to take, and finish off the proof with LftL-idx-bor-unnest.

11.6 Implementing the lifetime logic (without reborrowing)

We have seen how indexed borrows form the foundation of all forms of persistent borrowing, and how they reduce high-level rules such as LftL-reborrow to lower-level principles. In terms of the picture in Figure 11.4 (page 149) we have taken care of the topmost layer. Now it is time to take a look at how lifetime tokens as well as indexed and full borrows themselves can be implemented in Iris, and how the primitive proof rules (Figure 11.8, Figure 11.5, and the parts of Figure 11.1 that we have not derived yet) can be verified. We will not discuss in detail the proof of every single proof rule, but pick a few representative ones.

For now, we restrict ourselves to the lifetime logic without reborrowing/unnesting, *i.e.*, rule LftL-idx-bor-unnest will not hold in this model. LftL-bor-merge internally relies on reborrowing, so it will not be available either. In the next section, we present a model that supports all these missing pieces. The restricted model is simpler because without reborrowing, we can think of each lifetime as entirely independent of all the others. We will also not have LftL-idx-iff, which helps simplify the model a bit more.

Before we consider the formal definition in all its glory, we give some intuition for how lifetimes and their borrows are represented and managed using Iris ghost state.

What is a lifetime? The first question to consider is: what *is* a lifetime κ? What is the *type* of such an object? When a new lifetime is created (LftL-begin), we call that lifetime an *atomic lifetime*, which is represented by a unique identifier (similar to ghost names γ, we pick identifiers from $\mathbb{N}$). But we also have to define lifetime intersection, and to this end we say that a lifetime is defined by the atomic lifetimes it is the intersection of: a lifetime is a multiset of atomic lifetimes. This is defined formally in Figure 11.10. We use a multiset because verifying LftL-tok-inter requires keeping track of "how often" an atomic lifetime was intersected into a lifetime.[39]

[38] If the lifetime κ' is already dead, then so is $\kappa' \sqcap \kappa$ and thus we can complete the proof easily with LftL-bor-fake.

[39] In other words, if we had $\kappa \sqcap \kappa = \kappa$, LftL-tok-inter could be used to duplicate tokens.

Each lifetime will come with some ghost state to manage it. Concretely, for each lifetime we need two ghost names to keep track of the borrows at that lifetime and the inheritances activated when the lifetime ends.[40] These ghost names are allocated on-demand when a composite (*i.e.*, not necessarily atomic) lifetime is used for the first time. To keep track of these ghost names, we use another piece of ghost state: the resource algebra $\mathit{ILft} := \mathit{Auth}(\mathit{Lft} \xrightharpoonup{\mathsf{fin}} \mathit{Ag}(\mathbb{N} \times \mathbb{N}))$ authoritatively associates each lifetime with two ghost names (*i.e.*, natural numbers). The lifetime logic is parameterized by a global ghost name γ_{i} and will keep the authoritative element of this RA in a global invariant; the fragments that express knowledge about the ghost name of a particular lifetime are persistent and can thus be shared freely.

Talking about ownership of some lifetime's ghost state thus requires an indirection: first we own some part of γ_{i} to learn the actual ghost name γ used by the lifetime, and then we own some ghost state at γ. We introduce OwnBor and OwnInh to abstract away that indirection (Figure 11.10).

We also need some ghost state to represent lifetime tokens and dead tokens. The rule LftL-tok-inter already indicates that the token for a composite lifetime consists of tokens for all the involved atomic lifetimes, and that is indeed how the lifetime logic is defined: we have another piece of global ghost state (besides γ_{i}) which tracks, for each atomic lifetime, whether it is still alive. This ghost state uses the RA $\mathit{ALft} := \mathit{Auth}(\mathit{ALft} \xrightharpoonup{\mathsf{fin}} \mathit{Frac} +_{\lightning} ())$, which authoritatively associates each atomic lifetime with something akin to the oneshot RA (§4.1), except there is no data to agree about when the "shot" has happened: the unit RA $()$ only has a single persistent element, so once the transition from the left-hand side to the right-hand side of the sum has happened, we can never switch back. $\mathsf{inr}()$ serves as a witness that this transition has happened. $\mathsf{inl}(q)$ is a token we can own which shows that the transition has *not* yet happened, and the lifetime is still alive.

Thus we can already define the first parts of the public interface to the lifetime logic that we have seen before:

$$
\begin{aligned}
[\kappa]_q &:= \mathop{\Huge\ast}_{\Lambda \in \kappa} \boxed{\circ\, [\Lambda \leftarrow \mathsf{inl}(q)]}^{\gamma_{\mathsf{a}}} \\
[\dagger\kappa] &:= \exists \Lambda \in \kappa.\, \boxed{\circ\, [\Lambda \leftarrow \mathsf{inr}()]}^{\gamma_{\mathsf{a}}}
\end{aligned}
$$

To own fraction q of a composite lifetime, we need to own that fraction of *each* involved lifetime.[41] To show that a composite lifetime has ended, it is sufficient to demonstrate that *any* involved lifetime has ended.

Here we make use of another global parameter of the lifetime logic, γ_{a}, which indicates the ghost name we use to manage atomic lifetimes. γ_{i} and γ_{a}, and the namespace $\mathcal{N}_{\mathsf{lft}}$ we have already seen in the proof rules, are the only two parameters we will need. The lifetime logic needs more than one invariant, so we separate $\mathcal{N}_{\mathsf{lft}}$ into three disjoint sub-namespaces (which is always possible with all namespaces): $\mathcal{N}_{\mathsf{mgmt}}$ will contain the main management invariant, $\mathcal{N}_{\mathsf{bor}}$ will contain invariants to manage borrowed resources, and $\mathcal{N}_{\mathsf{inh}}$ will contain invariants to manage inheritances.

[40] In principle, we could bundle all ghost state of a lifetime under a single name, but using multiple names is more convenient as most of the time, we only talk about one of the two components.

[41] The big separating conjunction here takes multiplicities into account, so if an atomic lifetime Λ occurs multiple times in κ, we also own its token multiple times.

Domains.

$$\begin{aligned}
\Lambda \in \mathit{ALft} &:= \mathbb{N} && \text{(atomic lifetimes)}\\
\kappa \in \mathit{Lft} &:= \wp^{\mathsf{fin},+}(\mathit{ALft}) && \text{(lifetimes)}\\
i \in \mathit{Idx} &:= \mathit{Lft} \times \mathbb{N} && \text{(borrow indices)}\\
I \in \mathit{ILft} &:= \mathit{Auth}(\mathit{Lft} \overset{\mathsf{fin}}{\rightharpoonup} \mathit{Ag}(\mathbb{N} \times \mathbb{N})) && \text{(global ghost state to manage lifetimes; name: } \gamma_{\mathsf{i}}\text{)}\\
A \in \mathit{ALft} &:= \mathit{Auth}(\mathit{ALft} \overset{\mathsf{fin}}{\rightharpoonup} \mathit{Frac} +_{\lightning} ()) && \text{(global ghost state for atomic lifetimes; name: } \gamma_{\mathsf{a}}\text{)}\\
\mathit{BorSt} &:= \mathsf{in} \mid \mathsf{open}(q : \mathit{Frac}) && \text{(state of a borrow box)}\\
\mathit{BorBox} &:= \mathit{Auth}(\mathbb{N} \overset{\mathsf{fin}}{\rightharpoonup} \mathit{Ag}(\mathit{BorSt}) \times \mathit{Frac}) && \text{(per-lifetime ghost state to manage borrow box; } \gamma_{\mathsf{bor}}\text{)}\\
\mathit{InhBox} &:= \mathit{Auth}(\wp^{\mathsf{fin}}(\mathbb{N})) && \text{(per-lifetime ghost state to manage inheritance box; } \gamma_{\mathsf{inh}}\text{)}
\end{aligned}$$

Helper functions.

$$\begin{aligned}
\mathsf{OwnBor}(\kappa, a) &:= \exists \gamma_{\mathsf{bor}}.\, \boxed{\circ\, [\kappa \leftarrow \gamma_{\mathsf{bor}}, \_, \_]}^{\gamma_{\mathsf{i}}} * \boxed{a : \mathit{BorBox}}^{\gamma_{\mathsf{bor}}}\\
\mathsf{OwnInh}(\kappa, a) &:= \exists \gamma_{\mathsf{inh}}.\, \boxed{\circ\, [\kappa \leftarrow \_, \_, \gamma_{\mathsf{inh}}]}^{\gamma_{\mathsf{i}}} * \boxed{a : \mathit{InhBox}}^{\gamma_{\mathsf{inh}}}
\end{aligned}$$

Intersection, tokens and inclusion.

$$\begin{aligned}
\kappa_1 \sqcap \kappa_2 &:= \kappa_1 \cup \kappa_2 \qquad \text{(this is multiset union, adding up the multiplicities)}\\
[\kappa]_q &:= \mathop{\Huge\ast}_{\Lambda \in \kappa} \boxed{\circ\, [\Lambda \leftarrow \mathsf{inl}(q)]}^{\gamma_{\mathsf{a}}}\\
[\dagger\kappa] &:= \exists \Lambda \in \kappa.\, \boxed{\circ\, [\Lambda \leftarrow \mathsf{inr}()]}^{\gamma_{\mathsf{a}}}\\
\kappa \sqsubseteq \kappa' &:= \Box \Big(\Big(\forall q.\, \big([\kappa]_q \;\propto_{\mathcal{N}_{\mathsf{lft}}}\; q'.\, [\kappa']_{q'}\big) \Big) \;*\; \big([\dagger\kappa'] \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} [\dagger\kappa]\big) \Big)
\end{aligned}$$

Figure 11.10: Lifetime logic domains, helper functions and tokens (without reborrowing).

At this point, we can already show the shape of the global invariant that all lifetime logic proof rules assume:

$$\begin{aligned}
\mathsf{LftLInv} &:= \exists A, I.\, \boxed{\bullet A}^{\gamma_{\mathsf{a}}} * \boxed{\bullet I}^{\gamma_{\mathsf{i}}} * \mathop{\Huge\ast}_{\kappa \in \mathsf{dom}(I)} \mathsf{LftInv}(A, \kappa)\\
\mathsf{LftLCtx} &:= \boxed{\mathsf{LftInv}}^{\mathcal{N}_{\mathsf{mgmt}}}
\end{aligned}$$

All rules are proven assuming LftLCtx is in their context. In that invariant, we maintain ownership of the authoritative state for lifetime ghost names and atomic lifetimes. Everything else is captured by the per-lifetime invariant LftInv, and that is where things become more involved.

At a high level, each lifetime κ is in one of two possible states, as indicated by the states of atomic lifetimes tracked in A: either it is alive (so all atomic lifetimes $\Lambda \in \kappa$ are alive), or it is dead (so some atomic lifetime is dead).

$$\begin{aligned}
\mathsf{LftAliveIn}(A, \kappa) &:= \forall \Lambda \in \kappa.\, \exists q.\, A(\Lambda) = \mathsf{inl}(q)\\
\mathsf{LftDeadIn}(A, \kappa) &:= \exists \Lambda \in \kappa.\, A(\Lambda) = \mathsf{inr}()\\
\mathsf{LftInv}(A, \kappa) &:= \mathsf{LftAlive}(\kappa) * \mathsf{LftAliveIn}(A, \kappa) \lor{}\\
&\qquad \mathsf{LftDead}(\kappa) * \mathsf{LftDeadIn}(A, \kappa)
\end{aligned}$$

But what exactly are the resources held in the invariant for each lifetime? The rough idea is that there is some amount of borrowed resources that is currently being managed by the lifetime. Let us call that P_B. Each time LftL-borrow is used, the borrowed proposition P is being added to the P_B for that lifetime. However, P_B is not always fully owned by the lifetime logic: after all, users can open borrows and take resources "out of" the lifetime logic using LftL-bor-acc-strong. This is a non-mask-changing accessor, so the lifetime invariant has to be maintained even while the user holds on to these resources.[42]

To model this, we say that P_B is equal to the separating conjunction of all propositions P_i that have been borrowed at this lifetime: $P_B = P_0 * P_1 * \ldots * P_n$. A borrow logically represents ownership of one such piece. Each of these pieces is currently either held inside the invariant, or it has been given out to the client that owns it. The partitioning of P_B into these "slices" can be changed using rules like LftL-bor-split (which splits a slice into two), LftL-bor-merge (which merges two slices into one), and of course LftL-borrow (which adds a new slice). To separate the handling of lifetimes and borrows from the handling of these "slices", we introduce a separate abstraction for the latter: the idea of a *box*. This separation helps manage the quite large proof of the lifetime logic by breaking it into smaller pieces.

11.6.1 Boxes

A box is a logical data structure that manages some proposition P partitioned into slices P_i such that $P = \mathop{\ast}_i P_i$. Boxes support taking the resources out of each slice separately and putting them back in. Being able to take out the resources of a slice without outright removing it from the box is useful when the overall content of the box is part of a larger protocol—in the lifetime logic, we will have two boxes whose overall content is related (one for borrows, one for inheritances), and still we need to be able to take out the resources of individual slices of the borrow box.

The proof rules for a box are given in Figure 11.11. The two key propositions of this abstraction are $\mathsf{Box}(\mathcal{N}, P, f)$ and $\mathsf{BoxSlice}(\mathcal{N}, Q, \iota)$. $\mathsf{Box}(\mathcal{N}, P, f)$ says that in namespace $\mathcal{N}$, we own a box with total resources P, and $f \in \mathbb{N} \xrightarrow{\mathsf{fin}} \{\mathsf{empty}, \mathsf{full}\}$ describes the current status of each slice of this box, identified by its slice name $\iota \in \mathbb{N}$: each slice is either currently full (the resources of this slice are held in the box) or it is currently empty (the resources have been given out). $\mathsf{BoxSlice}(\mathcal{N}, Q, \iota)$ (which is persistent) says that the slice named ι manages proposition Q.

Initially, a box is created with no slices (Box-create). Then we can use Slice-insert-empty to add an empty slice with arbitrary (to-be-filled in the future) content Q to a box. That changes the total resources of the box from P to $P * Q$, and the slice map f is extended with some fresh slice name ι to record the new empty slice.[43]

A slice thus created can be filled using Slice-fill, which consumes the slice proposition Q and changes the slice state from empty to full. This action can be undone using Slice-empty, which gives back ownership of Q (albeit under a later, just like invariants[44]). Composing Slice-insert-empty and Slice-fill leads to Slice-insert-full, a rule that lets us add an already filled slice to a box.

[42] The only other primitive rule to access a borrow is LftL-bor-acc-atomic-strong, and that rule is mask-changing, so we can keep the invariant open while the user accesses the resources.

[43] This rule uses the *conditional later modality* $\triangleright^b$, which is indexed by a Boolean b: if b is true, then this is equal to $\triangleright$, otherwise it is just the identity modality (*i.e.*, the identity function on *iProp*). The proof rules involving conditional laters can be used either with or without the modality; the key part is that the use of the same Boolean b throughout the rule means that either *all* the indexed laters have to be present, or none of them. This is needed because the lifetime logic sometimes has to (carefully) work with resources under a later, with no chance of getting rid of the modality.

[44] Under the hood, boxes are implemented with higher-order ghost state (§6.1).

Basic proof rules.

Box-create

$$\mathsf{True} \Rrightarrow_{\mathcal{N}} \mathsf{Box}(\mathcal{N}, \mathsf{True}, \emptyset)$$

Slice-persist

$$\mathsf{persistent}(\mathsf{BoxSlice}(\mathcal{N}, Q, \iota))$$

Slice-insert-empty

$$\triangleright^b \mathsf{Box}(\mathcal{N}, P, f) \Rrightarrow_{\mathcal{N}} \exists \iota \notin \mathsf{dom}(f).\ \mathsf{BoxSlice}(\mathcal{N}, Q, \iota) * \triangleright^b \mathsf{Box}(\mathcal{N}, P * Q, f[\iota \leftarrow \mathsf{empty}])$$

Slice-delete-empty

$$\frac{f(\iota) = \mathsf{empty}}{\mathsf{BoxSlice}(\mathcal{N}, Q, \iota) \vdash \triangleright^b \mathsf{Box}(\mathcal{N}, P, f) \Rrightarrow_{\mathcal{N}} \exists P'.\ \triangleright^b(\triangleright(P = (P' * Q)) * \mathsf{Box}(\mathcal{N}, P', f[\iota \leftarrow \bot]))}$$

Slice-fill

$$\frac{f(\iota) = \mathsf{empty}}{\mathsf{BoxSlice}(\mathcal{N}, Q, \iota) \vdash \triangleright^b Q * \triangleright \mathsf{Box}(\mathcal{N}, P, f) \Rrightarrow_{\mathcal{N}} \triangleright^b \mathsf{Box}(\mathcal{N}, P, f[\iota \leftarrow \mathsf{full}])}$$

Slice-empty

$$\frac{f(\iota) = \mathsf{full}}{\mathsf{BoxSlice}(\mathcal{N}, Q, \iota) \vdash \triangleright^b \mathsf{Box}(\mathcal{N}, P, f) \Rrightarrow_{\mathcal{N}} \triangleright Q * \triangleright^b \mathsf{Box}(\mathcal{N}, P, f[\iota \leftarrow \mathsf{empty}])}$$

Box-fill

$$\frac{\forall \iota \in \mathsf{dom}(f).\ f(\iota) = \mathsf{empty}}{\triangleright P * \mathsf{Box}(\mathcal{N}, P, f) \Rrightarrow_{\mathcal{N}} \mathsf{Box}(\mathcal{N}, P, f[\iota \leftarrow \mathsf{full} \mid \iota \in \mathsf{dom}(f)])}$$

Box-empty

$$\frac{\forall \iota \in \mathsf{dom}(f).\ f(\iota) = \mathsf{full}}{\mathsf{Box}(\mathcal{N}, P, f) \Rrightarrow_{\mathcal{N}} \triangleright P * \mathsf{Box}(\mathcal{N}, P, f[\iota \leftarrow \mathsf{empty} \mid \iota \in \mathsf{dom}(f)])}$$

Derived proof rules.

Slice-insert-full

$$\triangleright Q * \triangleright^b \mathsf{Box}(\mathcal{N}, P, f) \Rrightarrow_{\mathcal{N}} \exists \iota \notin \mathsf{dom}(f).\ \Box\, \mathsf{BoxSlice}(\mathcal{N}, Q, \iota) * \triangleright^b \mathsf{Box}(\mathcal{N}, P * Q, f[\iota \leftarrow \mathsf{full}])$$

Slice-delete-full

$$\frac{f(\iota) = \mathsf{full}}{\mathsf{BoxSlice}(\mathcal{N}, Q, \iota) \vdash \triangleright^b \mathsf{Box}(\mathcal{N}, P, f) \Rrightarrow_{\mathcal{N}} \triangleright Q * \exists P'.\ \triangleright^b(\triangleright(P = P' * Q) * \mathsf{Box}(\mathcal{N}, P', f[\iota \leftarrow \bot]))}$$

Slice-split

$$\frac{f(\iota) = s}{\begin{array}{r}\mathsf{BoxSlice}(\mathcal{N}, Q_1 * Q_2, \iota) \vdash \triangleright^b \mathsf{Box}(\mathcal{N}, P, f) \Rrightarrow_{\mathcal{N}} \exists \iota_1 \notin \mathsf{dom}(f), \iota_2 \notin \mathsf{dom}(f).\ \iota_1 \neq \iota_2 * \\ \Box\, \mathsf{BoxSlice}(\mathcal{N}, Q_1, \iota_1) * \Box\, \mathsf{BoxSlice}(\mathcal{N}, Q_2, \iota_2) * \triangleright^b \mathsf{Box}(\mathcal{N}, P, f[\iota \leftarrow \bot][\iota_1 \leftarrow s][\iota_2 \leftarrow s])\end{array}}$$

Slice-merge

$$\frac{\iota_1 \neq \iota_2 \qquad f(\iota_1) = f(\iota_2) = s}{\begin{array}{r}\mathsf{BoxSlice}(\mathcal{N}, Q_1, \iota_1), \mathsf{BoxSlice}(\mathcal{N}, Q_2, \iota_2) \vdash \triangleright^b \mathsf{Box}(\mathcal{N}, P, f) \Rrightarrow_{\mathcal{N}} \exists \iota \notin \mathsf{dom}(f) \setminus \{\iota_1, \iota_2\}\,. \\ \Box\, \mathsf{BoxSlice}(\mathcal{N}, Q_1 * Q_2, \iota) * \triangleright^b \mathsf{Box}(\mathcal{N}, P, f[\iota_1 \leftarrow \bot][\iota_2 \leftarrow \bot][\iota \leftarrow s])\end{array}}$$

Figure 11.11: Proof rules for a box.

Slices can also be removed again: Slice-delete-empty says that we can take any slice and remove it from the box. This of course has an effect on the proposition P describing the full content of the box: we have to "remove" Q. Concretely, we learn that the previous content P is actually equal to $Q * P'$ for some P' describing the "remainder" of the content.[45] Slice-delete-full shows that we can also delete a slice that is currently full, and doing so returns its content to us.

Given $\mathsf{BoxSlice}(\mathcal{N}, Q_1 * Q_2, \iota)$, we can *split* this slice by removing it, and then adding two new slices for Q_1 and Q_2, respectively. This is expressed by the derived rule Slice-split. The total resources P, of course, do not change when doing this. The newly created slices are either both empty or both full, depending on the status of the old slice. Similarly, Slice-merge lets us *merge* two slices (that have to have identical state: either both empty or both full). This again happens by removing the two old slices and adding a new one.

And finally, we can fill and empty *the entire box*, without even knowing the contributions of the individual slices: Box-empty says that if all slices are full, we can empty them all and obtain $\triangleright P$; and Box-fill lets us do the inverse when all slices are empty. These two proof rules are really what makes boxes so powerful: they let us maintain two equivalent views on the resources in the box, either slice-by-slice or the entire box at once, and both views are permitted to take resources out of the box or put them back in.

The Iris technical appendix[46] shows how boxes can be implemented in Iris using higher-order ghost state (§6.1). However, the details of that implementation do not matter for the lifetime logic.

11.6.2 Controlling borrows and inheritances

With boxes at hand, we now have a tool to describe a lifetime that is alive: there is some proposition P_B describing everything borrowed at this lifetime, and the invariant maintains a box with total proposition P_B. We use some ghost state to keep track of the slices of that box, and hand that ghost state out to clients: a borrow $\&^{\kappa}_{\mathsf{full}}\, P$ really means controlling the state of some slice with content P in the box for lifetime κ.

This ghost state, in fact, is what we defined OwnBor for above: *BorSt* (Figure 11.10) describes the possible states of a slice in the borrow box. Either the resources are currently in the invariant (in), or the borrow is open, using fraction q of the lifetime token ($\mathsf{open}(q)$). We use the resource algebra $\mathit{BorBox} := \mathit{Auth}(\mathbb{N} \xrightharpoonup{\mathsf{fin}} \mathit{Ag}(\mathit{BorSt}) \times \mathit{Frac})$ to track this state. The fraction here is used to model the fraction in the borrow token ($[\mathsf{Bor} : i]_q$); this is the same pattern that we saw when modeling fractional points-to assertions for a heap (see §4.4, where we used $\mathit{Ag}(\mathit{Val}) \times \mathit{Frac}$).

After all this preparation, we can finally define borrow tokens and indexed borrows! There is just one last thing: we need to explain what exactly an *index* i is. To actually identify a particular slice of a borrow box, we need two pieces: a lifetime κ, and a slice name ι. Thus, we say that an index is a pair of a lifetime and a slice name: $i = (\kappa, \iota)$.

[45] All of this happens under a suitable amount of later modalities, again as a side-effect of using higher-order ghost state.

[46] Iris Team, "The Iris 3.2 documentation", 2019 [Iri19], §10.3.

Then borrow tokens and indexed borrows are defined as follows:

$$\begin{aligned} [\mathsf{Bor} : (\kappa', \iota)]_q &:= \mathsf{OwnBor}(\kappa', \circ(\iota \leftarrow (\mathsf{in}, q))) \\ \&^{\kappa}_{(\kappa',\iota)}\, P &:= \kappa \sqsubseteq \kappa' * \mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota) \end{aligned}$$

Remember that OwnBor looks up the ghost name γ_{bor} used by lifetime κ' to manage its borrow state, and asserts ownership at that ghost name. In other words, a borrow token is (fractional) ownership of the element ι in the borrow state of lifetime κ', stating that the borrowed resources are currently held in the invariant. And an indexed borrow says that the slice named ι contains the borrowed proposition P (and that the "externally visible" lifetime κ is a sublifetime of the lifetime κ' at which the borrow is actually stored).

We can also define full borrows, introducing an internal helper notion of a *raw borrow*:

$$\begin{aligned} \mathsf{RawBor}(\kappa, P) &:= \exists \iota.\, \mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota) * \mathsf{OwnBor}(\kappa, \circ(\iota \leftarrow (\mathsf{in}, 1))) \\ \&^{\kappa}_{\mathsf{full}}\, P &:= \exists \kappa'.\, \kappa \sqsubseteq \kappa' * \mathsf{RawBor}(\kappa', P) \end{aligned}$$

A raw borrow $\mathsf{RawBor}(\kappa, P)$ says that there exists some slice ι containing P, and that we fully own that slice as part of the borrow box of lifetime κ. A full borrow then quantifies over an arbitrary lifetime bigger than the user-visible κ, similar to what we already saw for fractured borrows—this is needed to make LftL-bor-shorten provable.

This should give a rough idea for how we model and manage the borrows at a lifetime. Before we can look at the per-lifetime invariant, we have to also briefly talk about inheritances. The inheritance $[\dagger\kappa] \mathrel{\Rrightarrow\!\!\ast}_{\mathcal{N}_{\mathsf{lft}}} \triangleright P$ is provided by LftL-borrow upon borrowing P. Similar to and independent of borrows, an inheritance claims ownership of some part of the resources in the borrow. To this end, a lifetime contains a *second* box: besides the borrow box that we already talked about, there is an inheritance box. Its slices are managed by the second bit of per-lifetime ghost state, OwnInh (which looks up the ghost name γ_{inh} used by a lifetime to control its inheritances). The protocol for inheritances is simpler than for borrows: unlike borrows, inheritances are claimed once and can then never be used again. Thus all we need is some per-inheritance token representing ownership of the respective slice in the inheritance box. This is controlled by the RA $\mathit{InhBox} := \mathit{Auth}(\wp^{\mathsf{fin}}(\mathbb{N}))$, an (authoritative) finite set of slice names with disjoint union for its composition.[47]

Initially, the inheritance box will have the same total content P_B as the borrow box. However, remember that LftL-bor-acc-strong lets the client *change* the borrowed proposition, which will under the hood translate to changing (removing and re-adding) the corresponding slice of the borrow box. But we cannot also adjust the inheritance box, as the resources controlling its slices are already backing the inheritance, owned by the client. Instead, we say that the two boxes in a lifetime have total content P_B (the borrow box) and P_I (the inheritance box), and we maintain (very roughly) that $P_B \mathrel{-\!\!*} P_I$.

When the lifetime is alive, the slices of the borrow box are controlled via OwnBor, but the inheritance box is entirely empty. The resources that

[47] This is isomorphic to $\mathit{Auth}(\wp^{\mathsf{fin}}(\mathbb{N}) \overset{\mathsf{fin}}{\rightharpoonup} \mathit{Ex}())$. In other words, each slice is either exclusively owned or not owned at all (not in the map).

will fill the inheritance box are still borrowed! When the lifetime ends, we will prove that all slices of the borrow box are full. That enables us to use Box-empty which empties the entire borrow box, providing ownership of P_B. Then we use the above magic wand to turn P_B into P_I, and finally we use Box-fill to fill the entire inheritance box. Henceforth, the slices of the inheritance box are controlled by OwnInh, while the borrow box is empty.

11.6.3 The per-lifetime invariant

All the pieces are now prepared to take a closer look at the formal definition of the per-lifetime invariant (without support for reborrowing) in Figure 11.12. We also repeat the definition of borrows and the global invariant that we have already introduced. Remember, this is not yet a model of the full lifetime logic; this is the simplified version that does not support reborrowing (*i.e.*, LftL-idx-bor-unnest does not hold).

Let us first look at the simpler case of a dead lifetime. LftDead says that a dead lifetime comes with some proposition P_I describing all the inheritances that have not been claimed yet.

In LftInh, we use the usual pattern of authoritative ghost state (§4.4): we quantify over the current state (in this case, E is a set containing the slice names of the remaining inheritances) and tie the authoritative element (OwnInh) to the state we want to control (the slices of the inheritance box). P_I is set to be the full proposition of the inheritance box, and all its slices (as given by E) are in the same state s—for a dead lifetime, they are all full.

The other part of a dead lifetime is LftBorDead, which controls what is left of the borrow box. One might think that we can just entirely forget about the borrow box when we are done, but that does not work: to support creating "fake" borrows via LftL-bor-fake (amongst other reasons), we need to keep the borrow box around. So we use another instance of the authoritative pattern and quantify over the set B of remaining borrows. In the authoritative state, all of the borrows are currently considered to have their resources in the lifetime, *i.e.*, their state is $(\mathsf{in}, 1)$.[48] However, the full content P_B of the borrow box is entirely irrelevant, and all slices are empty. There are no resources left here, just an empty skeleton of bookkeeping.

The more complicated case is the one of the lifetime still being alive. In LftAlive, we track the two propositions P_B and P_I describing all the borrowed and to-be-inherited resources, respectively. LftBorAlive manages the borrow box, LftVs manages the relationship between the two boxes, and LftInh manages the inheritance box.

We have already seen LftInh; the only difference to before is that now, of course, all its slices are *empty*. Nothing can be inherited while the lifetime is still alive. LftVs is pretty simple at this point, just a view shift that turns all the borrows into all the inheritances under the assumption that the lifetime is dead.

[48] The 1 here arises for the same reason as it did back in §4.4: the authoritative state has to subsume all the fragments combined.

Borrows.

$$\begin{aligned}
[\mathsf{Bor} : (\kappa', \iota)]_q &:= \mathsf{OwnBor}(\kappa', \circ(\iota \leftarrow (\mathsf{in}, q))) \\
\&^{\kappa}_{(\kappa',\iota)}\, P &:= \kappa \sqsubseteq \kappa' * \mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota) \\
\mathsf{RawBor}(\kappa, P) &:= \exists \iota.\ \mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota) * \mathsf{OwnBor}(\kappa, \circ(\iota \leftarrow (\mathsf{in}, 1))) \\
\&^{\kappa}_{\mathsf{full}}\, P &:= \exists \kappa'.\ \kappa \sqsubseteq \kappa' * \mathsf{RawBor}(\kappa', P)
\end{aligned}$$

Invariant.

$$\begin{aligned}
\mathsf{LftInh}(\kappa, P_I, s) &:= \exists E : \wp^{\mathsf{fin}}(\mathbb{N}).\ \mathsf{OwnInh}(\kappa, \bullet E) * \mathsf{Box}(\mathcal{N}_{\mathsf{inh}}, P_I, [\iota \leftarrow s \mid \iota \in E]) \\
\mathsf{LftBorDead}(\kappa) &:= \exists B : \wp^{\mathsf{fin}}(\mathbb{N}), P_B : \mathit{iProp}.\ \mathsf{OwnBor}(\kappa, \bullet[\iota \leftarrow (\mathsf{in}, 1) \mid \iota \in B]) * \\
&\qquad \mathsf{Box}(\mathcal{N}_{\mathsf{bor}}, P_B, [\iota \leftarrow \mathsf{empty} \mid \iota \in \mathsf{dom}(B)]) \\
\mathsf{LftDead}(\kappa) &:= \exists P_I.\ \mathsf{LftBorDead}(\kappa) * \mathsf{LftInh}(\kappa, P_I, \mathsf{full})
\end{aligned}$$

$$\begin{aligned}
\mathsf{bor\_to\_box}(s) &:= \begin{cases} \mathsf{full} & \text{if } s = \mathsf{in} \\ \mathsf{empty} & \text{otherwise} \end{cases} \\
\mathsf{LftBorDeposit}(\kappa, B) &:= \mathop{\Huge *}_{\iota \in \mathsf{dom}(B)} \left\{ \begin{matrix} \mathsf{True} & \text{if } B(\iota) = \mathsf{in} \\ [\kappa]_q & \text{if } B(\iota) = \mathsf{open}(q) \end{matrix} \right\} \\
\mathsf{LftBorAlive}(\kappa, P_B) &:= \exists B : \mathbb{N} \xrightharpoonup{\mathsf{fin}} \mathit{BorSt}.\ \mathsf{OwnBor}(\kappa, \bullet[\iota \leftarrow (B(\iota), 1) \mid \iota \in \mathsf{dom}(B)]) * \\
&\qquad \mathsf{Box}(\mathcal{N}_{\mathsf{bor}}, P_B, \mathsf{bor\_to\_box}\ \langle\$\rangle\ B) * \mathsf{LftBorDeposit}(\kappa, B) \\
\mathsf{LftVs}(\kappa, P_B, P_I) &:= \triangleright P_B * [\dagger\kappa] \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{bor}}} \triangleright P_I \\
\mathsf{LftAlive}(\kappa) &:= \exists P_B, P_I.\ \mathsf{LftBorAlive}(\kappa, P_B) * \mathsf{LftVs}(\kappa, P_B, P_I) * \mathsf{LftInh}(\kappa, P_I, \mathsf{empty})
\end{aligned}$$

$$\begin{aligned}
\mathsf{LftAliveIn}(A, \kappa) &:= \forall \Lambda \in \kappa.\ \exists q.\ A(\Lambda) = \mathsf{inl}(q) \\
\mathsf{LftDeadIn}(A, \kappa) &:= \exists \Lambda \in \kappa.\ A(\Lambda) = \mathsf{inr}() \\
\mathsf{LftInv}(A, \kappa) &:= \mathsf{LftAlive}(\kappa) * \mathsf{LftAliveIn}(A, \kappa) \vee \\
&\qquad \mathsf{LftDead}(\kappa) * \mathsf{LftDeadIn}(A, \kappa) \\
\mathsf{LftLInv} &:= \exists A, I.\ \boxed{\bullet A}^{\gamma_{\mathsf{a}}} * \boxed{\bullet I}^{\gamma_{\mathsf{i}}} * \mathop{\Huge *}_{\kappa \in \mathsf{dom}(I)} \mathsf{LftInv}(A, \kappa) \\
\mathsf{LftLCtx} &:= \boxed{\mathsf{LftInv}}^{\mathcal{N}_{\mathsf{mgmt}}}
\end{aligned}$$

Figure 11.12: Lifetime logic invariant (without reborrowing).

And finally, LftBorAlive is where the main borrow management happens. We quantify over the map B describing the state for each borrow, and make sure that it syncs up with OwnBor via the authoritative element. We also connect B to the state of the borrow box slices; bor_to_box is used to define that borrows in the in state have full slices while all remaining borrows have empty slices (because they are currently being held open by a client). Finally, in LftBorDeposit we claim a "deposit" for every borrow that is currently open in the form of some fraction of the corresponding lifetime token. The deposited amount is determined by the borrow state.

11.6.4 Lifetime logic proofs

Having laid down all the definitions, we now give a high-level intuition for what happens when proving the rules of the lifetime logic.

The rules for handling lifetime tokens (LftL-tok-timeless,LftL-end-persist, LftL-tok-fract, LftL-tok-inter, LftL-end-inter, LftL-tok-unit, LftL-end-unit, LftL-not-own-end) are all trivial based on our choice for the underlying models for lifetime tokens and dead tokens.

Proof sketch for LftL-begin. Let us look at a more interesting rule:

LftL-begin

$$\mathsf{True} \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \exists \kappa.\, [\kappa]_1 * \big([\kappa]_1 \mathrel{\triangleright\!\!\!\Rrightarrow\!\!\ast}_{\mathcal{N}_{\mathsf{lft}}} [\dagger\kappa]\big)$$

This rule allocates a new atomic lifetime. So we start by opening the lifetime invariant. We pick some fresh $\Lambda \notin A$ and add it to the authoritative ghost state, obtaining ownership of its entire lifetime token. When closing the invariant again, we have to show that adding a fresh element to A does not affect $\mathsf{LftAliveIn}(A, \_)$ or $\mathsf{LftDeadIn}(A, \_)$, but that is straightforward. And that is actually all that happens right now—we pick $\kappa := \{\Lambda\}$ as the lifetime returned to the client, and we already own the token for that.

Besides that, we also have to prove the view shift that takes a step to end the lifetime: $[\{\Lambda\}]_1 \mathrel{\triangleright\!\!\!\Rrightarrow\!\!\ast}_{\mathcal{N}_{\mathsf{lft}}} [\dagger\,\{\Lambda\}]$. In the full lifetime logic with reborrowing, this will be the most complicated part of the entire proof; for now, it is not quite so complicated but still nontrivial. After opening the lifetime logic invariant, the first thing we do is to exploit that we are proving a view shift *that takes a step*, so we can remove a $\triangleright$ from our assumption. Without this step, we would have no way to work with all the non-timeless resources in $\mathsf{LftLInv}$. It remains to show the following:

$$\mathsf{LftLInv} * [\{\Lambda\}]_1 \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}} \mathsf{LftLInv} * [\dagger\,\{\Lambda\}]$$

We switch the status of Λ in A to $\mathsf{inr}()$ which we can do because we own the full token, obtaining $[\dagger\,\{\Lambda\}]$. Next, we also have to actually end all the lifetimes κ that were obtained by intersecting Λ with some other lifetimes. In other words, we have to go over all lifetimes κ registered in I, and if any of them is currently still alive and $\Lambda \in \kappa$, then we have to switch the state of κ from "alive" to "dead":

$$\mathsf{LftAlive}(\kappa) * [\dagger\,\{\Lambda\}] * \Lambda \in \kappa \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}} \mathsf{LftDead}(\kappa) \qquad (\text{Lft-end1})$$

($\mathsf{LftDeadIn}$ is trivial because we just switched Λ to be dead in A.)

The first step is to empty the borrow box, which is expressed by the following helper lemma:

$$\mathsf{LftBorAlive}(\kappa, P_B) * [\dagger\,\{\Lambda\}] * \Lambda \in \kappa \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}} \mathsf{LftBorDead}(\kappa) * \triangleright P_B$$

This is done using Box-empty, so we have to prove that every slice of the borrow box is currently filled. Looking at $\mathsf{LftBorAlive}$, we learn the current borrow box state B. It is sufficient to show that $B(\iota) = \mathsf{in}$ for all ι. But we know this to be the case because if we had $B(\iota) = \mathsf{open}(q)$ for any ι, then by the definition of $\mathsf{LftBorDeposit}$ we would own $[\kappa]_q$. But we also own $[\dagger\{\Lambda\}]$, and since $\Lambda \in \kappa$, this implies $[\dagger\kappa]$, so we have a contradiction by LftL-not-own-end. It follows that all slices of the borrow box are full, and using Box-empty we obtain $\triangleright P_B$.[49] We also obtain whatever is left of $\mathsf{LftBorAlive}$, namely the authoritative OwnBor for κ as well as ownership of the emptied borrow box; this fits exactly to prove $\mathsf{LftBorDead}$.

The next step is easy: using LftVs, we turn $\triangleright P_B$ into $\triangleright P_I$. This requires $[\dagger\kappa]$, which we already established we can show at this point (having ended $\Lambda \in \kappa$).[50] We are free to eliminate the update modality and obtain $\triangleright P_I$.

The third and final step is to fill the inheritance box:

$$\mathsf{LftInh}(\kappa, P_I, \mathsf{empty}) * \triangleright P_I \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}} \mathsf{LftInh}(\kappa, P_I, \mathsf{full})$$

This is a fairly straightforward application of Box-fill, because we already know that all slices of the inheritance box are initially empty.[51]

This completes the proof of Lft-end1 and thus of LftL-begin.

Proof sketch for LftL-borrow. We have seen how borrowed resources turn into inheritances when the lifetime ends. Let us now consider how borrows are created in the first place, and how inheritances yield resources back to clients:

LftL-borrow
$$\triangleright P \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa}_{\mathsf{full}} P * \big([\dagger\kappa] \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}}} \triangleright P\big)$$

As usual, first we will open the lifetime logic invariant. Now we need to get hold of $\mathsf{LftInv}(A, \kappa)$, but what if κ is not yet in I? New lifetimes can be created via intersection by the client any time, so there is no guarantee that the lifetime logic has seen κ before. If it does not exist yet, we will have to create it.[52]

Initializing a lifetime with no borrows is a bit tedious but not very interesting, we just have to create lots of ghost state. We create an empty borrow box and an empty inheritance box, with $P_B := \mathsf{True}$ and $P_I := \mathsf{True}$. Both B and E (the state managing the borrow and inheritance box, respectively) are initially empty. We also exploit the fact that every lifetime κ is either dead or alive, or contains some unmanaged atomic lifetime:

$$\forall A, \kappa.\ \mathsf{LftAliveIn}(A, \kappa) \lor \mathsf{LftDeadIn}(A, \kappa) \lor \big(\exists \Lambda \in \kappa.\ \Lambda \notin \mathsf{dom}(A)\big)$$

In the first case, we can easily show $\mathsf{LftAlive}(\kappa)$ and be done. Similarly, in the second case we can show $\mathsf{LftDead}(\kappa)$. In the last, we can add that Λ to A as dead, and then prove $\mathsf{LftDeadIn}(\kappa)$.

At this point we can assume that $\kappa \in \mathsf{dom}(I)$. We also observe that $\mathsf{RawBor}(\kappa, P) \twoheadrightarrow\!\!* \&^{\kappa}_{\mathsf{full}} P$, so it suffices to show:

$$\begin{aligned}&\triangleright \mathsf{LftInv}(\kappa) * \triangleright P \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}}\\ &\qquad \triangleright \mathsf{LftInv}(\kappa) * \mathsf{RawBor}(\kappa, P) * \big([\dagger\kappa] \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}}} \triangleright P\big) \qquad (\text{Lft-bor})\end{aligned}$$

First, we perform case distinction on whether κ is alive or not.

[49] This rule requires mask $\mathcal{N}_{\mathsf{bor}}$. Our current mask is $\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}$ and $\mathcal{N}_{\mathsf{bor}}$ is picked to be contained in $\mathcal{N}_{\mathsf{lft}}$ but disjoint from $\mathcal{N}_{\mathsf{mgmt}}$, so we are good to go.

[50] Again we need mask $\mathcal{N}_{\mathsf{bor}}$, which works out for the same reason as before.

[51] We need mask $\mathcal{N}_{\mathsf{inh}}$ for this step, but $\mathcal{N}_{\mathsf{inh}}$ is disjoint from $\mathcal{N}_{\mathsf{mgmt}}$ like $\mathcal{N}_{\mathsf{bor}}$, so the same argument as before applies.

[52] Note that it might also be the case that κ has already ended; we will get to that point later.

If it is dead, we "fake" a borrow while keeping $\triangleright P$ to ourselves:

$$\triangleright \mathsf{LftBorDead}(\kappa) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}} \triangleright \mathsf{LftBorDead}(\kappa) * \mathsf{RawBor}(\kappa, P) \qquad (\textsc{Lft-fake})$$

To prove "faking", we exploit the fact that LftBorDead owns the borrow box with all slices being empty, so we apply Slice-insert-empty to add a new one, and extend the authoritative state and B appropriately to create the corresponding borrow token. That means we still own $\triangleright P$ when we prove the inheritance, which becomes trivial.

The interesting case of Lft-bor is when κ is still alive. The first helper lemma we show is for adding a slice to the borrow box:

$$\begin{aligned}&\triangleright \mathsf{LftBorAlive}(\kappa, P_B) * \triangleright P \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}} \\ &\qquad \triangleright \mathsf{LftBorAlive}(\kappa, P_B * P) * \mathsf{RawBor}(\kappa, P) \qquad (\textsc{Lft-bor'})\end{aligned}$$

To this end, we use Slice-insert-full.[53] We have to give up $\triangleright P$ but obtain some new $\iota \notin \mathsf{dom}(B)$, ownership of the slice $\mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota)$, and the changed box with new total content $P_B * P$. To re-establish LftBorAlive, we need to get the box state back in sync with the ghost state again, so we need to adjust B (as usual for the authoritative pattern). The new B will be $B[\iota \leftarrow \mathsf{in}]$. Since ι is fresh, we can do a frame-preserving update to add it to the authoritative state that we own in OwnBor, which also creates the fragment $\mathsf{OwnBor}(\kappa, \circ(\iota \leftarrow (\mathsf{in}, 1)))$ for us to keep (as per fmap-alloc-local on page 61). Changing B also requires us to re-prove LftBorDeposit, but since the new borrow we added has its resources in the lifetime invariant (borrow state in), nothing changes there. This means we can prove $\triangleright \mathsf{LftBorAlive}(\kappa, P_B * P)$, and the resources we have left are $\mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota) * \mathsf{OwnBor}(\kappa, \circ(\iota \leftarrow (\mathsf{in}, 1)))$, which is exactly what we need for $\mathsf{RawBor}(\kappa, P)$. Thus, this first step (Lft-bor') is done.

The second lemma we need lets us add a new inheritance:

$$\begin{aligned}\triangleright \mathsf{LftInh}(\kappa, P_I, \mathsf{empty}) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}} &\triangleright \mathsf{LftInh}(\kappa, P_I * P, \mathsf{empty}) * \\ &\exists \iota.\, \mathsf{OwnInh}(\kappa, \circ\{\iota\}) * \mathsf{BoxSlice}(\mathcal{N}_{\mathsf{inh}}, P, \iota)\end{aligned}$$

This time we use Slice-insert-empty[54] to create a new slice $\iota \notin E$ in the inheritance box. We pick $E \cup \{\iota\}$ as the new E, so we have to sync the authoritative ghost state with the box again via a frame-preserving update. As usual, this also allocates a fragment that we will own: $\mathsf{OwnInh}(\kappa, \circ\{\iota\})$. With that, the proof of the lemma is complete.

Since we changed both P_B and P_I, we also need to re-prove LftVs. However, showing the following is easy:

$$\triangleright \mathsf{LftVs}(\kappa, P_B, P_I) \mathrel{-\!\!*} \triangleright \mathsf{LftVs}(\kappa, P_B * P, P_I * P)$$

We can remove the $\triangleright$ on both sides using $\triangleright$-mono. Inside LftVs, the new resources P can simply be framed around the view shift that we already have.

With all these lemmas, we can re-establish $\triangleright \mathsf{LftInv}(\kappa)$, and we have already produced $\mathsf{RawBor}(\kappa, P)$. All that is left in showing Lft-bor is proving that we can actually take P out again when the lifetime is dead.

[53] Notice that we own the box itself only under a $\triangleright$, but the rule says that is sufficient (though of course the changed box we get back will then also be under a $\triangleright$). We heavily exploit that $\triangleright$ commutes with separating conjunction as well as existential quantifiers with non-empty domain.

[54] Again with the box owned under a $\triangleright$.

We have some resources left over from creating the inheritance slice in the second lemma, leading to the following remaining proof obligation:

$$\mathsf{OwnInh}(\kappa, \circ\{\iota\}) * \mathsf{BoxSlice}(\mathcal{N}_{\mathsf{inh}}, P, \iota) * [\dagger\kappa] \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \triangleright P$$

Again we start by opening the lifetime logic invariant. Since we own a token for κ, we know that $\kappa \in \mathsf{dom}(I)$. Moreover, we can show that the lifetime cannot be alive any more: $\boxed{\bullet A}^{\gamma_{\mathsf{a}}} * [\dagger\kappa] * \mathsf{LftAliveIn}(A, \kappa) -\!\!\ast \mathsf{False}$. Thus it suffices to show:

$$\begin{aligned}&\triangleright \mathsf{LftInh}(\kappa, P_I, \mathsf{full}) * \mathsf{OwnInh}(\kappa, \circ\{\iota\}) * \mathsf{BoxSlice}(\mathcal{N}_{\mathsf{inh}}, P, \iota) \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}} \\ &\qquad \exists P_I'.\ \triangleright \mathsf{LftInh}(\kappa, P_I', \mathsf{full}) * \triangleright P\end{aligned}$$

We unfold $\triangleright \mathsf{LftInh}(\kappa, P_I, \mathsf{full})$ to obtain authoritative ownership of the inheritance slice management, and ownership of the inheritance box. From our fragment of OwnInh combined with the authoritative element, we learn that $\iota \in E$ (otherwise $\bullet E \cdot \circ\{\iota\}$ would be invalid). Thus ι is a slice in the inheritance box, and it is currently full. We also have the matching $\mathsf{BoxSlice}$, so we can use Slice-delete-full to remove that slice, obtaining a new box (with a changed total resource, but P_I is otherwise unconstrained so we can change it easily[55]). That rule also provides the content of the slice to us, which we know to be $\triangleright P$. To re-establish LftInh, we need to adjust the authoritative OwnInh to remove ι from E, which we can do because we own the relevant fragment. Then we are done.

Proof sketch for LftL-bor-acc-strong. As the third and final proof rule that we will look at in detail, we consider LftL-bor-acc-strong. Or rather, we show the following variant for raw borrows:

$$\begin{aligned}&\mathsf{RawBor}(\kappa, P) * [\kappa]_q \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \triangleright P * \\ &\qquad \Big(\forall Q.\ \triangleright(\triangleright Q * [\dagger\kappa] \Rrightarrow\!\!\!\ast_{\emptyset} \triangleright P) * \triangleright Q \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \mathsf{RawBor}(\kappa, Q) * [\kappa]_q\Big) \\ &\hfill (\text{LftL-raw-bor-acc})\end{aligned}$$

Deriving LftL-bor-acc-strong from this is just a matter of unfolding the definition of full borrows.

We decompose this rule into two parts: opening the borrow, and closing it again. The first part is expressed by the following helper lemma, where we also unfold RawBor:[56]

$$\begin{aligned}&\mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota) * \mathsf{OwnBor}(\kappa, \circ(\iota \leftarrow (\mathsf{in}, 1))) * [\kappa]_q \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \\ &\qquad\qquad \mathsf{OwnBor}(\kappa, \circ(\iota \leftarrow (\mathsf{open}(q), 1))) * \triangleright P \\ &\hfill (\text{LftL-raw-bor-open})\end{aligned}$$

The key difference between the left-hand side and the right-hand side is that we lose $[\kappa]_q$ and gain $\triangleright P$, and the borrow state (that we track via OwnBor, *i.e.*, via the borrow token) changes from in to $\mathsf{open}(q)$ to record the fraction of the lifetime token that we lost.

We begin again by opening the lifetime logic invariant, and since we own a token for κ we know that $\kappa \in \mathsf{dom}(I)$. Thus we obtain $\triangleright \mathsf{LftInv}(\kappa)$. We also know that the lifetime has to be alive, because otherwise we obtain a contradiction: $\boxed{\bullet A}^{\gamma_{\mathsf{a}}} * [\kappa]_q * \mathsf{LftDeadIn}(A, \kappa) -\!\!\ast \mathsf{False}$. The interesting

[55] While the lifetime is active, we have to keep careful track of the total resources in both the borrow and inheritance box so that we can use the content of the former to fill the latter when the lifetime ends. But once the lifetime is over, the lifetime invariant does not care any more about the content of the boxes, all it does is bookkeeping to track the slices they contain.

[56] Remember that $\mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota)$ is persistent, hence we do not repeat it in the conclusion.

part of the lifetime invariant is $\mathsf{LftBorAlive}$, about which it suffices to show the following lemma:

$$\begin{aligned}&\triangleright \mathsf{LftBorAlive}(\kappa, P_B) *\\ &\mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota) * \mathsf{OwnBor}(\kappa, \circ(\iota \leftarrow (\mathsf{in}, 1))) * [\kappa]_q \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}}\\ &\qquad\qquad \triangleright \mathsf{LftBorAlive}(\kappa, P_B) * \mathsf{OwnBor}(\kappa, \circ(\iota \leftarrow (\mathsf{open}(q), 1))) * \triangleright P\end{aligned}$$

After unfolding $\mathsf{LftBorAlive}$, we combine our OwnBor with the authoritative element and learn that $B(\iota) = \mathsf{in}$.[57] Thus the slice ι is part of the borrow box and currently full. We use Slice-empty to take out the slice resources, which we know to be $\triangleright P$. Now we need to sync the box state with the authoritative state again, to which end we change $B(\iota)$ to $\mathsf{open}(q)$. We can do that because we own the authoritative element. But we also need to show $\mathsf{LftBorDeposit}$, which requires us to give up $[\kappa]_q$ matching the new borrow state of ι. This completes the lemma, and thus we are done with the opening phase of accessing the borrow (LftL-raw-bor-open).

For the closing part, we need the following lemma:

$$\begin{aligned}&\triangleright (\triangleright Q * [\dagger\kappa] \Rrightarrow\!\!\!*_{\emptyset} \triangleright P) * \triangleright Q *\\ &\mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota) * \mathsf{OwnBor}(\kappa, \circ(\iota \leftarrow (\mathsf{open}(q), 1))) \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}}}\\ &\qquad \exists \iota'.\, \mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, Q, \iota') * \mathsf{OwnBor}(\kappa, \circ(\iota' \leftarrow (\mathsf{in}, 1))) * [\kappa]_q\end{aligned}$$

(LftL-raw-bor-close)

Note that the conclusion of this rule is the same as the result of the last view shift in LftL-raw-bor-acc, we just unfolded the raw borrow. We need a new ι' because the content of the borrow might change, so we cannot reuse the old slice.

We open the lifetime logic invariant, and since we have an OwnBor for κ we know that $\kappa \in \mathsf{dom}(I)$ so we can access $\triangleright \mathsf{LftInv}(\kappa)$. We need to again argue that the lifetime is still alive, but the argument is a bit more subtle this time: if the lifetime was dead, we would get $\triangleright \mathsf{LftBorDead}(\kappa)$, which authoritatively says that the state of all borrows is in. But we have a borrow in a different state ($\mathsf{open}(q)$), so we have a contradiction.[58]

Since the lifetime is alive, we can access $\triangleright \mathsf{LftAlive}(\kappa)$. The first step in closing the borrow is to change the state of the borrow slice back to in, which is done via the following lemma:

$$\begin{aligned}&\triangleright \mathsf{LftBorAlive}(P_B) * \triangleright Q *\\ &\mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota) * \mathsf{OwnBor}(\kappa, \circ(\iota \leftarrow (\mathsf{open}(q), 1))) \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}}\\ &\qquad \exists P'_B.\, \triangleright \mathsf{LftBorAlive}(P'_B * Q) * \triangleright\triangleright(P_B = (P'_B * P)) *\\ &\qquad \exists \iota'.\, \mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, Q, \iota') * \mathsf{OwnBor}(\kappa, \circ(\iota' \leftarrow (\mathsf{in}, 1))) * [\kappa]_q\end{aligned}$$

As usual when applying the authoritative pattern, we combine our own fragment (OwnBor) with the authoritative state to learn that $B(\iota) = \mathsf{open}(q)$. This tells us that ι is an empty slice in the borrow box, which we already know to have content $\triangleright P$. We use Slice-delete-empty to get rid of that slice. In so doing, the total content of the borrow box changes to some P'_B such that $\triangleright\triangleright(P_B = (P'_B * P))$.[59] Next we use Slice-insert-full to add a new slice with content Q; let ι' be the name of that new slice. The content of the borrow box is now $P'_B * Q$, so we

[57] The reasoning here is very similar to fheap-auth-ag0 (page 57).

[58] This is the other reason (besides “faking” borrows) for why we need $\mathsf{LftBorDead}$.

[59] The nested laters may look daunting, but that is just a distraction.

can see how things are shaping up towards the conclusion of our lemma. What is left to do is to sync up the authoritative state with the borrow box again: we remove ι and add ι' in B, which consumes ownership of OwnBor for ι and produces $\mathsf{OwnBor}(\kappa, \circ(\iota' \leftarrow (\mathsf{in}, 1)))$. We also need to re-prove $\mathsf{LftBorDeposit}$, and since we removed a borrow with state $\mathsf{open}(q)$ from B, that means we gain ownership of fraction q of the lifetime token. (The new borrow ι' is in state in, so there is nothing to deposit there.) This concludes the first step of LftL-raw-bor-close.

For the second step, we need to adjust LftVs because the first step changed P_B. This is done via the following lemma:

$$\begin{aligned}&\triangleright \mathsf{LftVs}(\kappa, P_B, P_I) * \triangleright\triangleright(P_B = (P'_B * P)) * \triangleright(\triangleright Q * [\dagger\kappa] \Rrightarrow\!\!\ast_{\emptyset} \triangleright P) \mathrel{-\!\!*} \\ &\qquad \triangleright \mathsf{LftVs}(\kappa, P'_B * Q, P_I)\end{aligned}$$

Intuitively, this makes sense because all we do is remove a P from P_B and replace it with Q, and since we also know that we can turn Q back into P, it should be clear that we can turn $P'_B * Q$ back into P_B and then continue with the view shift to P_I as before. Actually getting there, however, requires shuffling around a lot of symbols.

Note that each assumption and the conclusion starts with a $\triangleright$, so we can remove one level of $\triangleright$ anywhere. After unfolding LftVs we see that it is sufficient to show:[60]

$$\begin{aligned}&(\triangleright P_B * [\dagger\kappa] \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{bor}}} \triangleright P_I) * \triangleright(P_B = (P'_B * P)) * (\triangleright Q * [\dagger\kappa] \Rrightarrow\!\!\ast_{\emptyset} \triangleright P) \mathrel{-\!\!*} \\ &\qquad \triangleright(P'_B * Q) * [\dagger\kappa] \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{bor}}} \\ &\qquad \triangleright P_I\end{aligned}$$

There are a lot of resources and magic wands flying around here that we have to chain in the right way. The first step is to take the $\triangleright Q$ that we own (remember that $\triangleright$ commutes with separating conjunction), together with the (persistent) $[\dagger\kappa]$, and turn it into $\triangleright P$ using the corresponding view shift (the one with mask $\emptyset$). We also apply the first view shift, which says that to prove $\triangleright P_I$ (under an update modality) it is sufficient to prove $\triangleright P_B * [\dagger\kappa]$. The remaining statement is already much cleaner:

$$\triangleright(P_B = (P'_B * P)) * \triangleright P'_B * \triangleright P * [\dagger\kappa] \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{bor}}} \triangleright P_B * [\dagger\kappa]$$

We remove the update modality from the goal. The second separating conjunct is easily proven as we own $[\dagger\kappa]$. Now the goal again starts with a $\triangleright$, so we may remove another level of $\triangleright$ from all assumptions, which leaves us with:

$$(P_B = (P'_B * P)) * P'_B * P \mathrel{-\!\!*} P_B$$

After rewriting with the equality, this goal is trivial. This concludes the proof of LftL-raw-bor-close and thus of LftL-raw-bor-acc.

Remaining proof rules for full borrows. The remaining proof rules for full borrows do not use anything we have not seen yet. LftL-bor-fake is an easy consequence of Lft-fake that we already used to prove LftL-borrow. LftL-bor-split adjusts the borrow box using Slice-split (of course, it also has to update the authoritative state managing B,

[60] Magic wand curries much like implication: $P \mathrel{-\!\!*} Q \mathrel{-\!\!*} R$ is equivalent to $P * Q \mathrel{-\!\!*} R$. In other words, everything except for the final $\triangleright P_I$ is an assumption in this statement, and $\models\!\!\!\Rrightarrow_{\mathcal{N}_{\mathsf{bor}}} \triangleright P_I$ is the goal.

but it owns all the tokens required to do so).[61] LftL-bor-acc-atomic-strong is similar to LftL-bor-acc-strong, except that the lifetime logic invariant is never closed between opening and closing the borrow, so there is no need to ever leave a token as a deposit. LftL-bor-shorten is trivial, given how full borrows are defined.

Proof rules for indexed borrows. Most proof rules for indexed borrows are easy to verify: LftL-bor-idx, LftL-idx-persist, LftL-idx-timeless, LftL-idx-fract, LftL-idx-shorten are all direct consequences of how indexed borrows and borrow tokens are defined. LftL-idx-acc is similar to LftL-bor-acc-strong, except that on the closing side we use Slice-fill to re-fill the existing slice (instead of deleting that slice and adding a different one). Likewise, LftL-idx-acc-atomic is a simpler variant of LftL-bor-acc-atomic-strong.

The key remaining primitive proof rule is LftL-idx-bor-unnest, and it simply does not hold in the model defined in Figure 11.12. To obtain that rule, we need to make some fundamental changes.

11.7 Implementing the full lifetime logic

This completes the model of the restricted lifetime logic, without reborrowing. To verify soundness of the full logic, we need to adjust the model such that it also validates the following proof rule:[62]

LftL-idx-bor-unnest

$$\&^{\kappa_1}_{(\kappa'_1,\iota)}\, P * \&^{\kappa_2}_{\mathsf{full}}([\mathsf{Bor} \,:\, (\kappa'_1,\iota)]_1) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa_1 \sqcap \kappa_2}_{\mathsf{full}}\, P$$

We also have to explain how we can obtain LftL-bor-merge and LftL-idx-iff, but that will turn out to be fairly straightforward.

Proving this rule will require us to add a third state to our management of the borrow box: a borrow cannot be just in (have its resources in the lifetime invariant) or $\mathsf{open}(q)$ (have its resources handed out to the client, with fraction q of the lifetime token left behind as a deposit), but now it can also be in state $\mathsf{rebor}(\kappa)$: reborrowed to lifetime κ. This state is similar to open in the sense that the corresponding slice of the borrow box is currently empty, but the resources are not owned by the client, either—they are owned by *the other lifetime κ*. To make sure that resources are given back in time will require the introduction of a whole new system of deposits, and we will also heavily exploit the fact that we defined lifetimes to be finite multisets of atomic lifetimes.

However, before we discuss in more detail how to adjust the model to support reborrowing, we first reduce the problem to proving a lower-level rule that talks directly about raw borrows and the multiset structure of lifetimes.

11.7.1 Reducing the problem

As we have already seen, LftL-idx-bor-unnest is sufficient to derive LftL-reborrow (see Figure 11.9) and LftL-bor-unnest. This rule can in turn be reduced further, as indicated in Figure 11.13 where we

[61] As already mentioned, LftL-bor-merge does not hold in the model as we presented it so far. The reason is that full borrows internally quantify over some larger lifetime: we can easily show merging for RawBor, but that does not entail merging for full borrows as the lifetimes of the two underlying raw borrows might differ.

[62] We have unfolded the borrow indices into pairs of lifetimes and slice names.

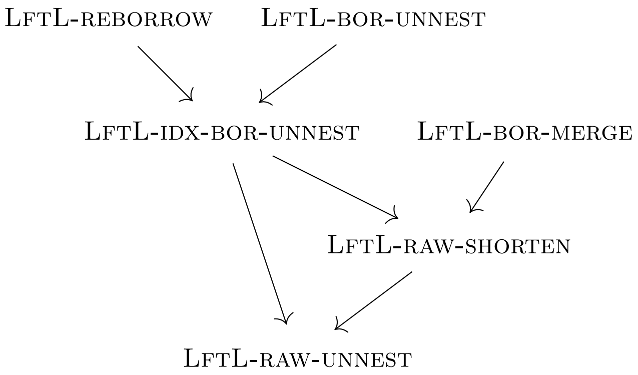

Figure 11.13: Reborrowing proof rules (arrows indicate dependencies)

show a dependency graph of reborrowing rules. At the bottom of this graph is the following lemma about raw borrows:

$$\kappa \subset \kappa' * \mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota) * \mathsf{RawBor}(\kappa', [\mathsf{Bor} : (\kappa, \iota)]_1) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \mathsf{RawBor}(\kappa', P) \qquad (\textsc{LftL-raw-unnest})$$

Here we use $\subset$ on lifetimes as multisets, *i.e.*, we demand that κ as a set of intersected atomic lifetimes is a strict subset of κ'. This means that κ is actually the *longer* lifetime as fewer atomic lifetimes are intersected in it. In other words, this lemma lets us do unnesting if we borrowed the token for P's borrow at lifetime κ at some lifetime κ' that is strictly shorter in a syntactic way: not only do we have $\kappa' \sqsubseteq \kappa$, but the underlying multiset actually directly reflects that κ' is an intersection of strictly more lifetimes.[63] This is useful because lifetime inclusion is a very *extensional* notion, and we have seen how creating a fractured borrow of one lifetime token at another lifetime can create new inclusion links between lifetimes. In contrast, syntactic inclusion of multisets is much more "static" and intensional, and we are going to heavily exploit that in the proof (namely, we are going to do induction on the size of these multisets).

Before we can show LftL-idx-bor-unnest from LftL-raw-unnest, we first need to derive a helper lemma that lets us shorten raw borrows:

$$\kappa \subseteq \kappa' * \mathsf{RawBor}(\kappa, P) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \mathsf{RawBor}(\kappa', P) \qquad (\textsc{LftL-raw-shorten})$$

Unlike rules like LftL-bor-shorten, this rule demands that κ' is *syntactically* (as a multiset) "shorter" than κ (larger sets intersect more atomic lifetimes and are hence shorter).

To prove LftL-raw-shorten, we first do case distinction on whether $\kappa = \kappa'$ or not: if they are equal, we are already done. If they are not equal, we unfold RawBor to learn the name ι of the borrow slice for this borrow. We create a new raw borrow of the borrow token for that slice ($[\mathsf{Bor} : (\kappa, \iota)]_1$) at lifetime κ' (and throw away the inheritance as we do not need it). At this point, our remaining goal is:

$$\kappa \subset \kappa' * \mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota) * \mathsf{RawBor}(\kappa', [\mathsf{Bor} : (\kappa, \iota)]_1) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \mathsf{RawBor}(\kappa', P)$$

But this is *exactly* LftL-raw-unnest, so we are done.

Now we are ready to prove LftL-idx-bor-unnest. We start by unfolding the indexed borrow and full borrow. The full borrow hides some

[63] Since we are dealing with multisets, it could also be the case that κ' just contains more intersections of the same atomic lifetimes that already make up κ. But that is fine, all we really need is a well-founded order on lifetimes.

lifetime κ_2' at which the token is actually borrowed. This turns our goal into:

$$\begin{aligned}&\kappa_1 \sqsubseteq \kappa_1' * \mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota) * \\ &\kappa_2 \sqsubseteq \kappa_2' * \mathsf{RawBor}(\kappa_2', [\mathsf{Bor} : (\kappa_1', \iota)]_1) \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}}} \\ &\quad \&_{\mathsf{full}}^{\kappa_1 \sqcap \kappa_2} P\end{aligned}$$

We need to separately handle the special case that κ_2' is the static lifetime ε (internally represented as $\emptyset$). Borrowing something for the static lifetime is basically the same as not borrowing it at all, which makes the proof easy to complete: we can conjure a token (LftL-tok-unit) and use that to open the borrow.[64] Thus we obtain ownership of $[\mathsf{Bor} : (\kappa_1', \iota)]_1$. Since we also own $\mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota)$, we can assemble the two to obtain $\mathsf{RawBor}(\kappa_1', P)$. Now completing this case of the proof is a matter of showing $\kappa_1 \sqcap \kappa_2 \sqsubseteq \kappa_1'$, which is trivial as we have $\kappa_1 \sqsubseteq \kappa_1'$.

So we are left with the interesting case, where κ_2' is a non-empty set of atomic lifetimes. First, we use LftL-raw-shorten to shorten the lifetime of our raw borrow to $\kappa_1' \sqcap \kappa_2'$. Since lifetime intersection is defined as multiset union, clearly we have $\kappa_2' \subseteq \kappa_1' \sqcap \kappa_2'$, so shortening is applicable. The next step is to apply LftL-raw-unnest with $\kappa := \kappa_1'$ and $\kappa' := \kappa_1' \sqcap \kappa_2'$. We know that κ_2' is a non-empty set of atomic lifetimes, so the strict inclusion in $\kappa \subset \kappa'$ is satisfied. At this point, our remaining goal is:

$$\kappa_1 \sqsubseteq \kappa_1' * \kappa_2 \sqsubseteq \kappa_2' * \mathsf{RawBor}(\kappa_1' \sqcap \kappa_2', P) \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}}} \&_{\mathsf{full}}^{\kappa_1 \sqcap \kappa_2} P$$

This goal is easy to show via LftL-bor-shorten, as lifetime intersection is covariant with respect to lifetime inclusion. Hence we are done showing LftL-idx-bor-unnest.

As mentioned before, we also need reborrowing to derive LftL-bor-merge:

$$\&_{\mathsf{full}}^{\kappa} P * \&_{\mathsf{full}}^{\kappa} Q \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}}} \&_{\mathsf{full}}^{\kappa} (P * Q)$$

Concretely, we will unfold both borrows to obtain raw borrows at potentially different lifetimes κ' and κ''. Then we apply LftL-raw-shorten to reborrow both of them at $\kappa \sqcap \kappa' \sqcap \kappa''$. The rest of the proof is straightforward but tedious: now that we have P and Q borrowed at the same lifetime, we can get access to the corresponding borrow slices and merge them using Slice-merge.[65]

We have now reduced all reborrowing and unnesting to LftL-raw-unnest. A key advantage of this reduction is that in the following, we no longer have to consider lifetime inclusion: the only lifetime relationship left in LftL-raw-unnest is strict inclusion of the underlying multisets. As we have seen, that is sufficient to derive all the higher-level rules that are written in terms of lifetime inclusion.

11.7.2 Adjusting the model

In Figure 11.14, we show the full model of the lifetime logic with support for reborrowing. Tokens and lifetime inclusion did not change at all, so

[64] We will actually never close this borrow again. Borrows at the static lifetime ε are basically equivalent to owning the resources directly (under a $\triangleright$) because tokens for ε can always be created freely, and recovering our lifetime token is the only reason for us to ever close a borrow again (if we used a non-atomic accessor).

[65] The lifetime might in fact be already dead, in which case we can use LftL-bor-fake.

we are not repeating them. We are also skipping the definition of the lifetime logic invariant in terms of LftAlive and LftDead, which likewise is unaffected by reborrows. We have highlighted the parts that changed relative to Figure 11.10 and Figure 11.12 from the previous section with an exclamation mark and blue color.

To support LftL-idx-iff, all it takes is to explicitly close indexed borrows and raw borrows under equivalence of propositions. This can get a bit tedious in the proofs, but does not impose any difficulties. In the following, we will ignore this equivalence to focus on the more challenging aspects of the lifetime logic.

As part of the reborrowing support, we added a third piece of per-lifetime ghost state: a *counter* with RA $Auth(\mathbb{N})$[66] that keeps track of how many outstanding reborrows there are in a lifetime, *i.e.*, how many borrow slices of this lifetime were created by reborrowing from another, longer lifetime. We add a short-hand OwnCnt for this, similar to OwnBor and OwnInh.

In LftBorDeposit, we can see that the fragments of this counter are owned by the lifetimes that resources have been reborrowed from. Reborrowing is only possible if the reborrowed-to lifetime κ' is a strictly bigger multiset, *i.e.*, a strictly smaller lifetime.

The authoritative part is owned by the lifetime itself. A dead lifetime does not hold any reborrows from other lifetimes (see LftDead), but a lifetime that is still alive can have any number of reborrows (see LftAlive). This is handled by LftVs, which is where most of the new complexity arises. Intuitively, what we want to say here is that there is some number n of reborrows that we got from other lifetimes, which means there will be n tokens owned by the LftBorDeposit of all these other lifetimes. Executing the view shift (which is run when a lifetime ends) will un-do all this reborrowing, putting all the resources back into the lifetimes they came from and obtaining ownership of these n tokens. But to be able to do that, the view shift needs to have *access to the resources of these other lifetimes*, which is where things get complicated.

First, a note on syntax: we use $P \mathrel{\Rrightarrow\!\!\ast}[R]_{\mathcal{N}}\, Q$ as syntactic sugar for $P * R \mathrel{\Rrightarrow\!\!\ast}_{\mathcal{N}} Q * R$, *i.e.*, R is a frame that is available to the view shift but is not affected by it. The new definition of LftVs is mutually recursive with LftAlive, but this is well-defined because the multiset representing the current lifetime gets strictly smaller, and all our lifetimes are *finite* multisets. Only having access to strictly smaller lifetimes (when viewed as multisets) is sufficient in LftVs as reborrowing can only happen from a lifetime to a strictly bigger lifetime—so everything we reborrow from (all the resources LftVs needs to put back to be able to collect all the reborrow-counting tokens) will be in strictly smaller lifetimes. This is exactly why we went through the effort of reducing LftL-idx-bor-unnest to LftL-raw-unnest earlier in this section.

To understand why LftVs contains all the pieces it does, it is best to look at the proof of LftL-raw-unnest. Afterwards, we will have to re-consider LftL-begin: that rule also proves how to end a lifetime, and ending a lifetime just got a *lot* more complicated through these changes

[66] $\mathbb{N}$ forms an RA with addition for composition, and no invalid elements.

Domains (only changed ones).

$$(!)\quad I \in \mathit{ILft} := \mathit{Auth}(\mathit{Lft} \overset{\text{fin}}{\rightharpoonup} \mathit{Ag}(\mathbb{N} \times \mathbb{N} \times \mathbb{N})) \qquad \text{(global ghost state to manage lifetimes; name: } \gamma_{\mathsf{i}}\text{)}$$

$$(!)\quad \mathit{BorSt} := \mathsf{in} \mid \mathsf{open}(q : \mathit{Frac}) \mid \mathsf{rebor}(\kappa : \mathit{Lft}) \qquad \text{(state of a borrow box)}$$

$$(!)\quad \mathit{Cnt} := \mathit{Auth}(\mathbb{N}) \qquad \text{(per-lifetime ghost state counting reborrows; } \gamma_{\mathsf{cnt}}\text{)}$$

Helper functions.

$$\begin{aligned}
\mathsf{OwnBor}(\kappa, a) &:= \exists \gamma_{\mathsf{bor}}.\, \boxed{\circ\,[\kappa \leftarrow \gamma_{\mathsf{bor}}, \_, \_]}^{\gamma_{\mathsf{i}}} * \boxed{a : \mathit{BorBox}}^{\gamma_{\mathsf{bor}}} \\
\mathsf{OwnInh}(\kappa, a) &:= \exists \gamma_{\mathsf{inh}}.\, \boxed{\circ\,[\kappa \leftarrow \_, \gamma_{\mathsf{inh}}, \_]}^{\gamma_{\mathsf{i}}} * \boxed{a : \mathit{InhBox}}^{\gamma_{\mathsf{inh}}} \\
(!)\quad \mathsf{OwnCnt}(\kappa, a) &:= \exists \gamma_{\mathsf{cnt}}.\, \boxed{\circ\,[\kappa \leftarrow \_, \_, \gamma_{\mathsf{cnt}}]}^{\gamma_{\mathsf{i}}} * \boxed{a : \mathit{Cnt}}^{\gamma_{\mathsf{cnt}}}
\end{aligned}$$

Borrows.

$$\begin{aligned}
[\mathsf{Bor} : (\kappa', \iota)]_q &:= \mathsf{OwnBor}(\kappa', \circ\iota \leftarrow (\mathsf{in}, q)) \\
(!)\quad \&^{\kappa}_{(\kappa',\iota)}\, P &:= \exists P'.\, \triangleright\Box(P \ast\!\!\ast\; P') * \kappa \sqsubseteq \kappa' * \mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P', \iota) \\
(!)\quad \mathsf{RawBor}(\kappa, P) &:= \exists \iota, P'.\, \triangleright\Box(P \ast\!\!\ast\; P') * \mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P', \iota) * \mathsf{OwnBor}(\kappa, \circ(\iota \leftarrow (\mathsf{in}, 1))) \\
\&^{\kappa}_{\mathsf{full}}\, P &:= \exists \kappa'.\, \kappa \sqsubseteq \kappa' * \mathsf{RawBor}(\kappa', P)
\end{aligned}$$

Invariant.

$$\begin{aligned}
\mathsf{LftInh}(\kappa, P_I, s) &:= \exists E : \wp^{\mathsf{fin}}(\mathbb{N}).\, \mathsf{OwnInh}(\kappa, \bullet E) * \mathsf{Box}(\mathcal{N}_{\mathsf{inh}}, P_I, [\iota \leftarrow s \mid \iota \in E]) \\
\mathsf{LftBorDead}(\kappa) &:= \exists B : \wp^{\mathsf{fin}}(\mathbb{N}), P_B : \mathit{iProp}.\, \mathsf{OwnBor}(\kappa, \bullet[\iota \leftarrow (\mathsf{in}, 1) \mid \iota \in B]) * {} \\
&\qquad \mathsf{Box}(\mathcal{N}_{\mathsf{bor}}, P_B, [\iota \leftarrow \mathsf{empty} \mid \iota \in \mathsf{dom}(B)]) \\
(!)\quad \mathsf{LftDead}(\kappa) &:= \exists P_I.\, \mathsf{LftBorDead}(\kappa) * \mathsf{OwnCnt}(\kappa, \bullet 0) * \mathsf{LftInh}(\kappa, P_I, \mathsf{full})
\end{aligned}$$

$$\mathsf{bor\_to\_box}(s) := \begin{cases} \mathsf{full} & \text{if } s = \mathsf{in} \\ \mathsf{empty} & \text{otherwise} \end{cases}$$

$$(!)\quad \mathsf{LftBorDeposit}(\kappa, B) := \mathop{\Huge\ast}_{\iota \in \mathsf{dom}(B)} \left\{ \begin{array}{ll} \mathsf{True} & \text{if } B(\iota) = \mathsf{in} \\ [\kappa]_q & \text{if } B(\iota) = \mathsf{open}(q) \\ \mathsf{OwnCnt}(\kappa', \circ 1) * \kappa \subset \kappa' & \text{if } B(\iota) = \mathsf{rebor}(\kappa') \end{array} \right\}$$

$$\begin{aligned}
\mathsf{LftBorAlive}(\kappa, P_B) &:= \exists B : \mathbb{N} \overset{\text{fin}}{\rightharpoonup} \mathit{BorSt}.\, \mathsf{OwnBor}(\kappa, \bullet[\iota \leftarrow (B(\iota), 1) \mid \iota \in \mathsf{dom}(B)]) * {} \\
&\qquad \mathsf{Box}(\mathcal{N}_{\mathsf{bor}}, P_B, \mathsf{bor\_to\_box} \langle\$\rangle\ B) * \mathsf{LftBorDeposit}(\kappa, B) \\
(!)\quad \mathsf{LftVs}(\kappa, P_B, P_I, n) &:= \forall I : \wp^{\mathsf{fin},+}(\mathbb{N}) \overset{\text{fin}}{\rightharpoonup} \mathcal{G} \times \mathcal{G} \times \mathcal{G}. \\
&\qquad \triangleright P_B * [\dagger\kappa] \Rrightarrow\!\!\ast \left[\boxed{\bullet I}^{\gamma_{\mathsf{i}}} * \mathop{\Huge\ast}_{\substack{\kappa' \in \mathsf{dom}(I) \\ \kappa' \subset \kappa}} \mathsf{LftAlive}(\kappa') \right]_{\mathcal{N}_{\mathsf{bor}}} \triangleright P_I * \mathsf{OwnCnt}(\kappa, \circ n) \\
(!)\quad \mathsf{LftAlive}(\kappa) &:= \exists P_B, P_I, n.\, \mathsf{LftBorAlive}(\kappa, P_B) * \mathsf{LftVs}(\kappa, P_B, P_I, n) * \mathsf{OwnCnt}(\kappa, \bullet n) * {} \\
&\qquad \mathsf{LftInh}(\kappa, P_I, \mathsf{empty})
\end{aligned}$$

Figure 11.14: Model of the lifetime logic *with* reborrowing.

to LftVs. However, all the other proof rules still work basically the same, so we will not reprove them here.

11.7.3 Unnesting of raw borrows

Our goal in this subsection is to prove the unnesting rule for raw borrows:

$$\kappa \subset \kappa' * \mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota) * \mathsf{RawBor}(\kappa', [\mathsf{Bor} : (\kappa, \iota)]_1) \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}}} \\ \mathsf{RawBor}(\kappa', P) \qquad \text{(LftL-raw-unnest)}$$

We start by opening the lifetime logic invariant, and observe that κ must already exist as a lifetime (because we have a raw borrow at that lifetime). If κ' is dead, we can easily solve the goal with faking. If it is alive, then κ must also be alive as it intersects strictly fewer atomic lifetimes. Note that $\kappa \neq \kappa'$, so at this point we own $\triangleright \mathsf{LftAlive}(\kappa) * \triangleright \mathsf{LftAlive}(\kappa')$ and we can temporarily break the invariant of both lifetimes at the same time.

The proof now consists of four steps:

1. Removing the borrow of $[\mathsf{Bor} : (\kappa, \iota)]_1$ from κ', so that we have full ownership of $\mathsf{RawBor}(\kappa, P)$.[67]
2. Opening the borrow slice ι in κ that holds P, putting it into state $\mathsf{rebor}(\kappa')$.
3. Using the $\triangleright P$ that we now hold to create a new borrow of P in κ'.
4. Patch everything up again in $\mathsf{LftVs}(\kappa', \ldots)$: that view shift runs when κ' (the shorter of the two lifetimes) ends, and it will take P back out of the borrow we created and put it back into the original borrow slice ι in κ that it came from—κ is a longer lifetime so it has not ended yet, it is not too late to do this. This changes the borrow state of ι back to in, which is good because that is exactly what we need to repair the "imbalance" between the borrow and inheritance box of κ' that we created in the first step.

   Basically, we use this view shift as a "hook" that runs when κ' ends, and since κ' ends before anyone can inherit the borrowed $[\mathsf{Bor} : (\kappa, \iota)]_1$ and before κ ends, the hook runs just in time before anything can be upset by how we rearranged all the resources.

The first step is to remove the borrow slice of κ' that holds the borrow token for (κ, ι), which also changes the total content of the borrow box of κ' to $P'_{B,\kappa'}$:

$$\triangleright \mathsf{LftBorAlive}(\kappa', P_{B,\kappa'}) * \mathsf{RawBor}(\kappa', [\mathsf{Bor} : (\kappa, \iota)]_1) \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}} \\ [\mathsf{Bor} : (\kappa, \iota)]_1 * \exists P'_{B,\kappa'}.\ \triangleright \mathsf{LftBorAlive}(\kappa', P'_{B,\kappa'}) * \\ \triangleright\triangleright(P_{B,\kappa'} = (P'_{B,\kappa'} * [\mathsf{Bor} : (\kappa, \iota)]_1))$$

We start by unfolding RawBor to learn the name of the underlying borrow slice, ι'. Combining the borrow token in that definition with the authoritative state in $\mathsf{LftBorAlive}$, we learn that the current borrow state is $B(\iota') = \mathsf{in}$, so the slice is full. We use Slice-delete-full to remove that slice, obtaining $\triangleright [\mathsf{Bor} : (\kappa, \iota)]_1$.[68] We remove ι' from B and the authoritative state. $\mathsf{LftBorDeposit}$ is unaffected because we just removed

[67] We do not just empty that borrow slice, we really remove it from the borrow box. Of course, this creates an imbalance with the inheritance box; we will use LftVs to put that borrow token back in place just in time.

[68] Borrow tokens are timeless, so we can remove the $\triangleright$.

a borrow in the in state, which does not have any deposit. Thus we are done.

In the second step, we open P borrowed at κ. Unlike LftL-bor-acc-strong, which moves borrows into state open and leaves a fraction of the lifetime token as a deposit, we will move the borrow into state rebor and leave a reborrow token of κ' as a deposit. We can obtain that reborrow token by increasing n by one in the authoritative state $\mathsf{OwnCnt}(\kappa', \bullet n)$.

$$\begin{aligned} &\triangleright \mathsf{LftBorAlive}(\kappa, P_{B,\kappa}) * [\mathsf{Bor} : (\kappa, \iota)]_1 * \\ &\mathsf{BoxSlice}(\mathcal{N}_{\mathsf{bor}}, P, \iota) * \mathsf{OwnCnt}(\kappa', \circ 1) \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}} \\ &\qquad \triangleright \mathsf{LftBorAlive}(\kappa, P_{B,\kappa}) * \triangleright P * \mathsf{OwnBor}(\kappa, \circ(\iota \leftarrow (\mathsf{rebor}(\kappa'), 1))) \end{aligned}$$

Based on the borrow token we own, we have $B(\iota) = \mathsf{in}$, so the slice is currently full. We use Slice-empty to empty it and obtain $\triangleright P$, and update B and the authoritative state of ι to $\mathsf{rebor}(\kappa')$. To re-prove $\mathsf{LftBorDeposit}$, this requires us to leave behind $\mathsf{OwnCnt}(\kappa', \circ 1)$.

For the third step, we create a new borrow in κ' as usual:

$$\begin{aligned} &\triangleright \mathsf{LftBorAlive}(\kappa', P'_{B,\kappa'}) * \triangleright P \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}} \\ &\qquad \triangleright \mathsf{LftBorAlive}(\kappa', P'_{B,\kappa'} * P) * \mathsf{RawBor}(\kappa', P) \end{aligned}$$

This is in fact an instance of Lft-bor' that we proved during LftL-borrow, so we can just reuse that lemma.

To close everything again, we need to re-establish the invariants: $\triangleright \mathsf{LftAlive}(\kappa) * \triangleright \mathsf{LftAlive}(\kappa')$. The first conjunct is easy (we did not change anything big in κ), but in κ' we changed the total borrow box content and the number of reborrows, so we need to re-establish LftVs:

$$\begin{aligned} &\triangleright \mathsf{LftVs}(\kappa', P_{B,\kappa'}, P_{I,\kappa'}, n) * \triangleright\triangleright(P_{B,\kappa'} = (P'_{B,\kappa'} * [\mathsf{Bor} : (\kappa, \iota)]_1)) * \\ &\mathsf{OwnBor}(\kappa, \circ(\iota \leftarrow (\mathsf{rebor}(\kappa'), 1))) \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}} \\ &\qquad \triangleright \mathsf{LftVs}(\kappa', P'_{B,\kappa'} * P, P_{I,\kappa'}, n + 1) \end{aligned}$$

For this, it is sufficient to prove:

$$\begin{gathered} \triangleright P * \mathsf{OwnBor}(\kappa, \circ(\iota \leftarrow (\mathsf{rebor}(\kappa'), 1))) \\ \Rrightarrow\!\!\!* \left[\boxed{\bullet I}^{\gamma_i} * \mathop{\huge *}_{\substack{\kappa'' \in \mathsf{dom}(I) \\ \kappa'' \subset \kappa'}} \mathsf{LftAlive}(\kappa'') \right]_{\mathcal{N}_{\mathsf{bor}}} \\ [\mathsf{Bor} : (\kappa, \iota)]_1 * \mathsf{OwnCnt}(\kappa', \circ 1) \end{gathered}$$

To see why, consider that the LftVs we aim to prove starts with resources $\triangleright(P'_{B,\kappa'} * P)$ and needs to produce $P_{I,\kappa'} * \mathsf{OwnCnt}(\kappa', \circ(n+1))$. The above lemma says that we can in a first step turn $\triangleright P$ together with the borrow token that we also have into $[\mathsf{Bor} : (\kappa, \iota)]_1$ and a reborrow token (only n tokens left to go). This means we own $\triangleright(P'_{B,\kappa'} * [\mathsf{Bor} : (\kappa, \iota)]_1)$ which we know to equal $\triangleright P_{B,\kappa'}$,[69] and we have a LftVs to turn that into $P_{I,\kappa'}$ and the remaining reborrow counting tokens.

In order to prove this huge view shift, we first observe that $\kappa \subset \kappa'$, so the view shift "frame" that we have access to contains $\mathsf{LftAlive}(\kappa)$ and

[69] This exploits that $\triangleright(P = Q) \twoheadrightarrow \triangleright P = \triangleright Q$.

in particular $\mathsf{LftBorAlive}(\kappa, \_)$. We use our borrow token for slice ι of the κ borrow box to learn that $B(\iota) = \mathsf{rebor}(\kappa')$. This means that in $\mathsf{LftBorDeposit}(\kappa, B)$, we can find $\mathsf{OwnCnt}(\kappa', \circ 1)$, satisfying the second separating conjunct of our goal. We update B and the authoritative state of ι to in, so no deposit is required any more. Our borrow token changes to $\mathsf{OwnBor}(\kappa, \circ(\iota \leftarrow (\mathsf{in}, 1)))$, satisfying the first separating conjunct of the goal.[70] After applying Slice-fill, borrow box state and borrow state match again. This uses up $\triangleright P$, but we re-established $\mathsf{LftBorAlive}(\kappa, \_)$, so we are done.

11.7.4 *Ending lifetimes in the presence of reborrowing*

The part of LftL-begin that is concerned with allocating a new atomic lifetime works just as before. However, the following lemma expressing the core of ending that lifetime again needs to be reproven:

$$\mathsf{LftLInv} * [\{\Lambda\}]_1 \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}} \mathsf{LftLInv} * [\dagger\, \{\Lambda\}] \qquad \text{(ALft-end)}$$

The reason this is now more tricky than before is that we changed LftVs. Using that view shift now requires access to resources of other lifetimes. Concretely, it requires $\mathsf{LftAlive}(\kappa')$ for all $\kappa' \subset \kappa$, *i.e.*, all the syntactically longer lifetimes (consisting of an intersection of fewer atomic lifetimes) need to be still alive. Ending the atomic lifetime Λ ends not just one lifetime but all $\kappa \ni \Lambda$ that have not been ended yet, and LftVs requires us to carefully do that in the right order.

We will start this proof bottom-up, with a lemma to end an individual lifetime κ:

$$\begin{aligned}
&(\forall \kappa'.\, \kappa' \in \mathsf{dom}(I) \land \kappa' \subset \kappa \Rightarrow \kappa' \in K_a) \, * \\
&(\forall \kappa'.\, \kappa' \in \mathsf{dom}(I) \land \kappa \subset \kappa' \Rightarrow \kappa' \in K_d) * \mathsf{LftAlive}(\kappa) * [\dagger\kappa] \\
&\Rrightarrow\!\!\ast \left[\boxed{\bullet I}^{\gamma_i} * \mathop{\Huge\ast}_{\kappa_a \in K_a} \mathsf{LftAlive}(\kappa_a) * \mathop{\Huge\ast}_{\kappa' \in K_d} \mathsf{LftDead}(\kappa_d) \right]_{\mathcal{N}_{\mathsf{lft}} \setminus \mathcal{N}_{\mathsf{mgmt}}} \\
&\mathsf{LftDead}(\kappa) \qquad \text{(Lft-end)}
\end{aligned}$$

Here, K_a is a finite set of *alive* lifetimes that must contain all κ' that are strict subsets of κ (and hence syntactically longer lifetimes), and K_d is a finite set of *dead* lifetimes that must contain all κ' that are strict supersets of κ (and hence syntactically shorter lifetimes). In other words, to end a lifetime, shorter lifetimes must be already dead and longer lifetimes must be still alive.

As before, the first step is to show that all borrow slices are full, *i.e.*, that $B(\iota) = \mathsf{in}$ for all ι. We can rule out $\mathsf{open}(q)$ because for open borrow slices, $\mathsf{LftBorDeposit}$ contains $[\kappa]_q$, which leads to a contradiction since we have $[\dagger\kappa]$. For reborrowed slices, intuitively the argument is that we only ever reborrow to syntactically shorter lifetimes, but those have all been ended already, and when a lifetime ends it puts its reborrowed resources back.[71] Such slices are in state $\mathsf{rebor}(\kappa')$, which means we have a deposit of $\mathsf{OwnCnt}(\kappa', \circ 1)$ and we also learn that $\kappa \subset \kappa'$. However, this implies $\kappa' \in K_d$, *i.e.*, κ' is known to be an already dead lifetime. Thus we have

[70] Remember that these tokens count the number of reborrows that a lifetime has taken from other lifetimes. The reason we can get back that token now is that, as part of this LftVs, we have put the reborrow back. This is important because LftVs must collect *all* reborrow tokens; we need to be sure no reborrows are "forgotten" when the lifetime is ended.

[71] Remember that when we proved LftVs for LftL-raw-unnest, we did exactly that: we put the resources back in the slice they got reborrowed from, and changed the state of that borrow slice back to in.

$\mathsf{LftDead}(\kappa')$, which implies $\mathsf{OwnCnt}(\kappa', \bullet 0)$. Now we have a contradiction: it is impossible for the authoritative state to be 0 when we own a fragment with value 1.[72] Thus we know $B(\iota) = \mathsf{in}$.

After emptying the borrow box (Box-empty), we obtain $\triangleright P_B$ and $\mathsf{LftBorDead}(\kappa)$ as before. Next, we apply LftVs. We have all its preconditions: $\triangleright P_B$, $[\dagger\kappa]$, as well as the frame: the authoritative I and $\mathsf{LftAlive}$ for all syntactically longer lifetimes (as those are all in K_a). We get back the frame and obtain $\triangleright P_I$ and $\mathsf{OwnCnt}(\kappa, \circ n)$. Since we also own the matching $\mathsf{OwnCnt}(\kappa, \bullet n)$ (it is in $\mathsf{LftAlive}(\kappa)$), we can do a frame-preserving update that consumes both of them and produces $\mathsf{OwnCnt}(\kappa, \bullet 0)$.[73] As before, we fill the inheritance box, obtaining $\mathsf{LftInh}(\kappa, P_I, \mathsf{full})$ which completes the proof of Lft-end.

Next, we show how to end a whole set K_e of lifetimes at once. To this end, the set K_e must be closed under syntactically shorter lifetimes, *i.e.*, larger sets. We cannot end a lifetime and keep some shorter lifetime alive! Furthermore, in the disjoint set K_a we collect all alive lifetimes that will not be ended—we need access to all of their invariants for this proof. Also note that lifetimes in K_e are allowed to be already dead; the lemma will simply end all the ones that are still alive.

$$\begin{aligned}
&K_e \cap K_a = \emptyset * (\forall \kappa \in K_e, \kappa' \in \mathsf{dom}(I).\, \kappa' \supseteq \kappa \Rightarrow \kappa' \in K_e) * \\
&(\forall \kappa \in \mathsf{dom}(I).\, \mathsf{LftAliveIn}(A, \kappa) \wedge \kappa \notin K_e \Rightarrow \kappa \in K_a) * \\
&\left(\mathop{\Huge *}_{\kappa \in K_e} \mathsf{LftInv}(A, \kappa) * [\dagger\kappa] \right) \\
&\Rrightarrow\!\!* \left[\boxed{\bullet I}^{\gamma_{\mathsf{i}}} * \mathop{\Huge *}_{\kappa' \in K_a} \mathsf{LftAlive}(\kappa') \right]_{-\mathcal{N}_{\mathsf{mgmt}}} \\
&\mathop{\Huge *}_{\kappa \in K_e} \mathsf{LftDead}(\kappa) \qquad \text{(Lfts-end)}
\end{aligned}$$

The key to this proof will be to end the lifetimes in the right order so that we can satisfy the requirements of Lft-end: that shorter lifetimes are already dead and longer ones are still alive. Formally, we perform induction over the metric $|K_e|$, *i.e.*, over the size of the kill-set K_e. We start by checking whether K_e contains any lifetimes κ such that $\mathsf{LftAliveIn}(A, \kappa)$. If no, we have nothing to do—all to-be-ended lifetimes are already ended. Otherwise, we select a *minimal* element $\kappa \in \{\kappa \in K_e \mid \mathsf{LftAliveIn}(A, \kappa)\}$ according to the relation $\subset$, *i.e.*, we pick some lifetime such that no longer lifetime is still alive and should be ended. This relation is acyclic, so such an element has to exist. We will first end *all the other lifetimes* and then end κ: "longest lifetime last".

We choose $K'_e := K_e \setminus \{\kappa\}$ and $K'_a := K_a \cup \{\kappa\}$ as kill-set and alive-set for the induction hypothesis. Clearly, K'_e is smaller than K_e, so the induction is well-founded. To actually apply the induction hypothesis, we have to show that K'_e and K'_a also satisfy the other preconditions. Clearly, they are still disjoint. K_e is also still closed under larger sets of atomic lifetimes because we removed a minimal element: κ is not only minimal in $\{\kappa \in K_e \mid \mathsf{LftAliveIn}(A, \kappa)\}$, it is in fact minimal in K_e (*i.e.*, when considering all to-be-ended lifetimes, not just the alive ones). This

[72] In other words, it is impossible to have an outstanding reborrow to an already dead lifetime. This argument is the reason why the borrow counting tokens exist in the first place.

[73] This is where we certify that we have put back all reborrows that we might have taken from other lifetimes. The reason this is possible is that LftVs is defined to produce a number of reborrow tokens exactly matching the total number n that LftAlive authoritatively keeps track of.

is because all lifetimes longer than κ are still alive since κ is still alive. Furthermore we have to show that K'_a contains all alive lifetimes. So let us assume that there is some $\kappa' \in K'_e$ that is alive. Either $\kappa' \notin K_e$ (so we are done by our assumption about K_a), or $\kappa' = \kappa$ (and we are done because $\kappa \in K'_a$). We also need $\mathsf{LftAlive}(\kappa)$ (since we added κ in K'_a), which we have because we specifically picked κ to be alive. After invoking the induction hypothesis, we have that all lifetimes in K'_e are dead. To complete our goal, all that is left to do is end κ.

To this end, we invoke Lft-end. Our K_a satisfies the requirement of containing all $\kappa' \subset \kappa$, because κ is minimal in K_e so $\kappa' \notin K_e$. We also have $\mathsf{LftAlive}$ for all these liftimes. For K_d we pick K'_e, for which we have $\mathsf{LftDead}$ because we just ended them all. It contains all supersets of κ because $\kappa \in K_e$ and K_e is closed under supersets. Thus all assumptions of Lft-end are met, and we are done proving Lfts-end.

Finally, we can come back to proving ALft-end. Based on the token we own, we can switch the status of Λ in A to $\mathsf{inr}()$, obtaining $[\dagger\Lambda]$. Now we apply Lfts-end with

$$\begin{aligned} K_e &:= \{\kappa \in \mathsf{dom}(I) \mid \Lambda \in \kappa\} \\ K_a &:= \{\kappa \in \mathsf{dom}(I) \mid \kappa \notin K_e \wedge \mathsf{LftAliveIn}(A, \kappa)\} \end{aligned}$$

K_e is clearly closed under supersets. And by definition, everything alive and not to be ended is in K_a. From $[\dagger\,\{\Lambda\}]$ we can get the corresponding token $[\dagger\kappa]$ for all $\kappa \in K_e$, and we defined everything in K_a to be alive. We thus satisfy all requirements of Lfts-end, which finishes the proof.

Chapter 12
Semantic type system soundness

In §10, we defined the semantic interpretation (size, ownership predicate, and in some cases sharing predicate) for some of the key λ_{Rust} types. To make these definitions fully precise, we had to develop the *lifetime logic* (§11). Now we can get back to the type system and complete the semantic model of types and all the other type system components. Once the semantic model is complete, we can verify the following key theorem relating our syntactic and semantic type systems:

Theorem 4 (Fundamental theorem of logical relations)**.**
For any inference rule of the type system given in §9, when we replace all occurrences of $\vdash$ *by* $\models$ *(replacing syntactic judgments by their semantic interpretation), the resulting Iris theorem holds.*

One important corollary of the fundamental theorem is that if a judgment can be derived syntactically, then it also holds semantically.[1] However, Theorem 4 is much stronger than this, because we can use it to glue together safe and `unsafe` code. Given a program that is syntactically well-typed except for certain components that are only semantically (but not syntactically) well-typed, the fundamental theorem tells us that the entire program is semantically well-typed.

On its own, the fundamental theorem is not terribly useful yet, as it just relates two different ways to "type" a program. This is where the adequacy theorem comes in, which relates semantic typing to *program behavior*:

Theorem 5 (Adequacy)**.** *Let* f *be a* λ_{Rust} *function such that* $\emptyset; \emptyset \mid \emptyset \models f \Dashv x.\, x \lhd \mathbf{fn}() \to \Pi[]$ *holds. When we execute* f *with the default continuation (a no-op), no execution reaches a stuck state,* i.e., *a state where any thread cannot make progress.*

In particular, the adequacy theorem guarantees that a semantically well-typed program is memory and thread safe: it will never perform any invalid memory access and will not have data races.

Put together, these theorems establish that, *if the only code in a* λ_{Rust} *program that is not syntactically well-typed appears in semantically well-typed libraries, then the program is safe to execute.* In other words, to ensure safety of a whole Rust program, we only need to verify its `unsafe` libraries.

[1] This follows trivially by induction on the derivation tree of the judgment. Every step in that tree is a proof rule which can be mirrored semantically using the fundamental theorem.

Semantic model of lifetimes and lifetime contexts.

$$
\begin{aligned}
\llbracket \kappa \rrbracket &: \mathit{Lft} \\
\llbracket \alpha \rrbracket &:= \alpha \\
\llbracket \textsf{static} \rrbracket &:= \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
\llbracket \mathbf{E} \rrbracket &: \mathit{iProp} \\
\llbracket \emptyset \rrbracket &:= \textsf{True} \\
\llbracket \mathbf{E}, \kappa \sqsubseteq_{\textsf{e}} \kappa' \rrbracket &:= \llbracket \kappa \rrbracket \sqsubseteq \llbracket \kappa' \rrbracket * \llbracket \mathbf{E} \rrbracket
\end{aligned}
$$

$$
\begin{aligned}
\llbracket \mathbf{L} \rrbracket &: \mathit{Fract} \to \mathit{iProp} \\
\llbracket \emptyset \rrbracket(q) &:= \textsf{True} \\
\llbracket \mathbf{L}, \kappa \sqsubseteq_{\textsf{l}} \bar{\kappa} \rrbracket(q) &:= \exists \kappa'.\, \llbracket \kappa \rrbracket = \kappa' \sqcap \Big(\bigsqcap\nolimits_{\kappa'' \in \llbracket \bar{\kappa} \rrbracket} \kappa''\Big) * [\kappa']_q * \Box\big([\kappa']_1 \;{}^{\emptyset}\!\!\Rrightarrow\!\!\ast^{\mathcal{N}_{\textsf{lft}}}\; [\dagger \kappa']\big) * \llbracket \mathbf{L} \rrbracket(q)
\end{aligned}
$$

Semantic model of lifetime judgments.

$$
\begin{aligned}
\mathbf{E}; \mathbf{L} \models \kappa \;\textsf{alive} &:= \Box \forall q.\, \llbracket \mathbf{E} \rrbracket \mathrel{-\!\!*} \big(\llbracket \mathbf{L} \rrbracket(q) \;\propto_{\mathcal{N}_{\textsf{lft}}, \mathcal{N}_{\textsf{rust}}}\; q'.\, [\llbracket \kappa \rrbracket]_{q'}\big) \\
\mathbf{E}; \mathbf{L} \models \kappa_1 \sqsubseteq \kappa_2 &:= \Box \forall q.\, \llbracket \mathbf{L} \rrbracket(q) \mathrel{-\!\!*} \Box(\llbracket \mathbf{E} \rrbracket \mathrel{-\!\!*} \llbracket \kappa_1 \rrbracket \sqsubseteq \llbracket \kappa_2 \rrbracket) \\
\mathbf{E}_1; \mathbf{L}_1 \models \mathbf{E}_2 &:= \Box \forall q.\, \llbracket \mathbf{L} \rrbracket(q) \mathrel{-\!\!*} \Box(\llbracket \mathbf{E}_1 \rrbracket \mathrel{-\!\!*} \llbracket \mathbf{E}_2 \rrbracket)
\end{aligned}
$$

Figure 12.1: Semantic models of lifetimes, lifetime contexts, and judgments.

Chapter outline. To complete our semantic model and show these two theorems, we start by giving a semantic interpretation of the $\lambda_{\textsf{Rust}}$ lifetime contexts (**E** and **L**) and the lifetime-related judgments (such as lifetime inclusion) in terms of the lifetime logic, and verifying that they properly model the syntactic judgments (§12.1). Then we finish the job that we started in §10 and give a semantic interpretation for every built-in type of $\lambda_{\textsf{Rust}}$ (§12.2). Next, we have to introduce the mechanism of *non-atomic invariants* (§12.3), which is required to verify non-thread-safe types with interior mutability (`Cell`, `RefCell` and `Rc`), as we will see in §13. Finally, we can give a semantic interpretation of the type system judgments (§12.4). With the right definitions, it will turn out that Theorem 5 is actually easy to prove.[2] Showing Theorem 4, on the other hand, requires a proof for *every single typing rule*; we will look at a few representative examples in §12.5.

12.1 Semantically modeling $\lambda_{\textsf{Rust}}$ lifetime judgments

As a simple example for how to define a semantic model of type system judgments and verify Theorem 4, we start by looking at the lifetime contexts **E** and **L** and the three lifetime-related judgments: liveness, lifetime inclusion, and external lifetime context satisfiability. The relevant definitions are all given in Figure 12.1.

Interpreting lifetime contexts. Both lifetime contexts are basically a big separating conjunction over the interpretation of each element in

[2] This is common for the semantic approach to type soundness.

the context. In the *external lifetime context* $\mathbf{E}$, each element $\kappa \sqsubseteq_{\mathsf{e}} \kappa'$ corresponds directly to a lifetime inclusion in the lifetime logic.

The *local lifetime context* $\mathbf{L}$ is slightly more involved. We say that $\kappa \sqsubseteq_{\mathsf{l}} \bar{\kappa}$ means that κ is the intersection of all lifetimes in $\bar{\kappa}$ further intersected with one more lifetime κ'. The purpose of κ' is to be able to end κ: for κ' we own the lifetime token and we also own the view shift (produced by LftL-begin) that lets us end this lifetime. The fraction q of the lifetime that we own is given as a parameter. This fraction will be universally quantified in the models of the lifetime judgments so that multiple judgments can be used at the same time by splitting the context (using $[\![\mathbf{L}]\!](q_1 + q_2) \ast\!\ast [\![\mathbf{L}]\!](q_1) * [\![\mathbf{L}]\!](q_2)$).

Finally, we define the **static** lifetime in λ_{Rust} to correspond to the ε lifetime of the lifetime logic, which ensures that **static** goes on forever and cannot be ended.

Interpreting lifetime judgments. With the semantic model of lifetime contexts in place, we can proceed by giving a semantic model to the three judgments that operate on these contexts. All of these interpretations are made persistent with an explicit $\square$ modality, to ensure that the proof of a judgment does not implicitly use any non-duplicable resources.

Liveness ($\mathbf{E}; \mathbf{L} \models \kappa$ alive) is the most straightforward of the judgments. We say that a lifetime κ is alive in some context if we can prove a symmetric accessor that provides some fraction of the token for that lifetime.

Lifetime inclusion in λ_{Rust} ($\mathbf{E}; \mathbf{L} \models \kappa_1 \sqsubseteq \kappa_2$) is interpreted using the notion of lifetime inclusion provided by the lifetime logic, as one would expect. What is subtle about the semantic model here is the second $\square$ modality: remember that the semantic interpretation of $\kappa \sqsubseteq_{\mathsf{l}} \bar{\kappa}$ in the local lifetime context says that we *own* fraction q of the token for κ', where κ is obtained by intersecting κ' with the lifetimes in $\bar{\kappa}$. This means that the local lifetime context is *not* persistent. However, for our semantic model of inclusion, we only want a proof of inclusion to be able to use the persistent parts of the context (notably, the fact that κ is κ' intersected with $\bar{\kappa}$). This is expressed by putting the persistence modality $\square$ around the conclusion, which makes sure that only persistent resources can be used to prove the remaining goal. We do this to ensure that using this lemma does not "use up" ownership of $[\![\mathbf{L}]\!](q)$.[3]

Similarly, *external lifetime context satisfiability* ($\mathbf{E}_1; \mathbf{L}_1 \models \mathbf{E}_2$) is defined as $[\![\mathbf{E}_2]\!]$ following from $[\![\mathbf{E}_1]\!]$, making use of only the persistent parts of $[\![\mathbf{L}]\!](q)$.

The fundamental theorem for lifetime judgments. We now have all the pieces needed to show Theorem 4 for lifetime judgments: for each of the rules of these three judgments, we can show the corresponding entailment of their semantic interpretations in Iris.

First, we consider lifetime inclusion. The typing rules for lifetime inclusion in λ_{Rust} are given in Figure 9.5 on page 115 (except for transitivity, which is on page 125 in §9.4). Thus we have to prove the Iris theorems shown in Figure 12.2.

[3] This exploits $\square$-wand-keep (page 69): when we have $P \twoheadrightarrow \square Q$, applying that magic wand does not "use up" P. To avoid the modality, we could have used $[\![\mathbf{L}]\!](q) * [\![\mathbf{E}]\!] \twoheadrightarrow [\![\mathbf{L}]\!](q) * [\![\mathbf{E}]\!] * ([\![\kappa_1]\!] \sqsubseteq [\![\kappa_2]\!])$ as the semantic interpretation, but that would be much more annoying to use in Coq proofs as we would have to constantly use up and re-acquire ownership of $[\![\mathbf{L}]\!](q)$ and $[\![\mathbf{E}]\!]$.

LINCL-STATIC-SEM
$$\mathbf{E}; \mathbf{L} \models \kappa \sqsubseteq \textsf{static}$$

LINCL-LOCAL-SEM
$$\frac{\kappa \sqsubseteq_{\mathsf{l}} \bar{\kappa} \in \mathbf{L} \qquad \kappa' \in \bar{\kappa}}{\mathbf{E}; \mathbf{L} \models \kappa \sqsubseteq \kappa'}$$

LINCL-EXTERN-SEM
$$\frac{\kappa \sqsubseteq_{\mathsf{e}} \kappa' \in \mathbf{E}}{\mathbf{E}; \mathbf{L} \models \kappa \sqsubseteq \kappa'}$$

LINCL-REFL-SEM
$$\mathbf{E}; \mathbf{L} \models \kappa \sqsubseteq \kappa$$

LINCL-TRANS-SEM
$$\frac{\mathbf{E}; \mathbf{L} \models \kappa \sqsubseteq \kappa' \qquad \mathbf{E}; \mathbf{L} \models \kappa' \sqsubseteq \kappa''}{\mathbf{E}; \mathbf{L} \models \kappa \sqsubseteq \kappa''}$$

Figure 12.2: Semantic rules for λ_{Rust} lifetime inclusion.

To show LINCL-REFL-SEM, we only need lifetime inclusion to be reflexive (LFTL-INCL-REFL on page 142). LINCL-STATIC-SEM is a consequence of LFTL-TOK-UNIT (which says that we can always create tokens "out of thin air" for the static lifetime[4]) and LFTL-END-UNIT (which says that a dead token for the static lifetime leads to a contradiction). LINCL-EXTERN-SEM is shown easily by induction over the context $\mathbf{E}$. Similarly, LINCL-LOCAL-SEM follows by induction over $\mathbf{L}$; the $\Box$ modality in the goal forces us to "throw away" most of $[\![\mathbf{L}]\!](q)$, but not before we remembered the (persistent) fact that $\kappa = \kappa' \sqcap \ldots$, from which inclusion follows easily.

Finally, we come to transitivity (LINCL-TRANS-SEM). This is the only one of these rules that also makes semantic *assumptions*. As a consequence, showing semantic versions of all syntactic proof rules like we are doing here is stronger than merely showing something like "syntactic inclusion implies semantic inclusion":

LINCL-SEM
$$\frac{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\mathbf{E}; \mathbf{L} \models \kappa \sqsubseteq \kappa'}$$

This rule is a *consequence* of the five lemmas we are proving (by induction on the derivation of $\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'$), but not vice versa: LINCL-TRANS-SEM does not follow from this rule because its semantic assumptions ($\models$) are weaker than the syntactic precondition ($\vdash$) of LINCL-SEM.

Actually showing LINCL-TRANS-SEM is not difficult, but a bit tedious. After unfolding the model of inclusion, we have to prove the following:

$$\begin{aligned}&\Box\big(\forall q.\, [\![\mathbf{L}]\!](q) \mathrel{-\!\!*} \Box([\![\mathbf{E}]\!] \mathrel{-\!\!*} [\![\kappa]\!] \sqsubseteq [\![\kappa']\!])\big) * \\ &\Box\big(\forall q.\, [\![\mathbf{L}]\!](q) \mathrel{-\!\!*} \Box([\![\mathbf{E}]\!] \mathrel{-\!\!*} [\![\kappa']\!] \sqsubseteq [\![\kappa'']\!])\big) \mathrel{-\!\!*} \\ &\qquad\qquad \Box\,\forall q.\, [\![\mathbf{L}]\!](q) \mathrel{-\!\!*} \Box([\![\mathbf{E}]\!] \mathrel{-\!\!*} [\![\kappa]\!] \sqsubseteq [\![\kappa'']\!])\end{aligned}$$

We show the proof outline in Figure 12.3. In the end, this boils down to LFTL-INCL-TRANS, but along the way we have to make use of $\Box$-WAND-KEEP which lets us keep $[\![\mathbf{L}]\!](q)$ when applying the first magic wand so that we can use it again for the second wand.[5]

The other two judgments, liveness and external lifetime context satisfiability, are handled in just the same way, with one Iris lemma per λ_{Rust} proof rule. But none of these proofs are very interesting, so we move on to (re-)considering the types of λ_{Rust} and the judgments relating them.

[4] Remember that $[\![\textsf{static}]\!] = \varepsilon$.

[5] We could alternatively complete the proof by splitting $[\![\mathbf{L}]\!](q)$, but we choose this opportunity to showcase this very useful and often overlooked interaction of magic wand and persistence.

$\{((\mathbf{1})\ \Box\,\forall q.\, \llbracket \mathbf{L} \rrbracket(q) \mathrel{-\!\!*} \Box(\llbracket \mathbf{E} \rrbracket \mathrel{-\!\!*} \llbracket \kappa \rrbracket \sqsubseteq \llbracket \kappa' \rrbracket)) * ((\mathbf{2})\ \Box\,\forall q.\, \llbracket \mathbf{L} \rrbracket(q) \mathrel{-\!\!*} \Box(\llbracket \mathbf{E} \rrbracket \mathrel{-\!\!*} \llbracket \kappa' \rrbracket \sqsubseteq \llbracket \kappa'' \rrbracket))\}$
Goal: $\Box\,\forall q.\, \llbracket \mathbf{L} \rrbracket(q) \mathrel{-\!\!*} \Box(\llbracket \mathbf{E} \rrbracket \mathrel{-\!\!*} \llbracket \kappa \rrbracket \sqsubseteq \llbracket \kappa'' \rrbracket)$
Introduce persistence modality ($\Box$-intro, page 69) and assumptions of magic wand.
$\{\llbracket \mathbf{L} \rrbracket(q)\}$
Instantiate (1) with $q := q$ and $\llbracket \mathbf{L} \rrbracket(q)$, which we can keep using $\Box$-wand-keep.
$\{\llbracket \mathbf{L} \rrbracket(q) * \Box(\llbracket \mathbf{E} \rrbracket \mathrel{-\!\!*} \llbracket \kappa \rrbracket \sqsubseteq \llbracket \kappa' \rrbracket)\}$
Instantiate (2) with $q := q$ and $\llbracket \mathbf{L} \rrbracket(q)$.
$\{((\mathbf{3})\ \Box(\llbracket \mathbf{E} \rrbracket \mathrel{-\!\!*} \llbracket \kappa \rrbracket \sqsubseteq \llbracket \kappa' \rrbracket)) * ((\mathbf{4})\ \Box(\llbracket \mathbf{E} \rrbracket \mathrel{-\!\!*} \llbracket \kappa' \rrbracket \sqsubseteq \llbracket \kappa'' \rrbracket))\}$
Goal: $\Box(\llbracket \mathbf{E} \rrbracket \mathrel{-\!\!*} \llbracket \kappa \rrbracket \sqsubseteq \llbracket \kappa'' \rrbracket)$
Introduce persistence modality ($\Box$-intro) and assumptions of magic wand.
$\{\llbracket \mathbf{E} \rrbracket\}$
Instantiate both (3) and (4) with $\llbracket \mathbf{E} \rrbracket$.
$\{\llbracket \kappa \rrbracket \sqsubseteq \llbracket \kappa' \rrbracket * \llbracket \kappa' \rrbracket \sqsubseteq \llbracket \kappa'' \rrbracket\}$
Lifetime inclusion is transitive (LftL-incl-trans).
Goal: $\llbracket \kappa \rrbracket \sqsubseteq \llbracket \kappa'' \rrbracket$

Figure 12.3: Proof outline for Lincl-trans-sem.

12.2 Semantically modeling λ_{Rust} types

In §10, we started defining semantic models of some key λ_{Rust} types. However, we did not go into the sharing predicate for owned pointers and mutable references; as it turns out, those are somewhat more complicated than one would expect. We also did not go through each type in the type system—however, most of the remaining types are not very interesting to look at. In this section, we will close those gaps.

Figure 12.4 repeats the *semantic domain of types* that we ended up with in §10. Defining a semantic type means picking an element of that domain. Figure 12.5 gives the semantic interpretation of each type in the type system, including those we already saw in §10. In the following, we go over the interesting aspects of those definitions that we have not yet considered.

Simple types. As discussed in §10.4, many of our types are what we call "simple types", which means they have size 1 and are `Copy`. For these types, it suffices to define a persistent predicate Φ that defines when a value v satisfies the type invariant in a given thread t. For example, values of type **bool** are either **`true`** or **`false`**, and shared references are defined by the sharing predicate of the referenced type τ (which is persistent by ty-shr-persist). Given such a predicate, the SimpleType constructor (defined at the bottom of Figure 12.5) determines both ownership and sharing predicate in terms of Φ. The ownership predicate is entirely straightforward; the sharing predicate makes use of fractured borrows (§11.3) to express that the pointed-to value cannot change.

We need to verify that SimpleType satisfies the conditions on semantic types as defined in Figure 12.4. Most of them are trivially satisfied, except for ty-share, where we need to prove:

$$\begin{aligned}&\&^{\kappa}_{\mathsf{full}}\big(\exists v.\, \ell \mapsto v * \Phi(t, v)\big) * [\kappa]_q \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}}} \\ &\qquad \exists v.\, \&^{\kappa}_{\mathsf{frac}}(\lambda q.\, \ell \overset{q}{\mapsto} v) * \triangleright \Phi(t, v) * [\kappa]_q\end{aligned}$$

$$
\begin{aligned}
\mathit{PreSemType} &:= \left\{\begin{array}{r}
\textsc{size} : \mathbb{N}, \\
\textsc{own} : \mathit{TId} \times \mathit{List}(\mathit{Val}) \to \mathit{iProp}, \\
\textsc{shr} : \mathit{Lft} \times \mathit{TId} \times \mathit{Loc} \to \mathit{iProp}
\end{array}\right\} \\
\mathit{SemType} &:= \left\{ T \in \mathit{PreSemType} \;\middle|\; \begin{array}{l}
\Big(\forall t, \bar{v}.\, T.\textsc{own}(t, \bar{v}) \mathrel{-\!\!*} |\bar{v}| = [\![T]\!].\textsc{size}\Big) \land \qquad (\textsc{ty-size}) \\
\Big(\forall \kappa, t, \ell.\, \mathsf{persistent}([\![\tau]\!].\textsc{shr}(\kappa, t, \ell))\Big) \land \qquad (\textsc{ty-shr-persist}) \\
\Big(\forall \kappa, \kappa', t, \ell.\, \kappa' \sqsubseteq \kappa * [\![\tau]\!].\textsc{shr}(\kappa, t, \ell) \mathrel{-\!\!*} [\![\tau]\!].\textsc{shr}(\kappa', t, \ell)\Big) \land \\
\qquad\qquad\qquad\qquad\qquad\qquad (\textsc{ty-shr-mono}) \\
\Big(\forall \kappa, t, \ell.\, \big(\&^{\kappa}_{\mathsf{full}}\big(\exists \overline{w}.\, \ell \mapsto \overline{w} * [\![\tau]\!].\textsc{own}(t, \overline{w})\big) * [\kappa]_q\big) \Rrightarrow\!\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \\
\qquad \big([\![\tau]\!].\textsc{shr}(\kappa, t, \ell) * [\kappa]_q\big)\Big) \qquad (\textsc{ty-share})
\end{array}\right\}
\end{aligned}
$$

Figure 12.4: Semantic domain of types.

We first use freezing (LftL-bor-exists, page 142) to swap the existential quantifier to the outside of the full borrow.[6] Next, we split $\&^{\kappa}_{\mathsf{full}}(\ell \mapsto v * \Phi(v))$ at the separating conjunction (LftL-bor-split). We use $[\kappa]_q$ to briefly open the borrow of $\Phi(t, v)$ to obtain a proof of $\triangleright \Phi(t, v)$ (remember Φ is persistent, and $\triangleright$ preserves persistence, so we can grab a copy). Finally, we turn the full borrow of $\ell \mapsto v$ into a fractional borrow (LftL-bor-fracture, page 146), and we are done.[7]

SimpleType is also used to define $\lightning_1$, the type representing an uninitialized block of memory of size 1. The interpretation of the general $\lightning_n$ is then defined as the product of n times $\lightning_1$.

Delayed sharing for owned pointers. We have already talked about the ownership predicate for owned pointers ($\mathbf{own}_n\ \tau$) in §10.3. In contrast to the ownership predicate, the sharing predicate is quite complicated:

$$
\begin{aligned}
[\![\mathbf{own}_n\ \tau]\!].\textsc{shr}(\kappa, t, \ell) := \exists \ell'.\ &\&^{\kappa}_{\mathsf{frac}}(\lambda q.\, \ell \overset{q}{\mapsto} \ell') * \\
&\Box\Big(\forall q.\, [\kappa]_q \mathrel{-\!\!*} {}^{\mathcal{N}_{\mathsf{shr}} \cup \mathcal{N}_{\mathsf{lft}}}\!\!\Rrightarrow^{\mathcal{N}_{\mathsf{lft}}} \triangleright\, {}^{\mathcal{N}_{\mathsf{lft}}}\!\!\Rrightarrow^{\mathcal{N}_{\mathsf{shr}} \cup \mathcal{N}_{\mathsf{lft}}} \\
&\qquad \big([\![\tau]\!].\textsc{shr}(\kappa, t, \ell') * [\kappa]_q\big)\Big)
\end{aligned}
$$

We call this pattern (the part after the first separating conjunction) "delayed sharing".[8] But why is it even needed? The reasons are subtle. The sharing predicate we would naively want to use is as follows:

$$
[\![\mathbf{own}_n\ \tau]\!].\textsc{shr}(\kappa, t, \ell) := \exists \ell'.\ \&^{\kappa}_{\mathsf{frac}}(\lambda q.\, \ell \overset{q}{\mapsto} \ell') * \triangleright [\![\tau]\!].\textsc{shr}(\kappa, t, \ell')
$$

This expresses that a shared borrow of an owned reference is a pointer to a pointer, and the outer pointer is handled just like a simple type—through a fractional borrow. While the owned pointer is shared, it cannot be mutated to point to something else. The second part says that what it points to is a valid shared instance of τ.[9]

However, if we went for this simple definition, we would find ourselves unable to prove ty-share. To see why, let us unfold (and slightly simplify)

[6] This is where the memory pointed to by the shared reference becomes read-only.

[7] The remaining borrow of $\Phi(t, v)$ is thrown away.

[8] The name will become clear soon.

[9] The recursive occurrence is guarded by a $\triangleright$ to account for recursive types, as discussed in §10.3.

$$\llbracket \textbf{bool} \rrbracket := \mathsf{SimpleType}(\lambda\_, v.\, v = \texttt{true} \vee v = \texttt{false})$$

$$\llbracket \textbf{int} \rrbracket := \mathsf{SimpleType}(\lambda\_, \bar{v}.\, \exists z.\, v = z)$$

$$\llbracket \textbf{own}_n\ \tau \rrbracket := \left\{\begin{array}{l}
\textsc{size} := 1, \\
\textsc{own} := \lambda t, \bar{v}.\, \exists \ell.\, \bar{v} = [\ell] * \triangleright\big(\exists \overline{w}.\, \ell \mapsto \overline{w} * \llbracket\tau\rrbracket.\textsc{own}(t, \overline{w})\big) * \\
\qquad\qquad \mathsf{Dealloc}(\ell, \llbracket\tau\rrbracket.\textsc{size}, n), \\
\textsc{shr} := \lambda \kappa, t, \ell.\, \exists \ell'.\, \&^{\kappa}_{\mathsf{frac}}(\lambda q'.\, \ell \overset{q'}{\mapsto} \ell') * \\
\qquad\qquad \square\Big(\forall q.\, [\kappa]_q \twoheadrightarrow {}^{\mathcal{N}_{\mathsf{shr}} \cup \mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{lft}}} \triangleright\, {}^{\mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{shr}} \cup \mathcal{N}_{\mathsf{lft}}} \\
\qquad\qquad\qquad \big(\llbracket\tau\rrbracket.\textsc{shr}(\kappa, t, \ell') * [\kappa]_q\big)\Big)
\end{array}\right\}$$

$$\llbracket \&^{\kappa}_{\textbf{mut}}\ \tau \rrbracket := \left\{\begin{array}{l}
\textsc{size} := 1, \\
\textsc{own} := \lambda t, \bar{v}.\, \exists \ell.\, \bar{v} = [\ell] * \&^{\llbracket\kappa\rrbracket}_{\mathsf{full}}\big(\exists \overline{w}.\, \ell \mapsto \overline{w} * \llbracket\tau\rrbracket.\textsc{own}(t, \overline{w})\big), \\
\textsc{shr} := \lambda \kappa', t, \ell.\, \exists \ell'.\, \&^{\kappa'}_{\mathsf{frac}}(\lambda q.\, \ell \overset{q}{\mapsto} \ell') * \\
\qquad\qquad \square\Big(\forall q.\, [\llbracket\kappa\rrbracket \sqcap \kappa']_q \twoheadrightarrow {}^{\mathcal{N}_{\mathsf{shr}} \cup \mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{lft}}} \triangleright\, {}^{\mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{shr}} \cup \mathcal{N}_{\mathsf{lft}}} \\
\qquad\qquad\qquad \big(\llbracket\tau\rrbracket.\textsc{shr}(\llbracket\kappa\rrbracket \sqcap \kappa', t, \ell') * [\llbracket\kappa\rrbracket \sqcap \kappa']_q\big)\Big)
\end{array}\right\}$$

$$\llbracket \&^{\kappa}_{\textbf{shr}}\ \tau \rrbracket := \mathsf{SimpleType}(\lambda t, v.\, \exists \ell.\, v = \ell * \llbracket\tau\rrbracket.\textsc{shr}(\llbracket\kappa\rrbracket, t, \ell))$$

$$\llbracket \Pi\bar{\tau} \rrbracket := \left\{\begin{array}{l}
\textsc{size} := \displaystyle\sum_i \llbracket\bar{\tau}_i\rrbracket.\textsc{size}, \\
\textsc{own} := \lambda t, \bar{v}.\, \exists \bar{\bar{v}}.\, \bar{v} = \displaystyle\sum_i \bar{\bar{v}}_i * \mathop{\Large\ast}_i \llbracket\bar{\tau}_i\rrbracket.\textsc{own}(t, \bar{\bar{v}}_i), \\
\textsc{shr} := \lambda \kappa, t, \ell.\, \displaystyle\mathop{\Large\ast}_i \llbracket\bar{\tau}_i\rrbracket.\textsc{shr}(\kappa, t, \ell + \textstyle\sum_{j<i} \llbracket\bar{\tau}_j\rrbracket.\textsc{size})
\end{array}\right\}$$

$$\llbracket \Sigma\bar{\tau} \rrbracket := \left\{\begin{array}{l}
\textsc{size} := 1 + \max\limits_i \llbracket\tau_i\rrbracket.\textsc{size}, \\
\textsc{own} := \lambda t, \bar{v}.\, \exists i, \bar{v}', \bar{v}_{\text{pad}}.\, \bar{v} = [i] \mathbin{+\!\!+} \bar{v}' \mathbin{+\!\!+} \bar{v}'' * \llbracket\bar{\tau}_i\rrbracket.\textsc{own}(t, \bar{v}') * \\
\qquad\qquad |\bar{v}_{\text{pad}}| = 1 + \max\limits_j \llbracket\bar{\tau}_j\rrbracket.\textsc{size}, \\
\textsc{shr} := \lambda \kappa, t, \ell.\, \exists i.\, \llbracket\bar{\tau}_i\rrbracket.\textsc{shr}(\kappa, t, \ell + 1) * \&^{\kappa}_{\mathsf{frac}}\big(\lambda q.\, \ell \overset{q}{\mapsto} i * \\
\qquad\qquad (\exists \bar{v}_{\text{pad}}.\, \ell + 1 + \llbracket\bar{\tau}_i\rrbracket.\textsc{size} \overset{q}{\mapsto} \bar{v}_{\text{pad}} * \llbracket\bar{\tau}_i\rrbracket.\textsc{size} + |\bar{v}_{\text{pad}}| = \max\limits_j \llbracket\bar{\tau}_j\rrbracket.\textsc{size})\big)
\end{array}\right\}$$

$$\llbracket \lightning_1 \rrbracket := \mathsf{SimpleType}(\lambda\_, \_.\, \mathsf{True})$$

$$\llbracket \lightning_n \rrbracket := \Pi[\lightning_1, \ldots, \lightning_1] \qquad \text{of length } \llbracket n \rrbracket$$

$$\begin{array}{l}
\llbracket \forall\bar{\alpha}.\, \textbf{fn}(\digamma : \mathbf{E}; \bar{\tau}) \to \tau \rrbracket := \mathsf{SimpleType}\Big(\lambda\_, v.\, \exists f, \bar{x}, k, F.\, v = \texttt{funrec}\, f(\bar{x})\, \texttt{ret}\, k := F * \triangleright \forall \bar{\kappa}, \kappa_{\digamma}, v_k, \bar{v}. \\
\qquad \square\big(\mathbf{E}; \digamma \sqsubseteq_{\mathsf{l}} [] \mid k \lhd \textbf{cont}(\digamma \sqsubseteq_{\mathsf{l}} []; x.\, x \lhd \textbf{own}\ \tau); \bar{x} \mathrel{\overline{\lhd}} \textbf{own}\ \bar{\tau} \models F[\texttt{funrec}\, f(\bar{x})\, \texttt{ret}\, k := F/f, \bar{v}/\bar{x}, v_k/k]\big)\Big)
\end{array}$$

$$\llbracket \mu T.\, \tau \rrbracket := \mathit{fix}(\lambda T.\, \llbracket\tau\rrbracket)$$

where

$$\mathsf{SimpleType}(\varPhi) := \left\{\begin{array}{l}
\textsc{size} := 1, \\
\textsc{own} := \lambda t, \bar{v}.\, \exists v.\, \bar{v} = [v] * \varPhi(t, v), \\
\textsc{shr} := \lambda \kappa, t, \ell.\, \exists v.\, \&^{\kappa}_{\mathsf{frac}}(\lambda q.\, \ell \overset{q}{\mapsto} v) * \triangleright \varPhi(t, v)
\end{array}\right\}$$

Figure 12.5: Interpretations of primitive types and type constructors.

the assumptions we are granted in that proof:

$$\&^{\kappa}_{\mathsf{full}}\Big(\exists \ell'.\, \ell \mapsto \ell' * \triangleright\big(\exists \overline{w}.\, \ell' \mapsto \overline{w} * [\![\tau]\!].\textsc{own}(t, \overline{w})\big) * \ldots\Big)$$

The first points-to permission ($\mapsto$) comes from the statement of ty-share, the second is from the definition of $[\![\mathbf{own}_n\, \tau]\!]$.own. We omitted the handling of the deallocation permission. After some amount of freezing and splitting of the borrow (LftL-bor-exists and LftL-bor-split), we arrive at:

$$\ldots * \&^{\kappa}_{\mathsf{full}} \triangleright\big(\exists \overline{w}.\, \ell' \mapsto \overline{w} * [\![\tau]\!].\textsc{own}(t, \overline{w})\big) * \ldots$$

At this point, to make progress in the proof, we need to use ty-share for τ to start sharing of the data the owned pointer points to. Given our sharing predicate above, which says that a shared owned pointer points to a shared τ, this is exactly what we would expect: when we start sharing the owned pointer, we have to recursively start sharing what it points to.

The trouble is that we cannot use ty-share here. If we compare what we have with what ty-share needs, we can see that there is a $\triangleright$ in the way: if we had a way to strip the $\triangleright$, we could complete the proof, but sadly that is impossible. Fundamentally, this is because $\triangleright$ does not commute with the update modality (*i.e.*, $\triangleright \dot{\Rrightarrow} P$ does not imply $\dot{\Rrightarrow} \triangleright P$).

The solution, then, is to "delay" executing ty-share: *delayed sharing.* We replace $\triangleright [\![\tau]\!].\textsc{shr}(\kappa, t, \ell')$ by

$$\Box\Big(\forall q.\, [\kappa]_q \mathrel{-\!\!*} {}^{\mathcal{N}_{\mathsf{shr}} \cup \mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{lft}}} \triangleright\, {}^{\mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{shr}} \cup \mathcal{N}_{\mathsf{lft}}} \big([\![\tau]\!].\textsc{shr}(\kappa, t, \ell') * [\kappa]_q\big)\Big)$$

This "view shift that takes a step" expresses that at any time, given a proof that κ is still alive, we can obtain the inner sharing *one step later.* Where $\triangleright [\![\tau]\!].\textsc{shr}(\kappa, t, \ell')$ says that we *own* the sharing predicate for τ (after a step of computation), the delayed sharing pattern says that *we can run some updates* (that take a step of computation) to own the sharing predicate for τ.[10] The actual sharing of τ (*i.e.*, applying ty-share) happens when these updates are run for the first time. Effectively, when we share a data structure with lots of nested owned references, sharing of the inner parts (the ones behind a pointer indirection) is done lazily. Only when someone actually accesses the inner pointer is the "view shift that takes a step" executed to start sharing the inner part.[11] At that point, the $\triangleright$ is no longer a problem because dereferencing that pointer requires the program to take a step.

Coming back to the proof of ty-share for $[\![\mathbf{own}_n\, \tau]\!]$, what we need to complete the proof is the following lemma to "initialize" delayed sharing:

$$\begin{aligned}&\&^{\kappa}_{\mathsf{full}} \triangleright\big(\exists \overline{w}.\, \ell' \mapsto \overline{w} * [\![\tau]\!].\textsc{own}(t, \overline{w})\big) \mathrel{-\!\!*} \\ &\Box\Big(\forall q.\, [\kappa]_q \mathrel{-\!\!*} {}^{\mathcal{N}_{\mathsf{shr}} \cup \mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{lft}}} \triangleright\, {}^{\mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{shr}} \cup \mathcal{N}_{\mathsf{lft}}} \big([\![\tau]\!].\textsc{shr}(\kappa, t, \ell') * [\kappa]_q\big)\Big)\end{aligned} \tag{own-share}$$

First of all, notice that, without the $\Box$, this proof would be trivial. We could introduce all our assumptions, open the full borrow with LftL-bor-acc-strong, then strip away the $\triangleright$ from both the assumption and the goal (we may take a step, after all), close the borrow again *without*

[10] This is similar to the difference in programming between having some data and having a thunk that can compute the data on-demand.

[11] We will see this in action in Figure 12.11 on page 203.

the later in the borrowed proposition, and finish by applying ty-share. However, the $\Box$ modality forces us to throw away non-persistent assertions before we go on introducing assumptions ($\Box$-intro on page 69). And this makes sense: after all, sharing is persistent for a reason! The Rust equivalent of $\&_{\mathsf{shr}}\, \mathbf{own}\ \tau$ is the type `&Box<T>`, which is duplicable, and whoever accesses it first will be the one to trigger the "delayed sharing" of the inner content. We thus have to arrange a little protocol making sure that even if two threads race for sharing the inner content, both threads have access to the resources they need.

Roughly speaking, the plan to prove own-share is as follows: we create an invariant with content

$$I := [\![\tau]\!].\textsc{shr}(\kappa, t, \ell') \lor \&^{\kappa}_{\mathsf{full}} \triangleright \big(\exists \overline{w}.\ \ell' \mapsto \overline{w} * [\![\tau]\!].\textsc{own}(t, \overline{w})\big)$$

Clearly, I initially holds, as we can easily satisfy the right conjunct. Because invariants are persistent, the $\Box$ is no longer a problem. For a moment, let us assume we could open this invariant and obtain I (rather than $\triangleright I$, which is what we actually get instead). We now distinguish two cases:

1. I is in the left conjunct. In this case, somebody else already started the sharing (performed the steps described in the second case); all we have to do is make a copy of the $[\![\tau]\!].\textsc{shr}(\kappa, t, \ell')$ that they put into the invariant and be done.

2. I is in the right conjunct. This means we are responsible for initiating sharing of this instance of τ. In this case, we proceed as sketched above, when we ignored the $\Box$. The key step is described by the following lemma:

   $$\&^{\kappa}_{\mathsf{full}} \triangleright P * [\kappa]_q \mathrel{-\!\!*} \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} \triangleright \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} \Big(\&^{\kappa}_{\mathsf{full}}\, P * [\kappa]_q\Big) \qquad \text{(LftL-bor-later)}$$

   This can be proven with LftL-bor-acc-strong by opening the borrow (obtaining $\triangleright \triangleright P$), taking a step (which removes a $\triangleright$ from the goal, but also means we now own $\triangleright P$), and then picking $Q := P$ when using the closing part of LftL-bor-acc-strong. After this, we can run ty-share, obtaining $[\![\tau]\!].\textsc{shr}(\kappa, t, \ell')$ which we use to satisfy I again (this time in the left conjunct) and to satisfy the conclusion of own-share.[12]

The actual proof is somewhat more complicated because we do not actually obtain I from our invariant, we obtain $\triangleright I$. The first case still works easily: after all, own-share concludes in a "view shift that takes a step", so we can use that step to strip away the later from our $\triangleright [\![\tau]\!].\textsc{shr}(\kappa, t, \ell')$. The second case is the problematic one, and to fix it, we have to actually change I. We have to find a way to put the full borrow into I without adding an extra later in front of it. This is exactly what *indexed borrows* (§11.5) let us do: LftL-bor-idx says that every full borrow can be split into a *persistent part* (the "indexed borrow") and a *timeless part* (the "token"). The timeless part can be put into the invariant without a $\triangleright$ being added. Thus, I looks as follows:

$$I_i := [\![\tau]\!].\textsc{shr}(\kappa, t, \ell') \lor [\mathsf{Bor} : i]_1$$

[12] Due to the way the masks are written in own-share, we are allowed to *first* change the mask from $\mathcal{N}_{\mathsf{shr}} \cup \mathcal{N}_{\mathsf{lft}}$ to $\mathcal{N}_{\mathsf{lft}}$, *then* apply the step-taking LftL-bor-later, and *then* close the invariant again, *i.e.*, we can keep the invariant open for the entire step. That is why $\mathcal{N}_{\mathsf{shr}}$ only occurs in the first and last mask, not in the two intermediate ones.

The first step in proving own-share is to show that the full borrow can be turned into the following:

$$\boxed{I_i}^{\mathcal{N}_{\mathsf{shr}}} * \&_i^{\kappa} \triangleright\big(\exists \overline{w}.\, \ell' \mapsto \overline{w} * [\![\tau]\!].\textsc{own}(t, \overline{w})\big)$$

These propositions are both persistent, so we can keep them and remove the $\square$ modality from our goal. The rest of the proof proceeds as before, except that the second case uses timelessness to remove the later in front of $[\mathsf{Bor} : i]_1$, and then uses LftL-bor-idx again to put the indexed borrow and the token back together and obtain the full borrow (*without* an extra $\triangleright$, *i.e.*, with just the $\triangleright$ inside the borrow that we started out with).

Delayed sharing for mutable references. Ownership of mutable references ($\&_{\mathsf{mut}}^{\kappa}\,\tau$) is very similar to that of owned pointers, except that of course everything is borrowed (and there is no deallocation permission). We have already seen this in §10.3.

Likewise, sharing of mutable references is *also* very similar to that of owned pointers. We use the same delayed sharing scheme for the same reason—the following "obvious" sharing predicate fails to satisfy ty-share:

$$[\![\&_{\mathsf{mut}}^{\kappa}\,\tau]\!].\textsc{shr}(\kappa', t, \ell) := \exists \ell'.\; \&_{\mathsf{frac}}^{\kappa}(\lambda q.\, \ell \overset{q}{\mapsto} \ell') * \triangleright [\![\tau]\!].\textsc{shr}([\![\kappa]\!] \sqcap \kappa', t, \ell')$$

The only difference to $[\![\mathsf{own}_n\,\tau]\!].\textsc{shr}$ is that mutable references have a lifetime parameter κ, which we take into account by saying that the referenced τ is shared for the *intersection* of both lifetimes in $\&_{\mathsf{shr}}^{\kappa'}\,\&_{\mathsf{mut}}^{\kappa}\,\tau$.

To prove ty-share, we need a lemma to initiate sharing of mutable references:

$$\begin{aligned}&\&_{\mathsf{full}}^{\kappa'}\,\&_{\mathsf{full}}^{\kappa}\big(\exists \overline{w}.\, \ell' \mapsto \overline{w} * [\![\tau]\!].\textsc{own}(t, \overline{w})\big) \mathrel{-\!\!*} \\ &\square\Big(\forall q.\, \big[[\![\kappa]\!] \sqcap \kappa'\big]_q \mathrel{-\!\!*} {}^{\mathcal{N}_{\mathsf{shr}} \cup \mathcal{N}_{\mathsf{lft}}}\!\!\Rrightarrow^{\mathcal{N}_{\mathsf{lft}}} \triangleright {}^{\mathcal{N}_{\mathsf{lft}}}\!\!\Rrightarrow^{\mathcal{N}_{\mathsf{shr}} \cup \mathcal{N}_{\mathsf{lft}}} \\ &\qquad\qquad\big([\![\tau]\!].\textsc{shr}([\![\kappa]\!] \sqcap \kappa', t, \ell') * \big[[\![\kappa]\!] \sqcap \kappa'\big]_q\big)\Big)\end{aligned}$$

(mut-share)

This lemma corresponds to own-share for owned pointers, and indeed the proof is very similar. We turn our assumptions into the following persistent resources in order to be able to use $\square$-intro:

$$\begin{gathered}\boxed{[\![\tau]\!].\textsc{shr}([\![\kappa]\!] \sqcap \kappa', t, \ell') \lor [\mathsf{Bor} : i]_1}^{\mathcal{N}_{\mathsf{shr}}} * \\ \&_i^{\kappa'}\,\&_{\mathsf{full}}^{\kappa}\big(\exists \overline{w}.\, \ell' \mapsto \overline{w} * [\![\tau]\!].\textsc{own}(t, \overline{w})\big)\end{gathered}$$

We perform the same case distinction as before, and there is no difference for the left conjunct. If we are in the right conjunct, instead of LftL-bor-later we use LftL-bor-unnest (page 155).

Products and sums. The product and sum types contain no major surprises. In §10.3 we saw the ownership and sharing predicates for pairs (binary products); this generalizes directly to n-ary products.

For sums ($\Sigma\overline{\tau}$), we define the representation in memory to be a single location storing the *tag* (indicating which variant of the sum is being used), followed by the data that matches the active variant, followed potentially by some junk data (typically called "padding") to make sure that the size of the sum is the same no matter the active variant. These components are called i, $\bar{v}'$ and $\bar{v}_{\mathrm{pad}}$ in the ownership predicate, respectively.

When a sum is shared, the tag is governed by a fractured borrow just like a simple type (so it can be read by anyone without synchronization, but cannot be mutated). At location $\ell + 1$, we expect a shared instance of the active variant i. And moreover, we also have a fractured borrow of the padding. The padding is just memory of the right length (regardless of its contents). We need this to be able to show that a sum is `Copy` when all its variants are: to make a copy, we have to be able to read the entire memory that stores the sum, including the padding.

Uninitialized memory ($\lightning_n$) is defined as being any memory of the right size. To simplify the verification of T-uninit-prod, we define uninitialized memory of size n as n times a single uninitialized location, but we also show the following theorem:

$$\text{UNINIT-OWN} \qquad [\![\lightning_n]\!].\mathrm{OWN}(t, \bar{v}) \ast\!\!\ast\ |\bar{v}| = n$$

Function types. A function is semantically well-typed when, given semantically well-typed arguments, it can be called safely and returns a semantically well-typed result—this is the characteristic function case of logical relations. Expressing this is somewhat cumbersome due to all the contexts that are involved, so we reuse the notion of a semantically well-typed function *body* which we will define in §12.4:[13] a function *value* is well-typed if its body is well-typed after substituting well-typed arguments.

Recursive types. The interpretation of recursive types ($\mu T.\, \tau$) is deceptively simple: τ contains T as a free variable, so it can be semantically interpreted as a function $\mathit{SemType} \to \mathit{SemType}$. To interpret the recursive type, we "just" take a fixed point of that function.

Of course, the devil is in the detail of why such a fixed point even should exist: as discussed in §10.3, well-definedness of recursive types rests on the type system restriction that each recursive use of T must be *guarded* by a pointer type. When interpreting types, we ensure that all pointer type constructors in turn *guard* their type arguments with the $\triangleright$ modality—this is why there is a $\triangleright$ in the interpretation of owned pointers and function types. For such functions whose recursive occurrences are guarded by a $\triangleright$, Iris guarantees the existence of a unique fixed point. We discussed the more subtle aspects of this at the end of §10.4.

Now that we have given semantic meaning to all λ_{Rust} types, the next step is to give semantic meaning to the typing *judgments*. However, before we can do that, we have to introduce the notion of *non-atomic invariants*: to verify `Cell` in §13.1, we need to make use of non-atomic invariants, and that means we need to prepare the semantic typing judgments to enable the use of these invariants.

[13] It may seem like this introduces a cycle into our semantic model, but that is not actually the case: semantically well-typed function bodies can be defined for typing contexts containing arbitrary semantic types, without referring to the specific interpretations we are defining here.

NAInv-persist
$$\mathsf{persistent}(\mathsf{NaInv}^{p.\mathcal{N}}(P))$$

NAInv-tok-timeless
$$\mathsf{timeless}([\mathsf{NaInv} : p.\mathcal{N}])$$

NAInv-new-pool
$$\mathsf{True} \Rrightarrow\!\!\ast_{\bot} \exists p.\ [\mathsf{NaInv} : p.\top]$$

NAInv-tok-split
$$[\mathsf{NaInv} : p.\mathcal{E}_1 \uplus \mathcal{E}_2] \mathrel{\ast\!\!\ast} [\mathsf{NaInv} : p.\mathcal{E}_1] \ast [\mathsf{NaInv} : p.\mathcal{E}_2]$$

NAInv-new-inv
$$\triangleright P \Rrightarrow\!\!\ast_{\mathcal{N}} \mathsf{NaInv}^{p.\mathcal{N}}(P)$$

NAInv-acc
$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\mathsf{NaInv}^{p.\mathcal{E}}(P) \mathrel{-\!\!\ast} \big([\mathsf{NaInv} : p.\mathcal{E}] \ \propto_{\mathcal{N}}\ [\mathsf{NaInv} : p.\mathcal{E} \setminus \mathcal{N}] \ast \triangleright P\big)}$$

Figure 12.6: Proof rules for non-atomic invariants.

12.3 Interlude: Non-atomic invariants and borrowing

To model `Cell`, we will have to define a sharing predicate that grants everyone mutable access to some area of memory. Usually we would of course use an Iris invariant to share ownership of this memory, but `Cell` actually uses *non-atomic* memory accesses, and invariants can only be used to reason about atomic instructions. On the other hand, we only need access to this state from within a single thread, while normal invariants are always accessible to all threads.[14] This combination of features—invariants that can be opened for many instructions but only from a single thread—is provided by *non-atomic* (or "thread-local") invariants.

Non-atomic invariants are very similar to the "atomic" Iris invariants that we have already seen in §5.1, except that opening them does not change the current mask. Instead, they come with their own tokens that are used to ensure that the same invariant is not opened twice. In fact, the mask attached to view shifts in Iris is also just syntactic sugar for very similar tokens:

$$^{\mathcal{E}_1}{\models\!\!\Rrightarrow}^{\mathcal{E}_2} P \approx \mathsf{InvTokens}(\mathcal{E}_1) \mathrel{-\!\!\ast} \dot{\models\!\!\Rrightarrow} \mathsf{InvTokens}(\mathcal{E}_2) \ast P$$

In other words, a (potentially mask-changing) fancy update with result P is something that *consumes* the right to access all invariants in $\mathcal{E}_1$, and then *produces* the right to access all invariants in $\mathcal{E}_2$.[15] That is all that there is to these mask annotations. With non-atomic invariants, we will have similar tokens, but we will not have the benefit of syntactic sugar for managing the tokens—we will pass them around explicitly everywhere,[16] which looks as follows:

$$P \ast [\mathsf{NaInv} : p.\mathcal{E}_1] \Rrightarrow\!\!\ast_{\mathcal{E}} [\mathsf{NaInv} : p.\mathcal{E}_2] \ast Q$$

Here, $[\mathsf{NaInv} : p.\mathcal{E}]$ claims ownership of the tokens for all invariants in *pool* p that match mask $\mathcal{E}$. Similar to how each thread in a concurrent program can have its own thread-local memory, non-atomic invariants also come with support for multiple "pools" that exist independently and whose masks and namespaces do not interact. The entire proposition above then describes a view shift which is non-mask-changing with regards to the "atomic" mask $\mathcal{E}$, but changes the non-atomic mask of pool p from $\mathcal{E}_1$ to $\mathcal{E}_2$.

[14] `Cell` would be unsound if it permitted non-atomic mutation of state that is shared across multiple threads: there would be data races.

[15] Here, $\dot{\models\!\!\Rrightarrow}$ is the "basic update modality" of the Iris base logic, and the definition also omits some other important aspects of fancy updates. For all the details, see "Iris from the ground up: A modular foundation for higher-order concurrent separation logic" [Jun+18b].

[16] We felt that having two masks at every view shift and Hoare triple ("normal" invariants and non-atomic invariants), and four for mask-changing view shifts, would just become too confusing.

LftL-bor-na

$$\&^{\kappa}_{\mathsf{full}}\, P \mathrel{\Rrightarrow\!\!\ast}_{\mathcal{N}} \&^{\kappa/p.\mathcal{N}}_{\mathsf{na}}\, P$$

LftL-na-shorten

$$\frac{\kappa' \sqsubseteq \kappa \qquad \mathcal{N} \sqsubseteq \mathcal{N}'}{\&^{\kappa/p.\mathcal{N}}_{\mathsf{na}}\, P \mathrel{-\!\!*} \&^{\kappa'/p.\mathcal{N}'}_{\mathsf{na}}\, P}$$

LftL-na-persist

$$\mathsf{persistent}(\&^{\kappa/p.\mathcal{N}}_{\mathsf{na}}\, P)$$

LftL-na-acc

$$\&^{\kappa/p.\mathcal{N}}_{\mathsf{na}}\, P \mathrel{-\!\!*} \big([\kappa]_q * [\mathsf{NaInv} : p.\mathcal{N}] \;\; \propto_{\mathcal{N}_{\mathsf{lft}},\mathcal{N}} \;\; \triangleright P\big)$$

Figure 12.7: Proof rules for non-atomic borrows.

We show the proof rules for non-atomic invariants in Figure 12.6. The key proposition is $\mathsf{NaInv}^{p.\mathcal{N}}(P)$, which states that in namespace $\mathcal{N}$ of pool p, we maintain invariant P. NAInv-new-pool creates a new pool and returns the token for *all* invariants in that pool. NAInv-tok-split establishes that separating conjunction of tokens for the same pool acts like disjoint union of the token sets. With NAInv-new-inv we can allocate a new invariant. And finally, the accessor NAInv-acc lets us open a non-atomic invariant. This accessor is non-mask-changing for the "atomic" mask, which means the invariant can remain open arbitrarily long, but it *does* change the current "non-atomic mask" from $\mathcal{E}$ to $\mathcal{E} \setminus \mathcal{N}$ as reflected by the token.

In the Iris technical appendix,[17] we explain how non-atomic invariants can be encoded in Iris in terms of invariants and some ghost state.

Non-atomic borrows. To actually use non-atomic invariants in the sharing predicate of `Cell`, we need to integrate them into the lifetime logic. After all, the sharing predicate describes sharing *for a lifetime*, so we need to be able to have non-atomic invariants that only hold while a lifetime is ongoing—we need *non-atomic* ("thread-local") *borrows.*

These non-atomic borrows are defined in terms of non-atomic invariants in exactly the same way we defined atomic borrows (§11.4) in terms of (atomic) invariants (§11.5):[18]

$$\&^{\kappa/p.\mathcal{N}}_{\mathsf{na}}\, P := \exists i.\ \&^{\kappa}_{i}\, P * \mathsf{NaInv}^{p.\mathcal{N}}([\mathsf{Bor} \,:\, i]_1) \qquad (\textsc{na-bor})$$

From this, we can derive the rules shown in Figure 12.7. As usual, non-atomic borrows can be created from full borrows (LftL-bor-na), can be shortened (LftL-na-shorten), and are persistent (LftL-na-persist). The latter rule can also be used to weaken the namespace of a borrow. And finally, the accessor LftL-na-acc lets us open the borrow given some fraction of the borrow token. Doing so leaves the atomic mask unchanged at $\mathcal{N}_{\mathsf{lft}}, \mathcal{N}$ but removes the borrow's namespace $\mathcal{N}$ from the non-atomic "mask" by consuming the corresponding token.

12.4 Semantically modeling λ_{Rust} typing judgments

At last, we have all the ingredients in place to define what the λ_{Rust} typing judgments semantically *mean*, and to verify that the syntactic rules given in §9.3 are compatible with that meaning. The semantic model of λ_{Rust} typing is defined in §12.8 (page 198).[19]

[17] Iris Team, "The Iris 3.2 documentation", 2019 [Iri19], §10.2.

[18] The only difference is that this time, we do not need to demand $\mathcal{N}$ to be disjoint from $\mathcal{N}_{\mathsf{lft}}$.

[19] To streamline the presentation, we omit the tracking of free variables in our semantic interpretations. Formally speaking, *every single judgment* is indexed by a context γ which maps free variables to their interpretation. This context is carried around everywhere and extended when descending below a binder. However, this adds a lot of clutter, and nothing interesting is happening there. Correspondingly, we will also omit the variable-binding context Γ from the type system judgments.

Marker traits. First, we interpret the judgments $\vdash \tau$ copy, $\vdash \tau$ send, and $\vdash \tau$ sync, which correspond to the Rust marker traits `Copy`, `Send`, and `Sync`, respectively.

Notice how the fact that Rust has two separate traits for thread safety of owned and shared data (`Send` and `Sync`) nicely corresponds to our semantic types having an ownership and a sharing predicate: `Send` corresponds to $\models \tau$ send,[20] which requires that the *ownership* predicate is independent of the thread identifier (and thus easily transferable from one thread to another); `Sync` corresponds to $\models \tau$ sync, which requires that the *sharing* predicate is independent of the thread identifier.

The semantic interpretation of `Copy` in $\models \tau$ copy is a bit more intricate. First of all we require the ownership predicate to be persistent; that part is easy. The second part of $\models \tau$ copy is needed to justify TREAD-BOR for shared references: we need to be able to make a copy of the data the shared reference points to. This is written as a symmetric accessor. If we ignore the non-atomic tokens for a moment, the condition looks as follows:

$$[\![\tau]\!].\textsc{shr}(\kappa, t, \ell) \mathrel{-\!\!*} \left([\kappa]_q \;\; \propto_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}} \;\; q'.\, \exists \bar{v}.\, \ell \xmapsto{q'} \bar{v} * \triangleright [\![\tau]\!].\textsc{own}(t, \bar{v})\right)$$

In other words, by giving up a fraction of the corresponding lifetime token we can get temporary, non-atomic read-only access to the memory behind ℓ, and we know that the data stored there satisfies the ownership predicate of τ. That ownership predicate is persistent, so we can keep a copy of it along with the copy of the data, and still close the accessor again.

However, we would like `Cell` to be `Copy`.[21] The sharing predicate of `Cell` will make use of non-atomic borrows, which means we need to add non-atomic tokens to the accessors so that those borrows can be used (LFTL-NA-ACC). Now the easiest solution would be to add $[\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}]$ to the left-hand side of the accessor, letting it open all non-atomic invariants in $\mathcal{N}_{\mathsf{shr}}$, where $\mathcal{N}_{\mathsf{shr}}$ is a namespace that we designate for the use of sharing predicates. But that would not work: we want $\tau_1 \times \tau_2$ to be `Copy` if both τ_1 and τ_2 are `Copy`. Proving the above accessor for $\tau_1 \times \tau_2$ will boil down to using the accessors for *both* τ_1 and τ_2 and *keeping them open at the same time* so that we can produce the permission to read the entire product. If opening the first accessor consumes all our non-atomic tokens, we cannot use the second accessor and are stuck.

To solve this, we separate the namespace $\mathcal{N}_{\mathsf{shr}}$ further into one sub-namespace $\mathcal{N}_{\mathsf{shr}}.\ell$ *per location*.[22] The starting mask of the shared reference accessor grants access to the locations in $[\geq\ell, <\ell + 1 + [\![\tau]\!].\textsc{size}]$, and the ending mask requires the last one of them ($\ell + [\![\tau]\!].\textsc{size}$) to still be available. It would be simpler to put $[\geq\ell, <\ell + [\![\tau]\!].\textsc{size}]$ on the left-hand side and leave it at that (having no token on the right-hand side of the symmetric accessor), but that would mean that types of size 0 would not get access to any non-atomic invariants at all, which would not work. Thus we provide the accessor with *temporary* access to one additional location—temporary because all borrows/invariants associated with that location must be closed again before the accessor produces its result. As we will see, this is sufficient for zero-sized `Cell`.

[20] In this presentation, we restrict the semantic interpretations of type system judgments to syntactic types τ. In the full formalization, the interpretation can be used on any semantic type (*i.e.*, any element of *SemType*), but here we stick to syntactic types only to avoid confusion between the two kinds of types (and because there is no good free metavariable left for semantic types).

[21] `Cell` is currently *not* `Copy` in Rust, but this is due to ergonomic concerns, not soundness reasons: accidentally copying a `Cell` could be quite confusing. The Rust language team believes that it would be sound to make `Cell` `Copy`, and indeed our proof shows that they are right.

[22] Any countable type can be used to form a namespace in Iris.

Next, we have to verify that the rules for $\vdash \tau$ copy given in §9.4 hold for $\models \tau$ copy (and similarly for the other two judgments). We do not show these proofs here, as they are not very interesting.

Typing contexts. We already interpreted the external and local lifetime contexts in Figure 12.1; here we discuss the two remaining contexts: the typing context $\mathbf{T}$ and the continuation context $\mathbf{K}$. We also have to define how the paths occurring in typing contexts are to be interpreted.

Note that type and continuation context are interpreted relative to a thread identifier t, just like types themselves (as we saw in §10). In the type system proof, we will create one non-atomic "pool" for each thread of the program and use the pool name p as thread identifier t. All typing judgments will thread through the tokens for the non-atomic pool of the current thread.

For the purpose of the type system, a path p denotes some value of λ_{Rust}. Note that paths themselves are *expressions* of λ_{Rust}, but not values. These expressions are related to the values they denote by the following key lemma:

wp-sem-path

$$\mathsf{wp}\, p\, \{v.\, v = [\![p]\!]\}$$

In other words, the value denoted by the path is what the path reduces to when being evaluated.

The interpretation of a *typing context* $\mathbf{T}$ is a big separating conjunction of each type assignment in the context. The individual type assignments $p \lhd \tau$ are interpreted using $[\![\tau]\!]$.own, the ownership predicate of τ. A *blocked* type assignment $p \lhd^{\dagger\kappa} \tau$ corresponds to an *inheritance* of the lifetime logic.[23]

Each element $k \lhd \mathbf{cont}(\mathbf{L}; \bar{x}.\, \mathbf{T})$ of the *continuation context* $\mathbf{K}$ represents some continuation k that we can call. This is expressed in Iris as a weakest precondition: under the assumption that the type and lifetime context given in the type are satisfied, the given code is safe to execute.[24] Note that the continuation also has access to $[\mathsf{NaInv} : t.\top]$, which are all the non-atomic tokens for the pool of thread t. Remarkably, the continuation context is a big *overlapping* ("normal") *conjunction* of its elements. This reflects the fact that in F-letcont, both the named continuation F' and the body F that will continue execution can make use of the same continuation context $\mathbf{K}$, implying that all continuations can share their resources. This sharing does not cause problems because only one of the continuations can actually be used—when jumping to a continuation, all the other continuations we locally had available stop mattering.

Subtyping judgments. The last component of the type system that we have to interpret are the typing judgments themselves, including our various helper judgments. We begin with our subtyping and coercion judgments.

For this, we first need to define what "subtyping" means for semantic types, which is captured by $\tau_1 \sqsubseteq^{\mathsf{ty}} \tau_2$. We follow the usual approach that for τ_1 to be a subtype of τ_2, we must have that τ_1 is a "subset" of τ_2 (if we think of semantic types as sets of values), which translates to our

[23] It may be surprising that there is no $\triangleright$ showing up here, like it does in LftL-borrow. This is because borrowing always happens at a pointer type (C-borrow), and we can use the $\triangleright$ that is inherent in the interpretation of pointer types: borrowing a proposition that is *already* of the shape $\triangleright P$ does not add another $\triangleright$ (just as with invariants).

[24] Continuations do not return, so the postcondition is trivial.

Semantic model of marker trait judgments.

$$\begin{aligned}
\models \tau\ \mathsf{send} &:= \Box\, \forall t_1, t_2, \bar{v}.\ \llbracket \tau \rrbracket.\textsc{own}(t_1, \bar{v}) \mathrel{-\!\!*} \llbracket \tau \rrbracket.\textsc{own}(t_2, \bar{v}) \\
\models \tau\ \mathsf{sync} &:= \Box\, \forall \kappa, t_1, t_2, \ell.\ \llbracket \tau \rrbracket.\textsc{shr}(\kappa, t_1, \ell) \mathrel{-\!\!*} \llbracket \tau \rrbracket.\textsc{shr}(\kappa, t_2, \ell) \\
\models \tau\ \mathsf{copy} &:= (\forall t, \bar{v}.\ \mathsf{persistent}(\llbracket \tau \rrbracket.\textsc{own}(t, \bar{v}))) \land \\
&\qquad \Box\, \forall \kappa, t, \ell, q.\ \llbracket \tau \rrbracket.\textsc{shr}(\kappa, t, \ell) \mathrel{-\!\!*} \Big([\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.[\geq \ell, < \ell + \llbracket \tau \rrbracket.\textsc{size} + 1]] * [\kappa]_q \ \propto_{\mathcal{N}_{\mathsf{shr}}, \mathcal{N}_{\mathsf{lft}}} \\
&\qquad\qquad q'.\ \exists \bar{v}.\ [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.(\ell + \llbracket \tau \rrbracket.\textsc{size})] * \ell \xmapsto{q'} \bar{v} * \triangleright \llbracket \tau \rrbracket.\textsc{own}(t, \bar{v}) \Big)
\end{aligned}$$

Semantic model of paths and typing contexts.

$$\llbracket p \rrbracket : \mathit{Val} \qquad\qquad \llbracket x \rrbracket := x \qquad\qquad \llbracket p.n \rrbracket := \llbracket p \rrbracket + n$$

$$\begin{aligned}
\llbracket \mathbf{T} \rrbracket &: \mathit{TId} \to \mathit{iProp} \\
\llbracket \emptyset \rrbracket(t) &:= \mathsf{True} \\
\llbracket \mathbf{T}, p \lhd \tau \rrbracket(t) &:= \llbracket \tau \rrbracket.\textsc{own}(t, [\llbracket p \rrbracket]) * \llbracket \mathbf{T} \rrbracket(t) \\
\llbracket \mathbf{T}, p \lhd^{\dagger\kappa} \tau \rrbracket(t) &:= \big([\dagger \llbracket \kappa \rrbracket] \Rrightarrow\!\!*_{\top}\ \llbracket \tau \rrbracket.\textsc{own}(t, [\llbracket p \rrbracket]) \big) * \llbracket \mathbf{T} \rrbracket(t)
\end{aligned}$$

$$\begin{aligned}
\llbracket \mathbf{K} \rrbracket &: \mathit{TId} \to \mathit{iProp} \\
\llbracket \emptyset \rrbracket(t) &:= \mathsf{True} \\
\llbracket \mathbf{K}, k \lhd \mathbf{cont}(\mathbf{L}; \bar{x}.\ \mathbf{T}) \rrbracket(t) &:= \big(\forall \bar{v}.\ [\mathsf{NaInv} : t.\top] * \llbracket \mathbf{L} \rrbracket(1) * \llbracket \mathbf{T} \rrbracket(t) \mathrel{-\!\!*} \mathsf{wp}\ \llbracket k \rrbracket(\bar{v})\ \{\mathsf{True}\} \big) \land \llbracket \mathbf{K} \rrbracket(t)
\end{aligned}$$

Semantic model of typing judgments.

$$\begin{aligned}
\tau_1 \sqsubseteq^{\mathsf{ty}} \tau_2 &:= \llbracket \tau_1 \rrbracket.\textsc{size} = \llbracket \tau_2 \rrbracket.\textsc{size} * \\
&\qquad (\Box\, \forall t, \bar{v}.\ \llbracket \tau_1 \rrbracket.\textsc{own}(t, \bar{v}) \mathrel{-\!\!*} \llbracket \tau_2 \rrbracket.\textsc{own}(t, \bar{v})) * \\
&\qquad (\Box\, \forall \kappa, t, \ell.\ \llbracket \tau_1 \rrbracket.\textsc{shr}(\kappa, t, \ell) \mathrel{-\!\!*} \llbracket \tau_2 \rrbracket.\textsc{shr}(\kappa, t, \ell)) \\
\mathbf{E}; \mathbf{L} \models \tau_1 \Rightarrow \tau_2 &:= \Box\, \forall q.\ \llbracket \mathbf{L} \rrbracket(q) \mathrel{-\!\!*} \Box(\llbracket \mathbf{E} \rrbracket \mathrel{-\!\!*} \tau_1 \sqsubseteq^{\mathsf{ty}} \tau_2) \\
\mathbf{E}; \mathbf{L} \models \mathbf{T}_1 \Rightarrow \mathbf{T}_2 &:= \Box\, \forall t, q.\ \llbracket \mathbf{E} \rrbracket * \llbracket \mathbf{L} \rrbracket(q) * \llbracket \mathbf{T}_1 \rrbracket(t) \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}}\ \llbracket \mathbf{L} \rrbracket(q) * \llbracket \mathbf{T}_2 \rrbracket(t) \\
\models \mathbf{T}_1 \Rightarrow^{\dagger\kappa} \mathbf{T}_2 &:= \Box\, \forall t.\ [\dagger\kappa] * \llbracket \mathbf{T}_1 \rrbracket(t) \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}}\ \llbracket \mathbf{T}_2 \rrbracket(t) \\
\mathbf{E} \models \mathbf{K}_1 \Rightarrow \mathbf{K}_2 &:= \Box\, \forall t.\ \llbracket \mathbf{E} \rrbracket * \llbracket \mathbf{K}_1 \rrbracket(t) \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}}\ \llbracket \mathbf{K}_2 \rrbracket(t)
\end{aligned}$$

$$\begin{aligned}
\mathbf{E}; \mathbf{L} \models \tau_1 \multimap^{\tau} \tau_2 &:= \Box\, \forall \ell, t, q.\ \llbracket \mathbf{E} \rrbracket * \llbracket \mathbf{L} \rrbracket(q) * \llbracket \tau_1 \rrbracket.\textsc{own}(t, [v]) \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}} \\
&\qquad \exists \ell, \bar{v}.\ \ell = v * |\bar{v}| = \llbracket \tau \rrbracket.\textsc{size} * \ell \mapsto \bar{v} * \\
&\qquad\quad \big(\forall \bar{v}'.\ \ell \mapsto \bar{v}' * \triangleright \llbracket \tau \rrbracket.\textsc{own}(t, \bar{v}') \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}}\ \llbracket \mathbf{L} \rrbracket(q) * \llbracket \tau_2 \rrbracket.\textsc{own}(t, v) \big) \\
\mathbf{E}; \mathbf{L} \models \tau_1 \multimapinv^{\tau} \tau_2 &:= \Box\, \forall \ell, t, q.\ \llbracket \mathbf{E} \rrbracket * \llbracket \mathbf{L} \rrbracket(q) * [\mathsf{NaInv} : t.\top] * \llbracket \tau_1 \rrbracket.\textsc{own}(t, [v]) \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}} \\
&\qquad \exists \ell, \bar{v}, q'.\ \ell = v * \ell \xmapsto{q'} \bar{v} * \triangleright \llbracket \tau \rrbracket.\textsc{own}(t, \bar{v}) * \\
&\qquad\quad \big(\ell \xmapsto{q'} \bar{v} \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}}\ \llbracket \mathbf{L} \rrbracket(q) * [\mathsf{NaInv} : t.\top] * \llbracket \tau_2 \rrbracket.\textsc{own}(t, v) \big)
\end{aligned}$$

$$\begin{aligned}
\mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \models I \dashv x.\ \mathbf{T}_2 &:= \Box\, \forall t.\ \llbracket \mathbf{E} \rrbracket * \llbracket \mathbf{L} \rrbracket(1) * \llbracket \mathbf{T}_1 \rrbracket(t) * [\mathsf{NaInv} : t.\top] \mathrel{-\!\!*} \\
&\qquad \mathsf{wp}\ I\ \{x.\ \llbracket \mathbf{L} \rrbracket(1) * \llbracket \mathbf{T}_2 \rrbracket(t) * [\mathsf{NaInv} : t.\top]\} \\
\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \models F &:= \Box\, \forall t.\ \llbracket \mathbf{E} \rrbracket * \llbracket \mathbf{L} \rrbracket(1) * \llbracket \mathbf{K} \rrbracket(t) * \llbracket \mathbf{T} \rrbracket(t) * [\mathsf{NaInv} : t.\top] \mathrel{-\!\!*} \mathsf{wp}\ F\ \{\mathsf{True}\}
\end{aligned}$$

Figure 12.8: Semantic models of typing contexts and judgments.

logical approach as the predicate describing τ_1 implying the predicate describing τ_2—and in separation logic, the implication turns into a magic wand. This must be the case for *both* the ownership and the sharing predicate (otherwise, we could not verify covariance of shared references as expressed by T-bor-shr on page 126).

On top of that, we also demand that both types have the same size. We have to do this to ensure that the product type is covariant (T-prod): the starting address of the second field depends on the size of the first, and that address must not change as subtyping is applied. When proving the inclusion of the sharing predicate of products, we cannot rely on ty-size, so size equality needs to be explicitly part of the subtyping relation. Most of the time, this is not actually a restriction: if τ_1 is non-empty, then ty-size implies that if the same list of values $\bar{v}$ satisfies the ownership predicates of τ_1 and τ_2, then the types both have size $|\bar{v}|$. However, a curious consequence of this size constraint is that the empty type is *not* a least element of the subtyping relation—in fact, no such element exists (nor is there a largest element).

Interpreting the *subtyping judgment* ($\mathbf{E}; \mathbf{L} \vdash \tau_1 \Rightarrow \tau_2$) in terms of semantic typing then works very similar to how we already interpreted the judgment for lifetime inclusion.

For *type coercion* ($\mathbf{E}; \mathbf{L} \vdash \mathbf{T}_1 \Rightarrow \mathbf{T}_2$), the key difference to subtyping (besides the fact that it acts on entire contexts, not individual types) is that we use a *view shift* instead of a magic wand. This is necessary to support C-share, which relies on ty-share so it needs to be a view shift.[25]

Type context unblocking ($\mathbf{T}_1 \Rightarrow^{\dagger\kappa} \mathbf{T}_2$) is very similar to a type coercion, except that we additionally may assume that lifetime κ is dead.

And finally, continuation coercions ($\mathbf{E} \vdash \mathbf{K}_1 \Rightarrow \mathbf{K}_2$) work just like type coercions.

Helper judgments for reads and writes. Next, we consider the two helper judgments that are used to type reading from and writing to memory. Both of these are (asymmetric) accessors.

The *writing judgment* ($\mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_2$) says that, given the contexts $\mathbf{E}$ and $\mathbf{L}$ as well as ownership of some value v at type τ_1, we can learn that v is actually a location ℓ and get access to ownership of the memory pointed to by ℓ, with enough space to put an instance of τ in that place ($|\bar{v}| = [\![\tau]\!].\textsc{size}$). The closing part of the accessor says that once we have written a valid instance $\bar{v}'$ of type τ to ℓ, we can get back the local lifetime context $\mathbf{L}$ as well as ownership of the original value v, now at type τ_2.[26]

The *reading judgment* ($\mathbf{E}; \mathbf{L} \vdash \tau_1 \multimapinv^{\tau} \tau_2$) is similar: given the lifetime contexts and ownership of some value v at type τ_1, we learn that v is a location ℓ and get access to *fractional* ownership of the memory at ℓ (because fractional ownership is sufficient for reading). We also learn that the data $\bar{v}$ at ℓ has type τ *later*.[27] This update is *non-atomic-mask-changing*: the update consumes all the non-atomic tokens of the current thread t, and only gives them back when the closing part of the accessor is invoked. To close the accessor again, we need to give back ownership of the unchanged underlying memory, but we do *not* need to give back the ownership predicate of τ—this is crucial for destructive reads that

[25] With view shifts, the trick for keeping assumptions around when the conclusion is persistent does not work any more:

$$P \equiv\!\!\!\ast \Box Q \nvdash P \equiv\!\!\!\ast P \ast \Box Q$$

For this reason, we have to repeat $[\![\mathbf{L}]\!](q)$ on both sides and do not use the $\Box$ modality like we did, *e.g.*, for subtyping.

[26] The external lifetime context $\mathbf{E}$ is persistent, so we do not need to get it back.

[27] The reading judgment will be used when actually reading from memory, and that read will perform the physical step necessary to get rid of the later (wp-step, page 75).

consume ownership of the data they read from and leave the memory effectively uninitialized (Tread-own-move). In exchange, we obtain not only the non-atomic tokens but also the local lifetime context and ownership of v, now at type τ_2.

Typing instructions and function bodies. With all contexts and helper judgments in place, we can now give meaning to the top-level typing judgments for individual instructions and entire function bodies (remember that the latter interpretation is also used to interpret function types in Figure 12.5).

Both of these judgments use the *weakest precondition* connective of Iris, which expresses that a piece of code must be safe to execute under certain conditions. In fact, we can equivalently write them both as Hoare triples:

$$\mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \models I \dashv x.\, \mathbf{T}_2 \quad \Longleftrightarrow \quad \{[\![\mathbf{E}]\!] * [\![\mathbf{L}]\!](1) * [\![\mathbf{T}_1]\!](t) * [\mathsf{NaInv} : t.\top]\} \;\; I \;\; \{x.\, [\![\mathbf{L}]\!](1) * [\![\mathbf{T}_2]\!](t) * [\mathsf{NaInv} : t.\top]\} \tag{sem-instr}$$

$$\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \models F \quad \Longleftrightarrow \quad \{[\![\mathbf{E}]\!] * [\![\mathbf{L}]\!](1) * [\![\mathbf{K}]\!](t) * [\![\mathbf{T}]\!](t) * [\mathsf{NaInv} : t.\top]\} \;\; F \;\; \{\mathsf{True}\} \tag{sem-fn-body}$$

The precondition is in each case given by interpreting all the relevant contexts. Additionally, we may access all the non-atomic borrows of the current thread t—crucially, this is the same t that is also used to interpret the type and continuation contexts. The thread identifier is universally quantified because the verified instruction/function must execute safely in *any* thread.

In the postcondition, instructions give back the local lifetime context and the non-atomic tokens, and they provide the changed typing context $\mathbf{T}_2$. For function bodies, the postcondition True is trivial due to continuation-passing style, which means that function bodies do not return (they invoke a continuation instead).

These definitions reduce verifying semantic well-typedness of a piece of code to standard Iris program verification.

12.4.1 Adequacy

Now that we have interpreted all our judgments, we can verify Theorem 5. So assume some f such that $\emptyset; \emptyset \mid \emptyset \models f \dashv x.\, x \lhd \mathbf{fn}() \rightarrow \Pi[]$. We have to show that when we execute f with a no-op continuation, no execution reaches a stuck state.

By sem-instr and the interpretation of function types in Figure 12.5, this unfolds to[28]

$$\forall k, \digamma.\, \emptyset; \digamma \sqsubseteq_{\mathsf{l}} [] \mid k \lhd \mathbf{cont}(\digamma \sqsubseteq_{\mathsf{l}} []; x.\, x \lhd \mathbf{own}\, ()); \emptyset \models \mathtt{call}\, f()\, \mathtt{ret}\, k$$

which in turn by sem-fn-body unfolds to:

$$\forall k, \digamma, t.\, \{[\![\digamma \sqsubseteq_{\mathsf{l}} []]\!](1) * [\![k \lhd \mathbf{cont}(\digamma \sqsubseteq_{\mathsf{l}} []; x.\, x \lhd \mathbf{own}\, ())]\!](t) * [\mathsf{NaInv} : t.\top]\} \;\; F \;\; \{\mathsf{True}\}$$

Following the statement of adequacy, we pick $k := (\mathtt{rec}\, k(x) := \text{☣})$. Using NAInv-new-pool, we can create some fresh thread identifier t and own all the non-atomic tokens for it, and with LftL-begin we can create the

[28] For demonstration purposes, we are glossing over some details here, such as $\mathsf{wp}\, v\, \{Q\}$ not actually being equivalent to $Q(v)$ (but rather to $\Rrightarrow Q(v)$), and we should substitute k into the body of f instead of using function application.

$\left\{\left((1)\ \Box\,\forall q.\, \llbracket \mathbf{E} \rrbracket \mathrel{-\!\!*} \left(\llbracket \mathbf{L} \rrbracket(q)\ \ \propto_{\mathcal{N}_{\mathsf{lft}},\mathcal{N}_{\mathsf{rust}}}\ \ q'.\, [\llbracket \kappa \rrbracket]_{q'}\right)\right)\right\}$

Goal: $\Box\,\forall t, q.\, \llbracket \mathbf{E} \rrbracket * \llbracket \mathbf{L} \rrbracket(q) * \llbracket p \lhd \&^{\kappa}_{\mathbf{mut}}\ \tau \rrbracket(t) \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}},\mathcal{N}_{\mathsf{rust}}} \llbracket \mathbf{L} \rrbracket(q) * \llbracket p \lhd \&^{\kappa}_{\mathbf{shr}}\ \tau \rrbracket(t)$

Introduce persistence modality ($\Box$-intro on page 69) and assumptions of view shift.

Instantiate (1) with q and $\llbracket \mathbf{E} \rrbracket$.

$\left\{\left(\llbracket \mathbf{L} \rrbracket(q)\ \ \propto_{\mathcal{N}_{\mathsf{lft}},\mathcal{N}_{\mathsf{rust}}}\ \ q'.\, [\llbracket \kappa \rrbracket]_{q'}\right) * \llbracket \mathbf{L} \rrbracket(q) * \llbracket p \lhd \&^{\kappa}_{\mathbf{mut}}\ \tau \rrbracket(t)\right\}_{\mathcal{N}_{\mathsf{lft}},\mathcal{N}_{\mathsf{rust}}}$

Open accessor (using up $\llbracket \mathbf{L} \rrbracket(q)$), get closing view shift $(\mathbf{2})$ $[\llbracket \kappa \rrbracket]_{q'} \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}},\mathcal{N}_{\mathsf{rust}}} \llbracket \mathbf{L} \rrbracket(q)$.

$\left\{[\llbracket \kappa \rrbracket]_{q'} * \llbracket p \lhd \&^{\kappa}_{\mathbf{mut}}\ \tau \rrbracket(t) * (2)\right\}_{\mathcal{N}_{\mathsf{lft}},\mathcal{N}_{\mathsf{rust}}}$

Unfold $\llbracket \&^{\alpha}_{\mathbf{mut}}\ \_ \rrbracket.\textsc{own}$ (mut-ref-own).

$\left\{[\llbracket \kappa \rrbracket]_{q'} * \&^{\llbracket \kappa \rrbracket}_{\mathsf{full}}\left(\exists \overline{w}.\, \llbracket p \rrbracket \mapsto \overline{w} * \llbracket \tau \rrbracket.\textsc{own}(t, \overline{w})\right) * (2)\right\}_{\mathcal{N}_{\mathsf{lft}},\mathcal{N}_{\mathsf{rust}}}$

Apply ty-share.

$\left\{[\llbracket \kappa \rrbracket]_{q'} * \llbracket \tau \rrbracket.\textsc{shr}(\llbracket \kappa \rrbracket, t, \llbracket p \rrbracket) * (2)\right\}_{\mathcal{N}_{\mathsf{lft}},\mathcal{N}_{\mathsf{rust}}}$

Apply (2).

$\{\llbracket \mathbf{L} \rrbracket(q) * \llbracket \tau \rrbracket.\textsc{shr}(\llbracket \kappa \rrbracket, t, \llbracket p \rrbracket)\}_{\mathcal{N}_{\mathsf{lft}},\mathcal{N}_{\mathsf{rust}}}$

Fold $\llbracket \&^{\alpha}_{\mathbf{shr}}\ \_ \rrbracket.\textsc{own}$ (shr-ref-own).

Goal: $\llbracket \mathbf{L} \rrbracket(q) * \llbracket p \lhd \&^{\kappa}_{\mathbf{shr}}\ \tau \rrbracket(t)$

Figure 12.9: Proof outline for C-share-sem.

lifetime $\digamma$ and own its token (which implies $\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(1)$). Verifying that k has type $\mathbf{cont}(\digamma \sqsubseteq_{\mathsf{l}} []; x.\, x \lhd \mathbf{own}\ ())$ is not very hard.

Now adequacy of the λ_{Rust} type system is a consequence of adequacy of the Iris program logic, which says that a verified program whose precondition is satisfied cannot get stuck and actually exhibits the behavior described by the specification.[29] All the hard work of relating our interpretation of function bodies to actual program executions was already done when we instantiated the Iris program logic for the λ_{Rust} language!

12.5 The fundamental theorem of λ_{Rust}

We have now completed the semantic interpretation of the λ_{Rust} type system. This means we are finally ready to verify Theorem 4: for any inference rule of the type system given in §9, when we replace all $\vdash$ by $\models$ (replacing syntactic judgments by their semantic interpretation), we have to prove the resulting Iris theorem. We will of course not do this for every single typing rule, but pick a few interesting ones.

Semantic correctness of C-share. This rule is responsible for turning a mutable reference into a shared reference. It has been mentioned several times as the motivation for ty-share, so it will not be surprising that the proof of C-share is mostly an exercise in rearranging things until we can apply ty-share. Concretely, we have to show:

C-share-sem

$$\frac{\mathbf{E}; \mathbf{L} \models \kappa\ \mathsf{alive}}{\mathbf{E}; \mathbf{L} \models p \lhd \&^{\kappa}_{\mathbf{mut}}\ \tau \Rightarrow p \lhd \&^{\kappa}_{\mathbf{shr}}\ \tau}$$

The proof outline is given in Figure 12.9. We basically just unfold everything, apply the liveness judgment to get a fraction of the lifetime token for κ, use that to apply ty-share, and fold everything back again.

[29] Jung *et al.*, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b], §6.4; Iris Team, "The Iris 3.2 documentation", 2019 [Iri19], §9.2.

$\big\{\big((\mathbf{1})\ n = \mathsf{size}(\tau)\big) * \llbracket\mathbf{E}\rrbracket * \llbracket\mathbf{L}\rrbracket(q) * [\mathsf{NaInv} : t.\top] * \llbracket\mathbf{own}_m\ \tau\rrbracket.\textsc{own}(t, [v])\big\}_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}}$

Unfold $\llbracket\mathbf{own}\ \_\rrbracket.\textsc{own}$ (own-ptr-own), use $\Rrightarrow$-timeless to remove some $\triangleright$.

$\{\llbracket\mathbf{L}\rrbracket(q) * [\mathsf{NaInv} : t.\top] * v = \ell * \ell \mapsto \overline{w} * \triangleright \llbracket\tau\rrbracket.\textsc{own}(t, \overline{w}) * \mathsf{Dealloc}(\ell, \llbracket\tau\rrbracket.\textsc{size}, n)\}_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}}$

By ty-size, we get $(\mathbf{2})\ |\overline{w}| = \mathsf{size}(\tau)$ (also using $\Rrightarrow$-timeless).

Goal: $\Rrightarrow_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}} \exists \ell, \bar{v}, q'.\ \ell = v * \ell \overset{q'}{\mapsto} \bar{v} * \triangleright \llbracket\tau\rrbracket.\textsc{own}(t, \bar{v}) *$
$\big(\ell \overset{q'}{\mapsto} \bar{v} \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}} \llbracket\mathbf{L}\rrbracket(q) * [\mathsf{NaInv} : t.\top] * \llbracket\mathbf{own}_m\ \lightning_n\rrbracket.\textsc{own}(t, v)\big)$

Introduce update modality. Instantiate $\ell := \ell$, $\bar{v} := \overline{w}$, $q' := 1$.

Discharge all separating conjuncts except the last.

$\{\llbracket\mathbf{L}\rrbracket(q) * [\mathsf{NaInv} : t.\top] * \mathsf{Dealloc}(\ell, \llbracket\tau\rrbracket.\textsc{size}, n)\}$

Goal: $\ell \mapsto \overline{w} \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}} \llbracket\mathbf{L}\rrbracket(q) * [\mathsf{NaInv} : t.\top] * \llbracket\mathbf{own}_m\ \lightning_n\rrbracket.\textsc{own}(t, v)$

Introduce assumption of magic wand.

$\{\llbracket\mathbf{L}\rrbracket(q) * [\mathsf{NaInv} : t.\top] * \mathsf{Dealloc}(\ell, \llbracket\tau\rrbracket.\textsc{size}, n) * \ell \mapsto \overline{w}\}_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}}$

By (1) and (2), $|\overline{w}| = n$. Use uninit-own.

$\{\llbracket\mathbf{L}\rrbracket(q) * [\mathsf{NaInv} : t.\top] * \mathsf{Dealloc}(\ell, \llbracket\tau\rrbracket.\textsc{size}, n) * \ell \mapsto \overline{w} * \llbracket\lightning_n\rrbracket.\textsc{own}(t, \overline{w})\}_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}}$

Fold $\llbracket\mathbf{own}\ \_\rrbracket.\textsc{own}$.

Goal: $\Rrightarrow_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}} \llbracket\mathbf{L}\rrbracket(q) * [\mathsf{NaInv} : t.\top] * \llbracket\mathbf{own}_m\ \lightning_n\rrbracket.\textsc{own}(t, v)$

Figure 12.10: Proof outline for Tread-own-move-sem.

Semantic correctness of Tread-own-move. Next, we consider the rule that lets us move the τ out of an **own** τ, leaving behind an **own** $\lightning_n$:

Tread-own-move-sem

$$\frac{n = \mathsf{size}(\tau)}{\Gamma \mid \mathbf{E}; \mathbf{L} \models \mathbf{own}_m\ \tau \multimapinv^{\tau} \mathbf{own}_m\ \lightning_n}$$

The proof outline for this rule is shown in Figure 12.10.[30] After unfolding our assumptions, we basically already have what we need to complete the first part of the accessor. We just have to use ty-size to remember for later that $\overline{w}$, the list of values the owned pointer points to, has length $\mathsf{size}(\tau)$. Then we go on with the accessor, and in the end we need to show that the owned pointer now points to $\lightning_n$. Here we use uninit-own which says that we only need to prove that $\overline{w}$ has the right length.[31]

Semantic correctness of S-deref-bor-mut *for shared references.* To demonstrate working with the "delayed sharing" of mutable references, we show how to verify the following rule:

S-deref-shr-mut-sem

$$\frac{\mathbf{E}; \mathbf{L} \models \kappa\ \mathsf{alive} \qquad \mathbf{E}; \mathbf{L} \models \kappa \sqsubseteq \kappa'}{\mathbf{E}; \mathbf{L} \mid p \lhd \&^{\kappa}_{\mathsf{shr}}\ \&^{\kappa'}_{\mathsf{mut}}\ \tau \models {}^{*}p \Dashv x.\ x \lhd \&^{\kappa}_{\mathsf{shr}}\ \tau}$$

The proof outline is given in Figure 12.11. Note that we are verifying a Hoare triple here, and the code we are verifying is ${}^{*}p$.[32]

After unfolding everything, we use the fact that κ is alive (the first premise) to obtain some fraction of the token for κ. We actually need two tokens for this lifetime, so we split the token, halving the fraction (LftL-tok-fract). One token is used to get access to $\llbracket p\rrbracket \overset{q}{\mapsto} \_$ via LftL-fract-acc, of which we obtained a fractured borrow from unfolding the shared and mutable references.

The other token is converted into a token for $\kappa \sqcap \kappa'$, which (by LftL-incl-glb and with our second assumption $\kappa \sqsubseteq \kappa'$) is the same lifetime

[30] This time, we immediately introduce all the assumptions of the goal into the first statement of our available resources (so we basically skip the first step of the previous proof outline).

[31] As one would expect, for uninitialized memory the actual data in the list does not matter.

[32] We use the same colors as before for resources in general and persistent resources specifically, and also use blue to indicate the code that is being verified.

$$\left\{\begin{array}{l}\Big((\mathbf{1})\ \Box\,\forall q.\ \llbracket\mathbf{E}\rrbracket \mathrel{-\!\!*} \big(\llbracket\mathbf{L}\rrbracket(q)\ \ \propto_{\mathcal{N}_{\mathsf{lft}},\mathcal{N}_{\mathsf{rust}}}\ q'.\ [\llbracket\kappa\rrbracket]_{q'}\big)\Big) * \Big((\mathbf{2})\ \Box\,\forall q.\ \llbracket\mathbf{L}\rrbracket(q) \mathrel{-\!\!*} \Box(\llbracket\mathbf{E}\rrbracket \mathrel{-\!\!*} \llbracket\kappa\rrbracket \sqsubseteq \llbracket\kappa'\rrbracket)\Big) * \\ \llbracket\mathbf{E}\rrbracket * \llbracket\mathbf{L}\rrbracket(1) * \llbracket p \lhd \&^{\kappa}_{\mathsf{shr}}\,\&^{\kappa'}_{\mathsf{mut}}\,\tau\rrbracket(t) * [\mathsf{NaInv} : t.\top]\end{array}\right\}_{\top}$$

Instantiate (2) with $q := 1$ and $\llbracket\mathbf{L}\rrbracket(1)$ (which we keep by $\Box$-wand-keep), and then with $\llbracket\mathbf{E}\rrbracket$.
Unfold $\llbracket\&^{\kappa}_{\mathsf{shr}}\,\_\rrbracket$.own (shr-ref-own).

$$\Big\{\big((\mathbf{3})\ \llbracket\kappa\rrbracket \sqsubseteq \llbracket\kappa'\rrbracket\big) * \llbracket\mathbf{L}\rrbracket(1) * \llbracket\&^{\kappa'}_{\mathsf{mut}}\,\tau\rrbracket.\mathrm{SHR}(\llbracket\kappa\rrbracket, t, \llbracket p\rrbracket) * [\mathsf{NaInv} : t.\top]\Big\}_{\top}$$

Unfold $\llbracket\&^{\kappa}_{\mathsf{mut}}\,\_\rrbracket$.shr (Figure 12.5).

$$\left\{\begin{array}{l}\llbracket\mathbf{L}\rrbracket(1) * \&^{\llbracket\kappa\rrbracket}_{\mathsf{frac}}(\lambda q.\ \llbracket p\rrbracket \overset{q}{\mapsto} \ell') * [\mathsf{NaInv} : t.\top] * \\ (\mathbf{4})\ \Box\Big(\forall q.\ [\llbracket\kappa'\rrbracket \sqcap \llbracket\kappa\rrbracket]_q \mathrel{-\!\!*} {}^{\mathcal{N}_{\mathsf{shr}}\cup\mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{lft}}} \triangleright\, {}^{\mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{shr}}\cup\mathcal{N}_{\mathsf{lft}}} \big(\llbracket\tau\rrbracket.\mathrm{SHR}(\llbracket\kappa'\rrbracket \sqcap \llbracket\kappa\rrbracket, t, \ell') * [\llbracket\kappa'\rrbracket \sqcap \llbracket\kappa\rrbracket]_q\big)\Big)\end{array}\right\}_{\top}$$

Use (1) with $q := 1$, split resulting token $[\llbracket\kappa\rrbracket]_{q_\kappa}$ into two (LftL-tok-fract on page 142),
get closing view shift: $(\mathbf{5})\ [\llbracket\kappa\rrbracket]_{q_\kappa} \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}},\mathcal{N}_{\mathsf{rust}}} \llbracket\mathbf{L}\rrbracket(1)$.

$$\Big\{\&^{\llbracket\kappa\rrbracket}_{\mathsf{frac}}(\lambda q.\ \llbracket p\rrbracket \overset{q}{\mapsto} \ell') * [\llbracket\kappa\rrbracket]_{q_\kappa/2} * [\llbracket\kappa\rrbracket]_{q_\kappa/2} * [\mathsf{NaInv} : t.\top] * (5)\Big\}_{\top}$$

Open fractured borrow (LftL-fract-acc, page 146), get closing view shift: $(\mathbf{6})\ \llbracket p\rrbracket \overset{q_p}{\longmapsto} \ell' \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}}} [\llbracket\kappa\rrbracket]_{q_\kappa/2}$.

$$\Big\{\llbracket p\rrbracket \overset{q_p}{\longmapsto} \ell' * [\llbracket\kappa\rrbracket]_{q_\kappa/2} * [\mathsf{NaInv} : t.\top] * (5) * (6)\Big\}_{\top}$$

By LftL-incl-glb and (3), we have $(\mathbf{7})\ \llbracket\kappa\rrbracket \sqsubseteq \llbracket\kappa'\rrbracket \sqcap \llbracket\kappa\rrbracket$. Thus by LftL-incl we can exchange $[\llbracket\kappa\rrbracket]_{q_\kappa/2}$ for $[\llbracket\kappa'\rrbracket \sqcap \llbracket\kappa\rrbracket]_{q_{\kappa'}}$, getting closing view shift: $(\mathbf{8})\ [\llbracket\kappa'\rrbracket \sqcap \llbracket\kappa\rrbracket]_{q_{\kappa'}} \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}}} [\llbracket\kappa\rrbracket]_{q_\kappa/2}$.

$$\Big\{\llbracket p\rrbracket \overset{q_p}{\longmapsto} \ell' * [\llbracket\kappa'\rrbracket \sqcap \llbracket\kappa\rrbracket]_{q_{\kappa'}} * [\mathsf{NaInv} : t.\top] * (5) * (6) * (8)\Big\}_{\top}$$

Instantiate (4) with $q := q_{\kappa'}$ and the token we just got.

$$\left\{\begin{array}{l}\llbracket p\rrbracket \overset{q_p}{\longmapsto} \ell' * [\mathsf{NaInv} : t.\top] * (5) * (6) * (8) * \\ {}^{\mathcal{N}_{\mathsf{shr}}\cup\mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{lft}}} \triangleright\, {}^{\mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{shr}}\cup\mathcal{N}_{\mathsf{lft}}} \big(\llbracket\tau\rrbracket.\mathrm{SHR}(\llbracket\kappa'\rrbracket \sqcap \llbracket\kappa\rrbracket, t, \ell') * [\llbracket\kappa'\rrbracket \sqcap \llbracket\kappa\rrbracket]_{q_{\kappa'}}\big)\end{array}\right\}_{\top}$$

$^{*}p$ (wp-sem-path and then lrust-deref-na, plus wp-step to strip away the $\Rrightarrow \triangleright \Rrightarrow$)

$$\Big\{\llbracket p\rrbracket \overset{q_p}{\longmapsto} \ell' * [\mathsf{NaInv} : t.\top] * (5) * (6) * (8) * \llbracket\tau\rrbracket.\mathrm{SHR}(\llbracket\kappa'\rrbracket \sqcap \llbracket\kappa\rrbracket, t, \ell') * [\llbracket\kappa'\rrbracket \sqcap \llbracket\kappa\rrbracket]_{q_{\kappa'}}\Big\}_{\top}$$

Use all the closing view shifts: (8), (6), (5).

$$\{\llbracket\mathbf{L}\rrbracket(1) * [\mathsf{NaInv} : t.\top] * \llbracket\tau\rrbracket.\mathrm{SHR}(\llbracket\kappa'\rrbracket \sqcap \llbracket\kappa\rrbracket, t, \ell')\}_{\top}$$

Apply ty-shr-mono with (7). Fold $\llbracket\&^{\kappa}_{\mathsf{shr}}\,\_\rrbracket$.own (shr-ref-own).

$$\{\llbracket\mathbf{L}\rrbracket(1) * [\mathsf{NaInv} : t.\top] * \llbracket\&^{\kappa}_{\mathsf{shr}}\,\tau\rrbracket.\mathrm{OWN}(t, \ell')\}_{\top}$$

Goal: $\Rrightarrow_{\top} (\llbracket\mathbf{L}\rrbracket(1) * \llbracket\ell' \lhd \&^{\kappa}_{\mathsf{shr}}\,\tau\rrbracket(t) * [\mathsf{NaInv} : t.\top])$

Figure 12.11: Proof outline for S-deref-shr-mut-sem.

as κ. This token for $\kappa \sqcap \kappa'$ is what we need to make use of the "delayed sharing" assertion that we obtained when unfolding the mutable reference:

$$\Box\Big(\forall q.\ [\llbracket\kappa'\rrbracket \sqcap \llbracket\kappa\rrbracket]_q \mathrel{-\!\!*} {}^{\mathcal{N}_{\mathsf{shr}}\cup\mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{lft}}} \triangleright\, {}^{\mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{shr}}\cup\mathcal{N}_{\mathsf{lft}}} \\ \big(\llbracket\tau\rrbracket.\mathrm{SHR}(\llbracket\kappa'\rrbracket \sqcap \llbracket\kappa\rrbracket, t, \ell') * [\llbracket\kappa'\rrbracket \sqcap \llbracket\kappa\rrbracket]_q\big)\Big)$$

To eliminate the three modalities that make up this "view shift that takes a step" (${}^{\mathcal{N}_{\mathsf{shr}}\cup\mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{lft}}} \triangleright\, {}^{\mathcal{N}_{\mathsf{lft}}}\!\Rrightarrow^{\mathcal{N}_{\mathsf{shr}}\cup\mathcal{N}_{\mathsf{lft}}}$), we use wp-step (page 75). This means that after executing $^{*}p$, we obtain the sharing predicate for τ, and we get back the lifetime token that we invested earlier. We can now close everything up again, and the postcondition is just one application of ty-shr-mono away.

Semantic correctness of F-newlft. The final proof rule we will look at is the one to start a new lifetime:

$$\textsc{F-newlft-sem}\qquad \frac{\forall \alpha.\ \mathbf{E}; \mathbf{L}, \alpha \sqsubseteq_{\mathsf{l}} \bar{\kappa} \mid \mathbf{K}; \mathbf{T} \models F}{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \models \mathtt{newlft}; F}$$

The proof outline is shown in Figure 12.12.

$\Big\{\Big((\mathbf{1})\ \forall \alpha.\ \Box\, \forall t.\ [\![\mathbf{E}]\!] * [\![\mathbf{L}, \alpha \sqsubseteq_{\mathsf{l}} \bar{\kappa}]\!](1) * [\![\mathbf{K}]\!](t) * [\![\mathbf{T}]\!](t) * [\mathsf{NaInv} : t.\top] \mathrel{-\!\!*} \mathsf{wp}\ F\ \{\mathsf{True}\}\Big)\Big\}$

Goal: $\Box\, \forall t.\ [\![\mathbf{E}]\!] * [\![\mathbf{L}]\!](1) * [\![\mathbf{K}]\!](t) * [\![\mathbf{T}]\!](t) * [\mathsf{NaInv} : t.\top] \mathrel{-\!\!*} \mathsf{wp}\ \mathtt{newlft}; F\ \{\mathsf{True}\}$

Introduce persistence modality (□-intro) and assumptions of magic wand.

$\{[\![\mathbf{E}]\!] * [\![\mathbf{L}]\!](1) * [\![\mathbf{K}]\!](t) * [\![\mathbf{T}]\!](t) * [\mathsf{NaInv} : t.\top]\}_{\top}$

$\mathtt{newlft};$

Make use of LftL-begin.

$\big\{[\kappa]_1 * \Box\big([\kappa]_1 \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}}} [\dagger\kappa]\big) * [\![\mathbf{E}]\!] * [\![\mathbf{L}]\!](1) * [\![\mathbf{K}]\!](t) * [\![\mathbf{T}]\!](t) * [\mathsf{NaInv} : t.\top]\big\}_{\top}$

Define $\alpha := \kappa \sqcap (\bigsqcap_{\kappa'' \in [\![\bar{\kappa}]\!]} \kappa'')$. Fold $[\![\alpha \sqsubseteq_{\mathsf{l}} \bar{\kappa}]\!]$ (Figure 12.1).

$\{[\![\alpha \sqsubseteq_{\mathsf{l}} \bar{\kappa}]\!](1) * [\![\mathbf{E}]\!] * [\![\mathbf{L}]\!](1) * [\![\mathbf{K}]\!](t) * [\![\mathbf{T}]\!](t) * [\mathsf{NaInv} : t.\top]\}_{\top}$

F (by (1), using all our resources)

Figure 12.12: Proof outline for F-newlft-sem.

There is actually not much happening here, we apply LftL-begin and let the lifetime logic do the heavy lifting. It is worth pointing out that the new lifetime α is *not* equal to the lifetime κ that is generated by the lifetime logic; instead, we pick $\alpha := \kappa \sqcap \bar{\kappa}$, matching how we set up the interpretation of external lifetime contexts in §12.1. This corresponds to the type system rule Lalive-local (page 115), which says when a local lifetime is alive: one might expect that local lifetimes are always alive, but they can be dead when one of their superlifetimes in $\bar{\kappa}$ has been ended. This can happen because the local lifetime context only owns the token for the extra lifetime κ that is intersected with all the others in $\bar{\kappa}$ to obtain α. Consequently, Lalive-local says that we need to prove that all the lifetimes in $\bar{\kappa}$ are still alive.

This concludes our tour through the semantic model of λ_{Rust}. To obtain Theorem 4, the fundamental theorem of logical relations, we have to complete proofs like the above for every single typing rule. Verifying *all* the typing rules by hand would be rather tedious and very error-prone, which is why we carried out this verification in Coq. Most of these proofs are easy; S-deref-shr-mut-sem is one of the few examples of typing rules where something interesting is happening. The key challenge in building a semantic model of λ_{Rust} was to come up with the right definitions so that everything actually fits together—almost everything complicated is already handled by the lifetime logic.

Chapter 13

Modeling types with interior mutability

As we have discussed in §8.6, the standard library of Rust provides types with *interior mutability*. These types, written in Rust using `unsafe` features, can nonetheless be used safely because the interface they provide to client code encapsulates these unsafeties behind well-typed abstractions. We have proven the safety of several such libraries, namely: `Cell`, `RefCell`, `Mutex`, `RwLock`, `Rc`, and `Arc`.[1] To fulfill this goal, we first had to pick semantic interpretations (in the sense of §10) for the abstract types exported by these libraries (*e.g.*, `Cell<T>`). We then proved that each publicly exported function from these libraries satisfies the semantic interpretation of its type.

When modeling types with interior mutability, the most difficult definition is usually that of the sharing predicate $[\![\tau]\!].\textsc{shr}$. The sharing predicate used by these types is different from the default read-only predicate for "simple types" that we described in §10.4, and it varies greatly depending on which operations are allowed by the type in question. This variety reflects the range of mechanisms that can be used to ensure safe encapsulation of shared mutable state.

We focus our explanations on two representative forms of interior mutability that we have already presented in §8.6: `Cell` (§13.1) and `Mutex` (§13.2). `Cell` is a zero-overhead variant of interior mutability that is safe because it confines sharing to a single thread, and because it does not permit interior pointers that could be invalidated by mutation. `Mutex` permits safe mutable sharing across thread boundaries via mutual exclusion.

13.1 Cell

In §8.6, we have seen that `Cell<T>` stores values of type `T` and provides two core functions: `Cell::get` and `Cell::replace`, which can be used for reading from and writing to the cell (`replace` returns the old, previously stored value).[2] These are their types:

```
    Cell::get :  fn(&Cell<T>) -> T where T: Copy
Cell::replace :  fn(&Cell<T>, T) -> T
```

The goal in this section is to define the semantic interpretation $[\![\textsf{cell}(\tau)]\!]$ as the equivalent of `Cell<T>` in RustBelt such that these operations are semantically well-typed.[3]

[1] Note that some simplifications of our setup make the proof of a few of these libraries simpler: we are not handling unwinding after panics, and we assume all atomic memory operations are sequentially consistent, while Rust's standard library uses weaker atomic accesses.
Beyond the libraries listed here, we have also verified some `unsafe` code that does not involve any interior mutability: `take_mut`, `mem::swap`, `thread::spawn` and `rayon::join`. The last two are particularly important as they give safe code access to concurrency.

[2] `Cell::set` is safely implemented in terms of `Cell::replace`, so it does not have to be verified.

[3] Here we pretend that $\textsf{cell}$ is a syntactic type. In our actual proof in Coq, we have defined a semantic type constructor $\textsf{cell}(\_)$ that can be applied to any semantic type in *SemType* (as mentioned before, all semantic interpretations also work for any semantic type, not just syntactic types). Thus, we do not actually have to extend the grammar of types to account for `Cell`.

13.1.1 Semantic interpretation of `Cell`

It turns out that *ownership* of $\mathsf{cell}(\tau)$ is the same as τ:

$$[\![\mathsf{cell}(\tau)]\!].\textsc{size} := [\![\tau]\!].\textsc{size}$$

$$[\![\mathsf{cell}(\tau)]\!].\textsc{own}(t, \bar{v}) := [\![\tau]\!].\textsc{own}(t, \bar{v}) \qquad (\textsc{cell-own})$$

This is even observable in the Rust standard library, which provides two functions for converting between `T` and `Cell<T>`: `Cell::new` and `Cell::into_inner`. Both of these are effectively the identity function. In other words, every owned τ is also an owned $\mathsf{cell}(\tau)$, and vice versa.[4]

The sharing predicate is where things become interesting. Remember that `replace` can be called even if you only have a *shared* reference to a `Cell<T>`. Translated to λ_{Rust}, their types are as follows:

$$\mathtt{get} : \tau \ \mathsf{copy} \Rightarrow \forall \alpha.\, \mathbf{fn}(\digamma : \digamma \sqsubseteq_{\mathsf{e}} \alpha; \&^{\alpha}_{\mathsf{shr}}\, \mathsf{cell}(\tau)) \to \tau$$

$$\mathtt{replace} : \forall \alpha.\, \mathbf{fn}(\digamma : \digamma \sqsubseteq_{\mathsf{e}} \alpha; \&^{\alpha}_{\mathsf{shr}}\, \mathsf{cell}(\tau), \tau) \to \tau$$

This means that `Cell<i32>` must use a very different sharing predicate than `i32`, which merely provides read-only access. In contrast, to verify `replace`, we need *full* (read/write) access for the duration of the function call. However, it is also important that all shared references to a `Cell` are confined to a single thread, since the `get` and `replace` operations are not thread safe. Recall that Rust enforces this by declaring that `Cell` is not `Sync`, which is equivalent to saying that `&Cell` is not `Send`, so that shared references to it cannot be sent to another thread—they must stay in the thread they have initially been created in.

In order to encode this idea, we use non-atomic borrows as discussed in §12.3:[5]

$$[\![\mathsf{cell}(\tau)]\!].\textsc{shr}(\kappa, t, \ell) := \&^{\kappa/t.\mathcal{N}_{\mathsf{shr}}.\ell}_{\mathsf{na}} \left(\exists \bar{v}.\, \ell \mapsto \bar{v} * [\![\tau]\!].\textsc{own}(t, \bar{v})\right) \qquad (\textsc{cell-shr})$$

The namespace for the non-atomic invariant is $\mathcal{N}_{\mathsf{shr}}.\ell$ in pool t; we have already seen this use of locations in namespaces when defining $\models \tau\ \mathsf{copy}$, and indeed that definition was carefully chosen to permit $\mathsf{cell}(\tau)$ to be `Copy`.

Well-formedness. To make sure $[\![\mathsf{cell}(\tau)]\!]$ is a well-formed semantic type, we have to show the conditions laid down in Figure 12.4. ty-size is easy (it follows from $[\![\tau]\!]$ satisfying it). ty-shr-persist and ty-shr-mono follow from LftL-na-persist and LftL-na-shorten, respectively. The most interesting property is usually ty-share, but in our case that just yields the following goal:[6]

$$\&^{\kappa}_{\mathsf{full}} \left(\exists \bar{w}.\, \ell \mapsto \bar{w} * [\![\tau]\!].\textsc{own}(t, \bar{w})\right) * [\kappa]_q \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}}} \&^{\kappa/t.\mathcal{N}_{\mathsf{shr}}.\ell}_{\mathsf{na}} \left(\exists \bar{v}.\, \ell \mapsto \bar{v} * [\![\tau]\!].\textsc{own}(t, \bar{v})\right) * [\kappa]_q$$

This follows directly via LftL-bor-na; we do not even need the lifetime token.

Marker traits. We also have to show that whenever the Rust type `Cell<T>` is `Copy`/`Send`/`Sync`, then we can prove the corresponding semantic properties for our $\mathsf{cell}(\tau)$.

[4] This also explains `Cell::get_mut`, which can turn a `&mut Cell<T>` into `&mut T`: mutable references are defined in terms of full ownership, so if an owned τ_1 and τ_2 are equivalent, then so are owned $\&^{\kappa}_{\mathsf{mut}}\, \tau_1$ and $\&^{\kappa}_{\mathsf{mut}}\, \tau_2$.

[5] With respect to the handling of recursive types that we mentioned at the end of §10.4, it is important to notice that the sharing predicate of $[\![\mathsf{cell}(\tau)]\!]$ only makes *guarded* use of the ownership predicate of τ (*i.e.*, the use is inside a borrow, which is almost the same as it being below a $\triangleright$). This is crucial for the soundness of recursive types that have their recursive occurrence inside a cell. It makes sure that even if the pointer type that guards the recursive type occurrence is a shared reference (where the sharing predicate of $\&^{\kappa}_{\mathsf{shr}}\, \tau$ makes unguarded use of the ownership predicate of τ), the underlying semantic type still appropriately guards all recursive occurrences on the Iris level.

[6] A note on parsing: $\&^{\kappa}_{\mathsf{full}}\, P * Q$ parses as $(\&^{\kappa}_{\mathsf{full}}\, P) * Q$, *i.e.*, borrows bind weaker than separating conjunctions. This means they bind like modalities, *e.g.*, $\Box$ and $\triangleright$.

For Copy, we prove that $\models \tau\ \mathsf{copy}$ implies $\models \mathbf{cell}(\tau)\ \mathsf{copy}$. The proof outline is shown in Figure 13.1. As already mentioned, Cell<T> is never actually Copy in Rust, but that is to avoid potential pitfalls, not due to soundness concerns.

First we treat $[\![\tau]\!].\textsc{size} = 0$ as a special case. Intuitively, zero-sized Copy types are special because everything about them is persistent: there is not even some memory that needs to be owned to access them. Hence we can prove:

$$[\![\mathbf{cell}(\tau)]\!].\textsc{shr}(\kappa, t, \ell) \mathrel{-\!\!*} \\ \Big([\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.\ell] * [\kappa]_q \ \propto_{\mathcal{N}_{\mathsf{shr}}, \mathcal{N}_{\mathsf{lft}}} \ [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.\ell] * \triangleright [\![\tau]\!].\textsc{own}(t, \epsilon)\Big)$$

This holds because we can use LftL-na-acc to open the non-atomic borrow, then use $\models \tau\ \mathsf{copy}$ to deduce that $[\![\tau]\!].\textsc{own}(t, \bar{v})$ is persistent (so we can duplicate it) and close the non-atomic borrow again. By ty-size, $\bar{v} = \epsilon$, which completes this accessor. Overall, this accessor is *non-mask-changing* for the non-atomic "mask". From this, $\models \mathbf{cell}(\tau)\ \mathsf{copy}$ easily follows because $\ell \mapsto \epsilon$ is trivial (lrust-heap-nil, page 133).

In the general case of the size not being zero, we prove:

$$[\![\mathbf{cell}(\tau)]\!].\textsc{shr}(\kappa, t, \ell) \mathrel{-\!\!*} \\ \Big([\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.\ell] * [\kappa]_q \ \propto_{\mathcal{N}_{\mathsf{shr}}, \mathcal{N}_{\mathsf{lft}}} \ q'.\, \exists \bar{v}.\, \ell \overset{q'}{\longmapsto} \bar{v} * \triangleright [\![\tau]\!].\textsc{own}(t, \epsilon)\Big)$$

To show this, we use LftL-na-acc to open the non-atomic borrow. Our accessor *does* change the non-atomic "mask" ($\mathcal{N}_{\mathsf{shr}}.\ell$ is opened and thus removed from the mask). But that is permitted by $\models \mathbf{cell}(\tau)\ \mathsf{copy}$ because the only part of the mask we have to keep is $\mathcal{N}_{\mathsf{shr}}.(\ell + [\![\tau]\!].\textsc{size})$, which is disjoint from the part we removed since $[\![\tau]\!].\textsc{size} \neq 0$.

As for Send, since we defined $[\![\mathbf{cell}(\tau)]\!].\textsc{own}$ via $[\![\tau]\!].\textsc{own}$, we can easily show that $\models \tau\ \mathsf{send}$ implies $\models \mathbf{cell}(\tau)\ \mathsf{send}$.

Finally, $\models \mathbf{cell}(\tau)\ \mathsf{sync}$ does not hold: the thread identifier t is crucial in the definition of our sharing predicate. This reflects the fact that Cell<T> is never Sync.

13.1.2 Soundness of the public operations

With the type interpretation in place and its basic properties verified,[7] the next step is to go over each operation that is defined on that type, and use sem-fn-body to prove that running this operation respects semantic typing. We will focus on *get* and *replace*.

Verifying get. The λ_{Rust} source code of *get* looks as follows:

```
funrec get_n(c) ret ret :=
  let c' = *c in
  let r = new(n) in
  r :=_n *c';
  delete(1, c);
  jump ret(r)
```

And this is its type:

$$get_{\mathsf{size}(\tau)} : \tau\ \mathsf{copy} \Rightarrow \forall \alpha.\, \mathbf{fn}(\digamma : \digamma \sqsubseteq_{\mathsf{e}} \alpha; \&^{\alpha}_{\mathbf{shr}}\, \mathbf{cell}(\tau)) \to \tau$$

[7] We are skipping variance in this discussion. Cell<T> is invariant, but we still have to prove that the type can be substituted by an *equivalent* type. We will not go into details of variance in this dissertation as there is nothing interesting happening there; suffice to say that the key proof rule used in proving variance lemmas of borrows is LftL-idx-iff (page 150).

Case $[\![\tau]\!].\textit{SIZE} = 0$.

$\left\{\left((\mathbf{1})\ \&_{\mathsf{na}}^{\kappa/t.\mathcal{N}_{\mathsf{shr}}.\ell}\left(\exists\bar{v}.\ \ell \mapsto \bar{v} * [\![\tau]\!].\textsc{own}(t,\bar{v})\right)\right) * [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.[\geq\ell, <\ell+0+1]] * [\kappa]_q\right\}_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}}$

Observe that $[\geq\ell, <\ell+0+1] = \{\ell\}$, so the token we own is $[\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.\ell]$.

Open non-atomic borrow (1) with LftL-na-acc,

get closing view shift:$(\mathbf{2})$ $\triangleright\left(\exists\bar{v}.\ \ell \mapsto \bar{v} * [\![\tau]\!].\textsc{own}(t,\bar{v})\right) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}} [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.\ell] * [\kappa]_q$.

$\{\ell \mapsto \bar{v} * \triangleright[\![\tau]\!].\textsc{own}(t,\bar{v}) * (2)\}_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}}$

($\triangleright[\![\tau]\!].\textsc{own}(t,\bar{v})$ is persistent by $\models \tau$ copy.)

By ty-size, we have $|\bar{v}| = 0$ and thus $\bar{v} = \epsilon$.

Use lrust-heap-nil to get a second $\ell \mapsto \epsilon$. Use (2) with $\bar{v} := \epsilon$.

$\left\{\ell \mapsto \epsilon * [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.\ell] * [\kappa]_q\right\}_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}}$

Goal: $\mathrel{|\!\!\Rrightarrow}_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}} (\exists q'.\ (\exists\bar{v}'.\ [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.(\ell+0)] * \ell \overset{q'}{\mapsto} \bar{v}' * \triangleright[\![\tau]\!].\textsc{own}(t,\bar{v}')) * \ldots)$

Introduce update modality, pick $q' := 1$ and $\bar{v}' := \epsilon$, and discharge first separating conjunct.

$\left\{[\kappa]_q\right\}$

Goal: $(\exists\bar{v}'.\ [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.\ell] * \ell \mapsto \bar{v}' * \triangleright[\![\tau]\!].\textsc{own}(t,\bar{v}')) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}} [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.\ell] * [\kappa]_q$

Introduce assumption of magic wand.

$\left\{[\kappa]_q * [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.\ell] * \ell \mapsto \bar{v}' * \triangleright[\![\tau]\!].\textsc{own}(t,\bar{v}')\right\}_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}}$

Goal: $\mathrel{|\!\!\Rrightarrow}_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}} \left([\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.\ell] * [\kappa]_q\right)$

Case $[\![\tau]\!].\textit{SIZE} \neq 0$.

$\left\{\left((\mathbf{1})\ \&_{\mathsf{na}}^{\kappa/t.\mathcal{N}_{\mathsf{shr}}.\ell}\left(\exists\bar{v}.\ \ell \mapsto \bar{v} * [\![\tau]\!].\textsc{own}(t,\bar{v})\right)\right) * [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.[\geq\ell, <\ell+1+[\![\tau]\!].\textsc{size}]] * [\kappa]_q\right\}_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}}$

Split off non-atomic token for $\mathcal{N}_{\mathsf{shr}}.\ell$ (NAInv-tok-split), needs $\ell \in [\geq\ell, <\ell+[\![\tau]\!].\textsc{size}+1]$.

Then open borrow (1) with LftL-na-acc,

get closing view shift: $(\mathbf{2})$ $\triangleright\left(\exists\bar{v}.\ \ell \mapsto \bar{v} * [\![\tau]\!].\textsc{own}(t,\bar{v})\right) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}} [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.\ell] * [\kappa]_q$.

$\{[\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.[\geq\ell+1, <\ell+[\![\tau]\!].\textsc{size}+1]] * \ell \mapsto \bar{v} * \triangleright[\![\tau]\!].\textsc{own}(t,\bar{v}) * (2)\}_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}}$

Split off non-atomic token for $\mathcal{N}_{\mathsf{shr}}.(\ell+[\![\tau]\!].\textsc{size})$, needs $(\ell+[\![\tau]\!].\textsc{size}) \in [\geq\ell+1, <\ell+[\![\tau]\!].\textsc{size}+1]$.

Goal: $\mathrel{|\!\!\Rrightarrow}_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}} (\exists q'.\ (\exists\bar{v}'.\ [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.(\ell+[\![\tau]\!].\textsc{size})] * \ell \overset{q'}{\mapsto} \bar{v}' * \triangleright[\![\tau]\!].\textsc{own}(t,\bar{v}')) * \ldots)$

Introduce update modality, pick $q' := 1$ and $\bar{v}' := \bar{v}$, and discharge first separating conjunct.

$\{[\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.[\geq\ell+1, <\ell+[\![\tau]\!].\textsc{size}]] * (2)\}$

Goal: $(\exists\bar{v}''.\ [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.\ell] * \ell \mapsto \bar{v}'' * \triangleright[\![\tau]\!].\textsc{own}(t,\bar{v}'')) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}}$
$[\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.[\geq\ell, <\ell+1+[\![\tau]\!].\textsc{size}]] * [\kappa]_q$

Introduce assumption of magic wand.

Merge the non-atomic tokens again (NAInv-tok-split).

$\{[\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.[\geq\ell+1, <\ell+[\![\tau]\!].\textsc{size}+1]] * (2) * \ell \mapsto \bar{v}'' * \triangleright[\![\tau]\!].\textsc{own}(t,\bar{v}'')\}_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}}$

Use (2), with $\bar{v} := \bar{v}''$.

$\left\{[\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.[\geq\ell+1, <\ell+[\![\tau]\!].\textsc{size}+1]] * [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.\ell] * [\kappa]_q\right\}_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}}$

Goal: $\mathrel{|\!\!\Rrightarrow}_{\mathcal{N}_{\mathsf{shr}},\mathcal{N}_{\mathsf{lft}}} \left([\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.[\geq\ell, <\ell+1+[\![\tau]\!].\textsc{size}]] * [\kappa]_q\right)$

Figure 13.1: Outline of `Copy` proof for `Cell<T>`.

Note that the code is parameterized by n, the size of the type τ. This is relevant to be able to copy the right amount of data.

This corresponds to the Rust source code of `Cell::get` after inlining a few extra layers of abstraction. We also account for the extra indirection of arguments and return values in λ_{Rust}: c is a *pointer to* a shared reference to a `Cell`, and we return a copy of the data contained in that `Cell`.

Our goal is to prove that get is semantically well-typed. Following S-fn, a function is well-typed if its body is well-typed. According to sem-fn-body, we can show that the body is well-typed by verifying the following Iris Hoare triple:[8]

$$\left\{\begin{array}{l} [\![\digamma \sqsubseteq_{\mathsf{e}} \alpha]\!] * [\![\digamma \sqsubseteq_{\mathsf{l}} [\,]]\!](q) * [\![\mathit{ret} \lhd \mathbf{cont}(\digamma \sqsubseteq_{\mathsf{l}} [\,];\, \mathit{r}.\, \mathit{r} \lhd \mathbf{own}\ \tau)]\!](t) * \\ [\![\mathit{c} \lhd \mathbf{own}\ \&^{\alpha}_{\mathbf{shr}}\, \mathbf{cell}(\tau)]\!](t) * [\mathsf{NaInv} : t.\top] \end{array}\right\}$$

$$\quad \mathtt{call}\ \mathit{get}_{[\![\tau]\!].\textsc{size}}(\mathit{c})\ \mathtt{ret}\ \mathit{ret}$$

$$\{\mathsf{True}\} \tag{cell-get}$$

The proof outline for this is shown in Figure 13.2.

As we can see, we end up having to prove a Hoare triple, so we are performing essentially standard Iris program verification. The first operation we have to justify is $\mathtt{let}\ \mathit{c'} = {}^{*}\mathit{c}$. Justifying such a load requires ownership of (a fraction of) $\mathit{c} \mapsto v$,[9] which we can obtain by unfolding the interpretation of our typing context: as defined in own-ptr-own (page 135), $\mathit{c} \lhd \mathbf{own}\ \&^{\alpha}_{\mathbf{shr}}\, \mathbf{cell}(\tau)$ means that c is a pointer that we can load from, and the value $\mathit{c'}$ it points to satisfies $[\![\&^{\alpha}_{\mathbf{shr}}\, \mathbf{cell}(\tau)]\!].\textsc{own}(t, [\mathit{c'}])$. By further unfolding ownership of a shared reference (shr-ref-own on page 137) we obtain $[\![\mathbf{cell}(\tau)]\!].\textsc{shr}(\alpha, t, \mathit{c'})$.

In the next step, the function allocates $[\![\tau]\!].\textsc{size}$ memory locations $(\mathtt{let}\ \mathit{r} = \mathtt{new}([\![\tau]\!].\textsc{size}))$. We make use of lrust-new (page 133) and acquire ownership of $\mathit{r} \mapsto \bar{v}_r$ for some list of values $\bar{v}_r$ of length $[\![\tau]\!].\textsc{size}$, and we also acquire $\mathsf{Dealloc}(\mathit{r}, [\![\tau]\!].\textsc{size}, [\![\tau]\!].\textsc{size})$, representing the permission to *deallocate* this block again.

Now we come to the key operation: $\mathit{r} :=_{[\![\tau]\!].\textsc{size}} {}^{*}\mathit{c'}$ creates a copy of the data $\mathit{c'}$ points to (*i.e.*, the data in the `Cell`) and puts that copy into r. The interesting part here is obtaining the permission to read from $\mathit{c'}$. To do this, we have to unfold our $[\![\mathbf{cell}(\tau)]\!].\textsc{shr}(\alpha, t, \mathit{c'})$; by our choice for the sharing predicate (cell-shr on page 206), we get a non-atomic borrow at lifetime α of $\exists \bar{v}.\ \mathit{c'} \mapsto \bar{v} * [\![\tau]\!].\textsc{own}(t, \bar{v})$. To access that borrow, we need the lifetime token for α. We use the following lemma to argue that since α is syntactically alive, it is also semantically alive:[10]

Lalive-sem

$$\frac{\mathbf{E}; \mathbf{L} \vdash \kappa\ \mathsf{alive}}{\forall q.\ [\![\mathbf{E}]\!] \mathrel{-\!\!*} \left([\![\mathbf{L}]\!](q) \ \propto_{\mathcal{N}_{\mathsf{lft}}, \mathcal{N}_{\mathsf{rust}}}\ q'.\, [[\![\kappa]\!]]_{q'}\right)}$$

The borrow also requires some non-atomic invariant tokens, which is where we make use of $[\mathsf{NaInv} : t.\top]$.[11] Thus we have both $\mathit{c'} \mapsto \bar{v}_{c'}$ and $\mathit{r} \mapsto \bar{v}_r$.[12] We can also show that both of these lists of values have length $[\![\tau]\!].\textsc{size}$ (by ty-size, $[\![\tau]\!].\textsc{own}(t, \bar{v}_{c'})$ implies that $\bar{v}_{c'}$ matches the size of τ).[13] Now we can apply lrust-memcpy (from page 133) to prove that

[8] We are slightly cheating here and abbreviate the body of get with a call—basically an η-expansion of the goal we would actually get.

[9] At this point we are conflating program-level variables c and logic-level variables such as ℓ' in the proof outline, as explained in §7.3.

[10] This lemma easily follows via the fundamental theorem (Theorem 4).

[11] The normal invariant mask remains unchanged, which means the non-atomic borrow can be kept open for the entire duration of the `memcpy`.

[12] We first obtain the borrowed proposition under a $\triangleright$ modality. However, we can commute that around the existential quantification over $\bar{v}$ as lists are inhabited (see Figure 5.1 on page 64), and then we can remove the $\triangleright$ from $\triangleright\, \mathit{c'} \mapsto \bar{v}_{c'}$ ($\mid\!\Rrightarrow$-timeless).

[13] Again ty-size first gives us $\triangleright(|\bar{v}_{c'}| = [\![\tau]\!].\textsc{size})$, but equality of integers is timeless.

Proof outline for CELL-GET.

$$\left\{\begin{array}{l}\llbracket \digamma \sqsubseteq_{\mathsf{e}} \alpha \rrbracket * \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * \Big((\mathbf{1})\ \llbracket \mathit{ret} \lhd \mathbf{cont}(\digamma \sqsubseteq_{\mathsf{l}} []; \mathit{r}.\ \mathit{r} \lhd \mathbf{own}\ \tau) \rrbracket(t) \Big) * \\ \llbracket c \lhd \mathbf{own}\ \&^{\alpha}_{\mathbf{shr}}\ \mathbf{cell}(\tau) \rrbracket(t) * [\mathsf{NaInv} : t.\top] \end{array}\right\}_{\top}$$

Unfold $\llbracket \mathbf{own}\ \_ \rrbracket$.OWN (OWN-PTR-OWN, page 135).

$$\{\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * (\exists \bar{v}_c.\ c \mapsto \bar{v}_c * \triangleright \llbracket \&^{\alpha}_{\mathbf{shr}}\ \mathbf{cell}(\tau) \rrbracket.\textsc{own}(t, \bar{v}_c) * \mathsf{Dealloc}(c, 1, 1)) * [\mathsf{NaInv} : t.\top]\}_{\top}$$

Unfold $\llbracket \&^{\alpha}_{\mathbf{shr}}\ \_ \rrbracket$.OWN (SHR-REF-OWN) and $\llbracket \mathbf{cell}(\tau) \rrbracket$.SHR (CELL-SHR).

$$\Big\{\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * \Big(\exists \ell'.\ c \mapsto \ell' * \triangleright \&^{\alpha/t.\mathcal{N}_{\mathsf{shr}}.\ell'}_{\mathsf{na}} \big(\exists \bar{v}.\ \ell' \mapsto \bar{v} * \llbracket \tau \rrbracket.\textsc{own}(t, \bar{v})\big)\Big) * \mathsf{Dealloc}(c, 1, 1) * [\mathsf{NaInv} : t.\top]\Big\}_{\top}$$

`let c' = *c in`   (LRUST-DEREF-NA, ℓ' becomes c')

$$\left\{\begin{array}{l}\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * \big((\mathbf{2})\ c \mapsto c' * \mathsf{Dealloc}(c, 1, 1)\big) * \\ \Big((\mathbf{3})\ \&^{\alpha/t.\mathcal{N}_{\mathsf{shr}}.c'}_{\mathsf{na}} \big(\exists \bar{v}.\ c' \mapsto \bar{v} * \llbracket \tau \rrbracket.\textsc{own}(t, \bar{v})\big)\Big) * [\mathsf{NaInv} : t.\top] \end{array}\right\}_{\top}$$

`let r = new(`$\llbracket \tau \rrbracket$`.SIZE) in`   (LRUST-NEW)

$$\{\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * (2) * r \mapsto \bar{v}_r * |\bar{v}_r| = \llbracket \tau \rrbracket.\textsc{size} * \big((\mathbf{4})\ \mathsf{Dealloc}(r, \llbracket \tau \rrbracket.\textsc{size}, \llbracket \tau \rrbracket.\textsc{size})\big) * [\mathsf{NaInv} : t.\top]\}_{\top}$$

Use $\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q)$ and $\digamma \sqsubseteq_{\mathsf{e}} \alpha; \digamma \sqsubseteq_{\mathsf{l}} [] \vdash \alpha\ \mathsf{alive}$ to get $[\alpha]_{q'}$ (LALIVE-SEM),
closing view shift: $(\mathbf{5})\ [\alpha]_{q'} \mathrel{\Rrightarrow\!\!\ast}_{\mathcal{N}_{\mathsf{lft}}} \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q)$.

$$\Big\{(1) * (2) * [\alpha]_{q'} * r \mapsto \bar{v}_r * (4) * (5) * [\mathsf{NaInv} : t.\top]\Big\}_{\top}$$

With that new token, open borrow (3) via LFTL-NA-ACC,
get closing view shift: $(\mathbf{6})\ \triangleright\big(\exists \bar{v}.\ c' \mapsto \bar{v} * \llbracket \tau \rrbracket.\textsc{own}(t, \bar{v})\big) \mathrel{\Rrightarrow\!\!\ast}_{\mathcal{N}_{\mathsf{shr}}, \mathcal{N}_{\mathsf{lft}}} [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.c'] * [\alpha]_{q'}$.

$$\{(1) * (2) * c' \mapsto \bar{v}_{c'} * \triangleright \llbracket \tau \rrbracket.\textsc{own}(t, \bar{v}_{c'}) * r \mapsto \bar{v}_r * (4) * (5) * (6) * [\mathsf{NaInv} : t.\top \setminus \mathcal{N}_{\mathsf{shr}}.c']\}_{\top}$$

`r :=`$_{\llbracket \tau \rrbracket.\textsc{size}}$` *c';`   (LRUST-MEMCPY with $|\bar{v}_r| = \llbracket \tau \rrbracket.\textsc{size}$, and $|\bar{v}_{c'}| = \llbracket \tau \rrbracket.\textsc{size}$ by TY-SIZE)

$$\{(1) * (2) * c' \mapsto \bar{v}_{c'} * \llbracket \tau \rrbracket.\textsc{own}(t, \bar{v}_{c'}) * r \mapsto \bar{v}_{c'} * (4) * (5) * (6) * [\mathsf{NaInv} : t.\top \setminus \mathcal{N}_{\mathsf{shr}}.c']\}_{\top}$$

$\llbracket \tau \rrbracket$.OWN is persistent by $\models \tau\ \mathsf{copy}$. Close borrow (6) with $\bar{v} := \bar{v}_{c'}$, and then lifetime context (5).

$$\{\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * (2) * r \mapsto \bar{v}_{c'} * (4) * [\mathsf{NaInv} : t.\top]\}_{\top}$$

`delete(1, c);`   (LRUST-DELETE, consumes (2))

$$\{\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * r \mapsto \bar{v}_{c'} * (4) * [\mathsf{NaInv} : t.\top]\}_{\top}$$

Fold $r \mapsto \bar{v}_{c'} * \llbracket \tau \rrbracket.\textsc{own}(t, \bar{v}_{c'}) * (4)$ into $\llbracket r \lhd \mathbf{own}\ \tau \rrbracket(t)$ (OWN-PTR-OWN).

$$\{\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * \llbracket r \lhd \mathbf{own}\ \tau \rrbracket(t) * [\mathsf{NaInv} : t.\top]\}_{\top}$$

`jump ret(r)`   (by (1), using up all the other resources)

Figure 13.2: Proof outline for `Cell::get`.

the **`memcpy`** is safe. Afterwards, we close the non-atomic borrow again, and we un-do the LALIVE-SEM accessor to get back our lifetime contexts.

Then the program deallocates c by running **delete**$(1, c)$.[14] By LRUST-DELETE and OWN-PTR-OWN, the deallocation permission $\mathsf{Dealloc}(c, 1, 1)$ we need for this is contained in the interpretation of the owned pointer c that we started with.

Finally, **jump** *ret*(r) calls the continuation *ret* with the pointer r, where we put the copy of the `Cell` content. This exactly uses up all the resources we have left: ownership of the local lifetime context, of r, and of the non-atomic invariant token.

Verifying `replace`. For *replace*, the λ_{Rust} source code looks like this:

```
funrec replace_n(c, x) ret ret :=
  let c' = *c in
  let r = new(n) in
  r :=_n *c';
  c' :=_n *x;
  delete(1, c); delete(n, x);
  jump ret(r)
```

[14] This corresponds to popping the stack frame and thus releasing the storage of local variables.

Proof outline for CELL-REPLACE.

$$\left\{\begin{array}{l}\llbracket \digamma \sqsubseteq_{\mathsf{e}} \alpha \rrbracket * \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * \Big(\mathbf{(1)}\ \llbracket \mathit{ret} \triangleleft \mathbf{cont}(\digamma \sqsubseteq_{\mathsf{l}} [];\, r.\, r \triangleleft \mathbf{own}\ \tau) \rrbracket(t)\Big) * \\ \Big(\mathbf{(2)}\ \llbracket x \triangleleft \mathbf{own}\ \tau \rrbracket(t)\Big) * \llbracket c \triangleleft \mathbf{own}\ \&^{\alpha}_{\mathbf{shr}}\ \mathbf{cell}(\tau) \rrbracket(t) * [\mathsf{NaInv} : t.\top] \end{array}\right\}_{\top}$$

Unfold $\llbracket \mathbf{own}\ \_ \rrbracket$.OWN (OWN-PTR-OWN, page 135).

Unfold $\llbracket \&^{\alpha}_{\mathbf{shr}}\ \_ \rrbracket$.OWN (SHR-REF-OWN) and $\llbracket \mathbf{cell}(\tau) \rrbracket$.SHR (CELL-SHR).

$$\left\{\begin{array}{l}\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * (2) * \Big(\exists \ell'.\ c \mapsto \ell' * \triangleright \&^{\alpha/t.\mathcal{N}_{\mathsf{shr}}.\ell'}_{\mathsf{na}} \big(\exists \bar{v}.\ \ell' \mapsto \bar{v} * \llbracket \tau \rrbracket.\mathrm{OWN}(t, \bar{v})\big)\Big) * \\ \mathsf{Dealloc}(c, 1, 1) * [\mathsf{NaInv} : t.\top] \end{array}\right\}_{\top}$$

`let c’ = *c in` (LRUST-DEREF-NA, ℓ' becomes c’)

$$\left\{\begin{array}{l}\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * (2) * \big(\mathbf{(3)}\ c \mapsto c' * \mathsf{Dealloc}(c, 1, 1)\big) * \\ \Big(\mathbf{(4)}\ \&^{\alpha/t.\mathcal{N}_{\mathsf{shr}}.c'}_{\mathsf{na}} \big(\exists \bar{v}.\ c' \mapsto \bar{v} * \llbracket \tau \rrbracket.\mathrm{OWN}(t, \bar{v})\big)\Big) * [\mathsf{NaInv} : t.\top] \end{array}\right\}_{\top}$$

`let r = new(⟦τ⟧.SIZE) in` (LRUST-NEW)

$$\{\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * (2) * (3) * r \mapsto \bar{v}_r * |\bar{v}_r| = \llbracket \tau \rrbracket.\mathrm{SIZE} * \big(\mathbf{(5)}\ \mathsf{Dealloc}(r, \llbracket \tau \rrbracket.\mathrm{SIZE}, \llbracket \tau \rrbracket.\mathrm{SIZE})\big) * [\mathsf{NaInv} : t.\top]\}_{\top}$$

Use $\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q)$ with LALIVE-SEM to get $[\alpha]_{q'}$, closing view shift: $\mathbf{(6)}\ [\alpha]_{q'} \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}}} \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q)$.

With that new token, open borrow (4) via LFTL-NA-ACC,

get closing view shift: $\mathbf{(7)}\ \triangleright\big(\exists \bar{v}.\ c' \mapsto \bar{v} * \llbracket \tau \rrbracket.\mathrm{OWN}(t, \bar{v})\big) \Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{shr}}, \mathcal{N}_{\mathsf{lft}}} [\mathsf{NaInv} : t.\mathcal{N}_{\mathsf{shr}}.c'] * [\alpha]_{q'}$.

$$\{(1) * (2) * (3) * c' \mapsto \bar{v}_{c'} * \triangleright \llbracket \tau \rrbracket.\mathrm{OWN}(t, \bar{v}_{c'}) * r \mapsto \bar{v}_r * (5) * (6) * (7) * [\mathsf{NaInv} : t.\top \setminus \mathcal{N}_{\mathsf{shr}}.c']\}_{\top}$$

`r :=⟦τ⟧.SIZE *c’;` (LRUST-MEMCPY with $|\bar{v}_r| = \llbracket \tau \rrbracket.\mathrm{SIZE}$, and $|\bar{v}_{c'}| = \llbracket \tau \rrbracket.\mathrm{SIZE}$ by TY-SIZE)

$$\{(1) * (2) * (3) * c' \mapsto \bar{v}_{c'} * \llbracket \tau \rrbracket.\mathrm{OWN}(t, \bar{v}_{c'}) * r \mapsto \bar{v}_{c'} * (5) * (6) * (7) * [\mathsf{NaInv} : t.\top \setminus \mathcal{N}_{\mathsf{shr}}.c']\}_{\top}$$

Now things start to diverge from CELL-GET.

Fold $r \mapsto \bar{v}_{c'} * \llbracket \tau \rrbracket.\mathrm{OWN}(t, \bar{v}_{c'}) * (5)$ into $\llbracket r \triangleleft \mathbf{own}\ \tau \rrbracket(t)$. Unfold (2).

$$\left\{\begin{array}{l}(1) * x \mapsto \bar{v}_x * \llbracket \tau \rrbracket.\mathrm{OWN}(t, \bar{v}_x) * \big(\mathbf{(8)}\ \mathsf{Dealloc}(x, \llbracket \tau \rrbracket.\mathrm{SIZE}, \llbracket \tau \rrbracket.\mathrm{SIZE})\big) * (3) * c' \mapsto \bar{v}_{c'} * \\ \llbracket r \triangleleft \mathbf{own}\ \tau \rrbracket(t) * (6) * (7) * [\mathsf{NaInv} : t.\top \setminus \mathcal{N}_{\mathsf{shr}}.c'] \end{array}\right\}_{\top}$$

`c’ :=⟦τ⟧.SIZE *x;` (LRUST-MEMCPY with $|\bar{v}_{c'}| = \llbracket \tau \rrbracket.\mathrm{SIZE}$, and $|\bar{v}_x| = \llbracket \tau \rrbracket.\mathrm{SIZE}$ by TY-SIZE)

$$\{(1) * x \mapsto \bar{v}_x * \llbracket \tau \rrbracket.\mathrm{OWN}(t, \bar{v}_x) * (8) * (3) * c' \mapsto \bar{v}_x * \llbracket r \triangleleft \mathbf{own}\ \tau \rrbracket(t) * (6) * (7) * [\mathsf{NaInv} : t.\top \setminus \mathcal{N}_{\mathsf{shr}}.c']\}_{\top}$$

Close borrow (7) with $\bar{v} := \bar{v}_x$, and then lifetime context (6).

$$\{\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * x \mapsto \bar{v}_x * (8) * (3) * \llbracket r \triangleleft \mathbf{own}\ \tau \rrbracket(t) * [\mathsf{NaInv} : t.\top \setminus \mathcal{N}_{\mathsf{shr}}]\}_{\top}$$

`delete(1, c);` (LRUST-DELETE, consumes (3))

`delete(⟦τ⟧.SIZE, x);` (LRUST-DELETE, consumes $x \mapsto \bar{v}_x * (8)$)

$$\{\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * \llbracket r \triangleleft \mathbf{own}\ \tau \rrbracket(t) * [\mathsf{NaInv} : t.\top \setminus \mathcal{N}_{\mathsf{shr}}]\}_{\top}$$

`jump ret(r)` (by (1), using up all the other resources)

Figure 13.3: Proof outline for `Cell::replace`.

And its type is:

$$\mathit{replace}_{\mathsf{size}(\tau)} : \forall \alpha.\ \mathbf{fn}(\digamma : \digamma \sqsubseteq_{\mathsf{e}} \alpha;\ \&^{\alpha}_{\mathbf{shr}}\ \mathbf{cell}(\tau), \tau) \to \tau$$

This is very similar to what we saw for *get*, except that after putting a copy of the old `Cell` content into r, we overwrite the `Cell` with what is stored at x. Also, this time we do not assume τ copy.

Following S-FN and SEM-FN-BODY, we have to show the following:

$$\left\{\begin{array}{l}\llbracket \digamma \sqsubseteq_{\mathsf{e}} \alpha \rrbracket * \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * \llbracket \mathit{ret} \triangleleft \mathbf{cont}(\digamma \sqsubseteq_{\mathsf{l}} [];\, r.\, r \triangleleft \mathbf{own}\ \tau) \rrbracket(t) * \\ \llbracket c \triangleleft \mathbf{own}\ \&^{\alpha}_{\mathbf{shr}}\ \mathbf{cell}(\tau), x \triangleleft \mathbf{own}\ \tau \rrbracket(t) * [\mathsf{NaInv} : t.\top] \end{array}\right\}$$
$$\texttt{call}\ \mathit{replace}_{\llbracket \tau \rrbracket.\mathrm{SIZE}}(c, x)\ \texttt{ret}\ \mathit{ret}$$
$$\{\mathsf{True}\}$$

(CELL-REPLACE)

The proof outline for this is shown in Figure 13.3.

The first part of this proof works the same as it did with `Cell::get`. The additional ownership of x is just carried along. The two proofs only start to diverge after $r :=_n {}^*c'$, where we make a copy of the `Cell`'s content and put it into the return value. Previously, we went on by closing the borrow. But if we did that now, we would have to give up ownership of $[\![\tau]\!].\textsc{own}(t, \bar{v}_{c'})$, which says that $\bar{v}_{c'}$ (the list of values that is now stored both in c' and r) is a valid instance of τ. Previously, τ was a `Copy` type so this proposition was persistent; that is not the case in `Cell::replace` though. Instead of closing the borrow again, we keep it open for now.

The next instruction is $c' :=_n {}^*x$. We own the memory pointed to by c' because we got it out of the borrow, and we own the memory pointed to by x because it was passed to us in an owned pointer, so we can satisfy the precondition of lrust-memcpy (page 133). *Now* we are ready to close the borrow again: the data stored at c' changed to $\bar{v}_x$, and we do own $[\![\tau]\!].\textsc{own}(t, \bar{v}_x)$ because we know x has type **own** τ. Together this means that the cell now contains a list of values that is valid for τ, and we can take all that ownership and put it back into the borrow.[15] After closing the borrow, we can also run the closing part of Lalive-sem to re-gain ownership of our lifetime contexts.

Next, we have to do some cleanup. $\texttt{delete}(1, c); \texttt{delete}([\![\tau]\!].\textsc{size}, x)$ deallocates the memory that its arguments were passed in; just like before we acquire the necessary precondition of lrust-delete from the fact that c and x were fully owned pointers (**own** _), which means we have full ownership of the memory and permission to deallocate it (own-ptr-own on page 135).

And finally, $\texttt{jump}\ ret(r)$ jumps to the return continuation. We have set up r to point to an instance of τ (the data we got out of c'), so the precondition $[\![r \lhd \textbf{own}\ \tau]\!](t)$ is easily satisfied. Other than that, we only need to pass on the lifetime context and non-atomic invariant token.

Remaining `Cell` methods. This completes the proof of the key `Cell` operations. The full API surface of `Cell` includes some more functions. Some of those are implemented entirely in safe code (and without accessing private fields) on top of `Cell::get` and `Cell::replace`, so from a safety standpoint we already covered them. This includes, in particular, `Cell::set`, which just calls `Cell::replace` and then throws away the old value. Beyond this, we have the following operations that all "do nothing" except for returning the input argument at a different type:

```
       Cell::new :  fn(T) -> Cell<T>
Cell::into_inner :  fn(Cell<T>) -> T
   Cell::get_mut :  fn(&mut Cell<T>) -> &mut T
  Cell::from_mut :  fn(&mut T) -> &Cell<T>
```

All of these are based on the fact that *ownership* of a `Cell<T>` is the same as ownership of `T` (cell-own), and thus the same goes for `&mut Cell<T>` and `&mut T`. These types are identical and can be freely converted back-and-forth.[16] `Cell` only becomes special once shared references are involved (cell-shr).

[15] Of course this means we lose ownership of $[\![\tau]\!].\textsc{own}(t, \bar{v}_x)$. That is expected; it reflects the fact that the content of x was *moved* (not just copied) into the cell. The only remaining operation we perform on x is to deallocate it, so we do not care any more which values are stored there.

[16] In Rust terms, they can be safely "transmuted", referring to the unsafe `transmute` operation which is an unchecked cast, like `reinterpret_cast` in C++.

The first three functions are direct realizations of this principle. The fourth function is not, but it follows easily: we can turn the argument of type `&mut T` into `&mut Cell<T>`, and then the shared reference `&Cell<T>` is obtained via normal sharing that can be performed safely with C-share.

Beyond what we have proven, two more methods have been added more recently: `Cell:swap` and `Cell::as_slice_of_cells`.

`Cell::swap` takes two `&Cell<T>` and swaps their content; proving it should just be a more tedious variant of `Cell::replace` where two `Cell` need to be opened at the same time (which we can do because the namespace incorporates the location of the `Cell`).

`Cell::as_slice_of_cells` is another function that reflects a "safe cast", but it is a more interesting cast than the ones we just discussed because it involves shared references. A *slice* is a dynamically sized array in Rust, and this method turns a "shared cell of an array" into a "shared array of cells". λ_{Rust} does not support dynamically sized types such as slices, but we can recast this method into an equivalent operation on pairs, turning a "shared cell of a pair" into a "shared pair of cells":

```
fn(&Cell<(T, U)>) -> &(Cell<T>, Cell<U>)
```

This method was only added to stable Rust in late 2019 (long after the initial publication of this work), so we did not consider it when designing the semantic model of λ_{Rust}. Unfortunately, it cannot easily be verified with the setup we have described so far: the location-dependent namespace in cell-shr becomes a problem here. The function `as_pair_of_cells` is supposed to return a reference to two separate cells at different locations (unless `T` has size 0), but the non-atomic borrow of `&Cell<(T, U)>` is tied to the location of the *product*:

$$\&_{\mathsf{na}}^{\kappa/t.\mathcal{N}_{\mathsf{shr}}.\ell}\left(\exists \bar{v}.\, \ell \mapsto \bar{v} * [\![\tau_1 \times \tau_2]\!].\textsc{own}(t, \bar{v})\right)$$

The ℓ here is the place in memory where $\tau_1 \times \tau_2$ *begins*, the first location that this compound data occupies in memory. To split this into two borrows, we would need the second borrow to have its namespace related to the place in memory where the *second* component begins. Changing the namespace of an already created borrow (or invariant) is impossible, so we are stuck here.

One very invasive option to make this work is to change what happens when sharing of $\textsf{\textbf{cell}}(\tau_1 \times \tau_2)$ starts: we could put each component of the pair into a separate borrow at the right location, so that `as_pair_of_cells` only has to split ownership of two already separate borrows. However, this requires some fundamental refactoring of our logical relation, and it is unclear how that would affect everything else. Verifying such a cast is thus left to future work.

13.2 Mutex

`Mutex` is the other example of interior mutability that we presented in §8.6. `Mutex<T>` uses a lock to safely grant multiple threads read and write access to a shared object of type `T`. The real `Mutex` uses platform-specific synchronization primitives but our model is implemented as a simple spinlock.

13.2.1 Semantic interpretation of `Mutex`

We start as usual by giving its size and ownership predicate:

$$[\![\textbf{mutex}(\tau)]\!].\textsc{size} := 1 + [\![\tau]\!].\textsc{size}$$

$$[\![\textbf{mutex}(\tau)]\!].\textsc{own}(t, \bar{v}) := [\![\textbf{bool} \times \tau]\!].\textsc{own}(t, \bar{v}) \qquad (\textsc{mutex-own})$$

When it is not shared, **mutex**(τ) is exactly the same as a pair of a **bool** (representing the status of the spinlock, and of an object of type τ (the content). ty-size holds trivially.

The sharing predicate is more complex: it cannot use fractured borrows, because we cannot afford getting only a *fraction* of ownership; and it cannot use non-atomic borrows, because mutexes are `Sync` and thus can be shared across thread boundaries (that is, of course, their entire purpose), but non-atomic borrows would tie the predicate to a particular thread identifier t as we have seen with **cell**(τ). Instead, it uses *atomic borrows* as introduced in §11.4 (page 148). These are basically normal Iris invariants (providing full access to their content for *atomic* operations in any thread), except that they are additionally tied to a lifetime so that the invariant gets "canceled automatically" when the lifetime ends.

For `Mutex`'s sharing predicate, we can use atomic borrows because the content of its borrow will only get accessed when changing the status of the lock, and doing so will require atomic memory accesses. Of course, this corresponds to the fact that, in our spinlock implementation, we are only using atomic sequentially consistent instructions to read or write the status flag. Using non-atomic accesses would lead to data races.

Thus we define the sharing predicate as follows:

$$[\![\textbf{mutex}(\tau)]\!].\textsc{shr}(\kappa, t, \ell) := \exists \kappa'.\, \kappa \sqsubseteq \kappa' * \&_{\mathsf{at}}^{\kappa/\mathcal{N}_{\mathsf{rust}}} M_\tau(\kappa', \ell, t) \qquad (\textsc{mutex-shr})$$

where

$$\begin{aligned} M_\tau(\kappa', \ell, t) := {}& \ell \mapsto \texttt{true} \lor \\ & \ell \mapsto \texttt{false} * \&_{\mathsf{full}}^{\kappa'}\big(\exists \bar{v}.\, (\ell + 1) \mapsto \bar{v} * [\![\tau]\!].\textsc{own}(t, \bar{v})\big) \end{aligned} \qquad (\textsc{mutex-inv})$$

This is quite a mouthful: first, we use an existential quantification at the beginning to close the predicate under shorter lifetimes, satisfying ty-shr-mono. We use an atomic borrow to share ownership of the status flag at location ℓ. This defines an invariant that is maintained *until κ ends*. For invariants like this that are required by the type system or unsafe libraries, we use the $\mathcal{N}_{\mathsf{rust}}$ namespace-[17] The invariant will come up a lot in the following, so we introduce the name $M_\tau(\kappa', \ell, t)$ for a mutex guarding data of type τ in thread t at location ℓ shared for lifetime κ'. This invariant can be in one of two states: In the first state, the flag is **`true`**, in which case the lock is locked and no other resource is stored in the borrow. Ownership of the content is currently held by whichever

[17] This namespace is disjoint from $\mathcal{N}_{\mathsf{lft}}$, and nothing else really matters about it.

thread acquired the lock. In the second state, the flag is **`false`**. This means the lock is unlocked, and the borrow also stores the ownership of the content of type τ at location $\ell + 1$. When acquiring or releasing the lock, we can atomically open the atomic borrow and change the branch of the disjunction, thus acquiring or releasing ownership of the content.

Curiously, ownership of the content is wrapped in a full borrow. One might expect instead that it should be directly contained in the outer atomic borrow. In that case, acquiring the lock would result in acquiring full (unborrowed) ownership of the content of the mutex. That, however, does not work: we would be unable to show TY-SHARE. Remember that to change the borrowed proposition, LFTL-BOR-ACC-STRONG requires us to prove a view shift *back* from the changed borrow to the original one (so that, at the end of the lifetime, the inheritance can be satisfied). Here, this means we would have to prove a view shift back from $M_\tau(\kappa', \ell, t)$ to $\exists \bar{v}.\, \ell \mapsto \bar{v} * [\![\textsf{bool} \times \tau]\!].\textsc{own}(t, \bar{v})$. But in the left case of the disjunction, we simply do not own the memory required by the right-hand side: we only own the location storing the Boolean flag, but we also need to own the memory storing the content of type τ.

To understand intuitively why this makes sense, imagine κ' ends while the lock is held by some thread that consequently owns the content.[18] In this case, ownership of the content would never be returned to the borrow. However, when κ' ends, the `Mutex` is again fully owned by someone, which means *they* expect to be the exclusive owner of the content! This is why the full borrow is necessary: when taking the lock, one gets the inner resource only under a borrow at lifetime κ', guaranteeing that ownership is returned when κ' ends.

Well-formedness. We focus on TY-SHARE, the only non-trivial part of well-formedness. Concretely, we need to show the following:

$$\&^{\kappa}_{\mathsf{full}}\big(\exists \bar{w}.\, \ell \mapsto \bar{w} * [\![\textsf{bool} \times \tau]\!].\textsc{own}(t, \bar{w})\big) * [\kappa]_q \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}}} \\ [\kappa]_q * \exists \kappa'.\, \kappa \sqsubseteq \kappa' * \&^{\kappa/\mathcal{N}_{\mathsf{rust}}}_{\mathsf{at}}\, M_\tau(\kappa', \ell, t)$$

In a first step, we split up the **bool** $\times\, \tau$ that we start with into its two components:

$$\&^{\kappa}_{\mathsf{full}}\big(\exists \bar{w}.\, \ell \mapsto \bar{w} * [\![\textsf{bool} \times \tau]\!].\textsc{own}(t, \bar{w})\big) * [\kappa]_q \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathsf{lft}}} \\ \&^{\kappa}_{\mathsf{full}}\big(\exists b.\, \ell \mapsto b * (b = \texttt{true} \vee b = \texttt{false})\big) * \\ \&^{\kappa}_{\mathsf{full}}\big(\exists \bar{v}.\, (\ell + 1) \mapsto \bar{v}[\![\tau]\!].\textsc{own}(t, \bar{v})\big) * [\kappa]_q$$

This part of the proof relies on LFTL-BOR-ACC-STRONG (page 155), BOOL-OWN (page 135), PAIR-OWN (page 135), LRUST-HEAP-APP (page 133), and LFTL-BOR-SPLIT (page 142).

In the second step, we use LFTL-BOR-ACC-STRONG again to open the borrow of the first component (the Boolean), and we pick $M_\tau(\kappa', \ell, t)$ for the new content Q of the borrow. The backwards direction from Q to the current content $\exists b.\, \ell \mapsto b * (b = \texttt{true} \vee b = \texttt{false})$ is easy: we throw away the inner borrow in case $\ell \mapsto \texttt{false}$. The forwards direction is also easy since we do in fact own that borrow of $\ell + 1$, so we can move it into the Boolean borrow if $b = \texttt{false}$ (and throw it away otherwise).[19]

With this, TY-SHARE is easily proven by picking $\kappa' := \kappa$.

[18] That is possible, for example, if the `MutexGuard` is leaked and hence its destructor never gets called.

[19] The only resources thrown away here are borrows, which is okay because the ownership they govern will be recovered when the lifetime ends and the inheritance gets used.

Marker traits. We also have to show that our interpretation adequately reflects the conditions under which `Mutex<T>` is `Copy`/`Send`/`Sync`. `Mutex<T>` is never `Copy`, so there is nothing to show there.[20]

However, `Mutex<T>` is *both* `Send` and `Sync` under the assumption that `T` is `Send`.[21] This corresponds to the fact that both the ownership and sharing predicate for **mutex**(τ) only use t for the ownership predicate of τ. So, if we have $\models \tau$ send and its ownership predicate does not depend on t, then we easily have both $\models$ **mutex**(τ) send and $\models$ **mutex**(τ) sync.

13.2.2 Semantic interpretation of `MutexGuard`

The Rust mutex library actually consists of two types: besides `Mutex<T>`, which represents a container guaranteeing mutual exclusion for accesses to some data of type `T`, we also have `MutexGuard<'a, T>`, which is the type returned by `Mutex::lock`. Owning this type reflects that we currently (for lifetime `'a`) have mutable access to some data of type `T` guarded by a `Mutex<T>`. `MutexGuard<'a, T>` acts basically like `&'a mut T`, except that the implicit destructor that runs when `MutexGuard<'a, T>` goes out of scope also releases the mutex it is associated with.

This means that on top of defining the semantic model of **mutex**(τ), we also have to define a model of **mutex_guard**(κ, τ), and use that model when verifying `Mutex::lock`. Operationally, a `MutexGuard<'a, T>` is a pointer to the `Mutex<T>` it is locking. Thus it has size 1 in λ_{Rust}. Ownership of a `MutexGuard<'a, T>` means that we are the party that successfully acquired the lock, *i.e.*, we own the full borrow that it is guarding. We also put the lock invariant (the atomic borrow) in here, because `MutexGuard` needs access to that to release the lock when it gets dropped.

$$\llbracket \textbf{mutex\_guard}(\kappa,\tau) \rrbracket.\textsc{size} := 1$$

$$\llbracket \textbf{mutex\_guard}(\kappa,\tau) \rrbracket.\textsc{own}(t,[\ell]) := \exists \kappa'.\ \kappa \sqsubseteq \kappa' * \&_{\mathsf{at}}^{\kappa/\mathcal{N}_{\mathsf{rust}}} M_\tau(\kappa',\ell,t) * \\ \&_{\mathsf{full}}^{\kappa'}\Big(\exists \bar{v}.\ (\ell+1) \mapsto \bar{v} * \llbracket \tau \rrbracket.\textsc{own}(t,\bar{v})\Big) \qquad (\textsc{mutexguard-own})$$

The full borrow here formally reflects the idea that `MutexGuard<'a, T>` is almost the same as `&'a mut T`: they both share this part of their interpretation.

Likewise, the type `&'b MutexGuard<'a, T>` is almost the same as `&'b &'a mut T`, also inheriting the complexities of sharing mutable references.[22] First of all, since `MutexGuard<'a, T>` itself is a pointer to a `Mutex<T>`, we need to make sure that when we share that pointer we can still dereference it: the pointer becomes read-only, and we use a sharing predicate similar to “simple types” in §10.4:

$$\llbracket \textbf{mutex\_guard}(\kappa,\tau) \rrbracket.\textsc{shr}(\kappa,t,\ell') := \exists \ell.\ \&_{\mathsf{frac}}^{\kappa}(\lambda q.\ \ell' \overset{q}{\mapsto} \ell) * \ldots$$

Remember that ℓ' here is the location at which the shared `MutexGuard` is stored, so it is a *pointer to a pointer*. ℓ is the location of the `Mutex` (same as the ℓ in the ownership predicate above).

To complete the sharing predicate, intuitively we have to say that at location $\ell + 1$, we can find a shared τ. This would be expressed as

[20] This is crucial for soundness: when a `Mutex` is currently locked, other threads must not read from it (and in particular make no copies of it) lest they cause a data race. This matches the fact that the sharing predicate does not actually own the content of the `Mutex` when the lock is held.

[21] Curiously, this is a quite literal type system translation of “`Mutex` adds synchronization to a type”.

[22] The complex sharing predicate is needed to verify soundness of `MutexGuard::deref`, which takes `&'b MutexGuard<'a, T>` as argument and returns `&'b T`. If it were not for that operation, we could use a trivial sharing predicate of True.

$\triangleright\, [\![\tau]\!].\mathrm{SHR}(\kappa \sqcap \kappa', t, \ell + 1)$.[23] We use lifetime $\kappa \sqcap \kappa'$ to incorporate both the lifetime for which the `MutexGuard<'a, T>` itself is valid (the `'a`) as well as the lifetime of the shared reference to this guard. However, just like with owned pointers and mutable references in §12.2, this definition fails to satisfy TY-SHARE. We have to use delayed sharing again: instead of sharing the instance of τ when we share the $\textbf{mutex\_guard}(\kappa, \tau)$, we make the sharing predicate of **mutex_guard** a *view shift that takes a step*, and only after completing the view shift we get the sharing predicate of τ. Thus, the sharing predicate of **mutex_guard** looks as follows:

$$\begin{aligned}[\![\textbf{mutex\_guard}(\kappa, \tau)]\!].\mathrm{SHR}(\kappa', t, \ell') := \exists \ell.\ \&^{\kappa}_{\mathsf{frac}}(\lambda q.\, \ell' \overset{q}{\mapsto} \ell) *{} \\ \Box\, \forall q.\, [[\![\kappa]\!] \sqcap \kappa']_q \mathrel{-\!\!*} {}^{\mathcal{N}_{\mathsf{shr}} \cup \mathcal{N}_{\mathsf{lft}}}\!\!\Rrightarrow^{\mathcal{N}_{\mathsf{lft}}} \triangleright\, {}^{\mathcal{N}_{\mathsf{lft}}}\!\!\Rrightarrow^{\mathcal{N}_{\mathsf{shr}} \cup \mathcal{N}_{\mathsf{lft}}} \\ \big([\![\tau]\!].\mathrm{SHR}([\![\kappa]\!] \sqcap \kappa', t, \ell + 1) * [[\![\kappa]\!] \sqcap \kappa']_q\big) \\ (\textsc{mutexguard-shr})\end{aligned}$$

Well-formedness. The proof of TY-SHARE proceeds in much the same way as it did in §12.2 for mutable references. The other properties required for semantic types pose no difficulties.

Marker traits. `MutexGuard<'a, T>` is not `Copy`, so no proof is required.

For `Send` and `Sync`, while our ownership predicate inherits its "sendabiltiy" from the interpretation of τ (so we could prove that $\models \tau\ \mathsf{send}$ implies $\models \textbf{mutex\_guard}(\kappa, \tau)\ \mathsf{send}$), the real `MutexGuard<'a, T>` is actually *never* `Send` because some platform-specific locking APIs do not permit releasing a lock from a different thread than the one that acquired it. However, `MutexGuard<'a, T>` is `Sync` under some conditions. In fact, when we performed this verification, `MutexGuard` was *always* `Sync`, no matter whether `T` was `Sync` or not—but if we try to prove $\models \textbf{mutex\_guard}(\kappa, \tau)\ \mathsf{sync}$, we will quickly notice that this requires $\models \tau\ \mathsf{sync}$. This is not a limitation in our model, it is a bug in Rust![24] We have submitted a fix to the Rust project such that `MutexGuard<'a, T>` is only `Sync` when `T` is, and it is easy to show the corresponding implication for our semantic model as defined above.

13.2.3 Soundness of the public operations

This completes the definitions of the types and verification of their basic properties. Now we take a close look at the two key operations of `Mutex`, acquiring and releasing the lock (`Mutex::lock` and `MutexGuard::drop`). We will also verify `MutexGuard::deref_mut`, which is the method that provides a mutable reference to the lock contents and thus enables actually accessing and mutating the data guarded by the lock. The types of these methods are as follows:[25]

```
          Mutex::lock :  fn<'a>(&'a Mutex<T>) -> MutexGuard<'a, T>
MutexGuard::deref_mut :  fn<'a, 'b>(&'b mut MutexGuard<'a, T>) -> &'b mut T
     MutexGuard::drop :  fn(MutexGuard<'_, T>)
```

[23] We need the $\triangleright$ because `MutexGuard<'a, T>` only uses `T` behind a pointer indirection, and thus serves as a "guard" for recursive types in Rust.

[24] Jung, "MutexGuard<Cell<i32>> must not be Sync", 2017 [Jun17].

[25] We are being a bit sloppy here with the type of `MutexGuard::drop`. The actual `drop` method one would write in Rust has type `fn(&mut MutexGuard<'_, T>) -> ()`. However, it cannot safely be *called* at that type, it can only be called by the implicit drop code that gets run when a variable of type `MutexGuard` goes out of scope. What *can* be safely called is the free function `drop`, and its type (specialized to `MutexGuard`) is indeed `fn(MutexGuard<'_, T>) -> ()`. Then, as usual, we omit the `()` return type.

Verifying `Mutex::lock`*.* The λ_{Rust} translation of this method, *lock*, looks as follows:

```
funrec lock(m) ret ret :=
  let m' = *m in
  delete(1, m);
  let r = new(1) in
  r := m';
  letcont loop(m', r) :=
    let b = CAS(m', false, true) in
    if b then jump ret(r) else jump loop(m', r)
   in jump loop(m', r)
```

And its type is:

$$\mathit{lock} : \forall\alpha.\, \mathbf{fn}(\digamma : \digamma \sqsubseteq_{\mathsf{e}} \alpha;\, \&^{\alpha}_{\mathsf{shr}}\, \mathbf{mutex}(\tau)) \to \mathbf{mutex\_guard}(\alpha, \tau)$$

In the beginning we have the usual preamble: *m* is a *pointer to* a shared reference to a **mutex**, so we get rid of the extra indirection. We also delete the memory the argument was passed in.[26] Then we allocate space for the return value, which has type **mutex_guard**, meaning it is just a pointer to the **mutex** it is guarding. Finally, we come to the actual meat of this function: *loop* implements a spinlock, which we use to model the platform-specific lock APIs that Rust actually uses under the hood. The loop keeps trying to compare-and-swap the lock state from `false` to `true`.[27] The `CAS` returns whether it succeeded, so once it returns `true` we know it successfully changed the lock state and thus we currently own the lock—so we are done and can return. Otherwise, we retry.

Following S-fn and sem-fn-body, we have to show the following:

$$\left\{\begin{array}{l} \llbracket \digamma \sqsubseteq_{\mathsf{e}} \alpha \rrbracket * \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * \\ \llbracket \mathit{ret} \triangleleft \mathbf{cont}(\digamma \sqsubseteq_{\mathsf{l}} [];\, r.\, r \triangleleft \mathbf{own}\ \mathbf{mutex\_guard}(\alpha, \tau)) \rrbracket(t) * \\ \llbracket m \triangleleft \mathbf{own}\ \&^{\alpha}_{\mathsf{shr}}\, \mathbf{mutex}(\tau) \rrbracket(t) * [\mathsf{NaInv} : t.\top] \end{array}\right\}$$
$$\texttt{call}\ \mathit{lock}(m)\ \texttt{ret}\ \mathit{ret}$$
$$\{\mathsf{True}\}$$

(mutex-lock)

The proof outline for this is shown in Figure 13.4.

The part before the loop is rather boring and very similar to what we have already seen in §13.1, so we focus on the loop itself, which is described by the following lemma:

$$\left\{\begin{array}{l} \llbracket \digamma \sqsubseteq_{\mathsf{e}} \alpha \rrbracket * \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * \\ \llbracket \mathit{ret} \triangleleft \mathbf{cont}(\digamma \sqsubseteq_{\mathsf{l}} [];\, r.\, r \triangleleft \mathbf{own}\ \mathbf{mutex\_guard}(\alpha, \tau)) \rrbracket(t) * \\ \mathsf{Dealloc}(r, 1, 1) * r \mapsto m' * \\ \alpha \sqsubseteq \kappa' * \&^{\alpha/\mathcal{N}_{\mathsf{rust}}}_{\mathsf{at}}\, M_{\tau}(\kappa', m', t) * [\mathsf{NaInv} : t.\top] \end{array}\right\}$$
$$\texttt{jump}\ \mathit{loop}(m', r)$$
$$\{\mathsf{True}\}$$

(mutex-lock-loop)

To prove this lemma, we apply hoare-rec (page 133), which means we may assume that the specification has already been proven for recursive

[26] We usually do this later, but here doing it early helps to keep the loop cleaner.

[27] The only reason we bind the result of the `CAS` to a name (*b*) is to later make the proof outline easier to write. Likewise, we would not usually make *m'* and *r* parameters of the loop continuation, but doing so makes it easier to state the loop invariant as a separate lemma.

Proof outline for MUTEX-LOCK.

$$\left\{\begin{array}{l} \llbracket \digamma \sqsubseteq_{\mathsf{e}} \alpha \rrbracket * \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * \Big((\mathbf{1})\ \llbracket \mathit{ret} \lhd \mathbf{cont}(\digamma \sqsubseteq_{\mathsf{l}} [];\, r.\, r \lhd \mathbf{own\ mutex\_guard}(\alpha, \tau)) \rrbracket(t) \Big) * \\ \llbracket m \lhd \mathbf{own}\ \&^{\alpha}_{\mathsf{shr}}\ \mathbf{mutex}(\tau) \rrbracket(t) * [\mathsf{NaInv} : t.\top] \end{array}\right\}_{\top}$$

Unfold $\llbracket \mathbf{own}\ \_ \rrbracket$.own (OWN-PTR-OWN, page 135).

Unfold $\llbracket \&^{\alpha}_{\mathsf{shr}}\ \_ \rrbracket$.own (SHR-REF-OWN) and $\llbracket \mathbf{mutex}(\tau) \rrbracket$.shr (MUTEX-SHR).

$\left\{ \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * m \mapsto \ell' * \mathsf{Dealloc}(m, 1, 1) * \triangleright(\alpha \sqsubseteq \kappa') * \triangleright \&^{\alpha/\mathcal{N}_{\mathsf{rust}}}_{\mathsf{at}}\ M_\tau(\kappa', \ell', t) * [\mathsf{NaInv} : t.\top] \right\}_{\top}$

$\texttt{let}\ m\text{'} = {}^{*}m\ \texttt{in}$ (LRUST-DEREF-NA, ℓ' becomes m')

$\left\{ \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * m \mapsto m\text{'} * \mathsf{Dealloc}(m, 1, 1) * \alpha \sqsubseteq \kappa' * \&^{\alpha/\mathcal{N}_{\mathsf{rust}}}_{\mathsf{at}}\ M_\tau(\kappa', m\text{'}, t) * [\mathsf{NaInv} : t.\top] \right\}_{\top}$

$\texttt{delete}(1, m);$ (LRUST-DELETE, consumes $m \mapsto m\text{'} * \mathsf{Dealloc}(m, 1, 1)$)

$\left\{ \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * [\mathsf{NaInv} : t.\top] \right\}_{\top}$

$\texttt{let}\ r = \texttt{new}(1)\ \texttt{in}$ (LRUST-NEW)

$\{ \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * r \mapsto v_r * \mathsf{Dealloc}(r, 1, 1) * [\mathsf{NaInv} : t.\top] \}_{\top}$

$r := m\text{'};$ (LRUST-ASSIGN-NA)

$\{ \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * r \mapsto m\text{'} * \big((\mathbf{2})\ \mathsf{Dealloc}(r, 1, 1) \big) * [\mathsf{NaInv} : t.\top] \}_{\top}$

$\texttt{letcont}\ \mathit{loop}(m\text{'}, r) := \ldots\ \texttt{in jump}\ \mathit{loop}(m\text{'}, r)$ (MUTEX-LOCK-LOOP)

Proof outline for MUTEX-LOCK-LOOP.

(We keep using the abbreviation numbers from above.)

Use HOARE-REC. Induction hypothesis: MUTEX-LOCK-LOOP

$\left\{ \llbracket \digamma \sqsubseteq_{\mathsf{e}} \alpha \rrbracket * \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * (2) * r \mapsto m\text{'} * \alpha \sqsubseteq \kappa' * \Big((\mathbf{3})\ \&^{\alpha/\mathcal{N}_{\mathsf{rust}}}_{\mathsf{at}}\ M_\tau(\kappa', m\text{'}, t) \Big) * [\mathsf{NaInv} : t.\top] \right\}_{\top}$

Use $\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q)$ with LALIVE-SEM to get $[\alpha]_{q'}$, closing view shift: $(\mathbf{4})\ [\alpha]_{q'} \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q)$.

With that token, open atomic borrow (3) via LFTL-AT-ACC-TOK (page 148), get closing view shift: $(\mathbf{5})\ \triangleright M_\tau(\kappa', m\text{'}, t)\ {}^{\top\setminus\mathcal{N}_{\mathsf{rust}}}\!\!\Rrightarrow\!\!\ast^{\top}\ [\alpha]_{q'}$. Unfold $M_\tau(\kappa', m\text{'}, t)$ (MUTEX-INV).

$$\left\{\begin{array}{l} (1) * (2) * r \mapsto m * \Big(m\text{'} \mapsto \texttt{true} \vee m\text{'} \mapsto \texttt{false} * \triangleright (\mathbf{6})\ \&^{\kappa'}_{\mathsf{full}} \big(\exists \bar{v}.\ (m\text{'} + 1) \mapsto \bar{v} * \llbracket \tau \rrbracket.\textsc{own}(t, \bar{v}) \big) \Big) * \\ (4) * (5) * [\mathsf{NaInv} : t.\top] \end{array}\right\}_{\top\setminus\mathcal{N}_{\mathsf{rust}}}$$

$\texttt{let}\ b = \texttt{CAS}(m\text{'}, \texttt{false}, \texttt{true})\ \texttt{in}$ (LRUST-CAS-INT)

$\left\{ (1) * (2) * r \mapsto m * m\text{'} \mapsto \texttt{true} * \Big(b = \texttt{false} \vee b = \texttt{true} * (6) \Big) * (4) * (5) * [\mathsf{NaInv} : t.\top] \right\}_{\top\setminus\mathcal{N}_{\mathsf{rust}}}$

Fold $M_\tau(\kappa', m\text{'}, t)$ with left disjunct (MUTEX-INV).

$\left\{ (1) * (2) * r \mapsto m * M_\tau(\kappa', m\text{'}, t) * \Big(b = \texttt{false} \vee b = \texttt{true} * (6) \Big) * (4) * (5) * [\mathsf{NaInv} : t.\top] \right\}_{\top\setminus\mathcal{N}_{\mathsf{rust}}}$

Close borrow (5), then lifetime context (4).

$\left\{ \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * (2) * r \mapsto m * \Big(b = \texttt{false} \vee b = \texttt{true} * (6) \Big) * [\mathsf{NaInv} : t.\top] \right\}_{\top}$

$\texttt{if}\ b$

Case $b = \texttt{true}$.

$\{ \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * (2) * r \mapsto m * (6) * [\mathsf{NaInv} : t.\top] \}_{\top}$

Fold $r \mapsto m * (2) * \alpha \sqsubseteq \kappa' * \&^{\alpha/\mathcal{N}_{\mathsf{rust}}}_{\mathsf{at}}\ M_\tau(\kappa', m\text{'}, t) * (6)$ into $\llbracket r \lhd \mathbf{own\ mutex\_guard}(\alpha, \tau) \rrbracket(t)$.

$\{ \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * \llbracket r \lhd \mathbf{own\ mutex\_guard}(\alpha, \tau) \rrbracket(t) * [\mathsf{NaInv} : t.\top] \}_{\top}$

$\texttt{then jump}\ \mathit{ret}(r)$ (by (1), using up all the other resources)

Case $b = \texttt{false}$.

$\{ \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * (2) * r \mapsto m * [\mathsf{NaInv} : t.\top] \}_{\top}$

$\texttt{else jump}\ \mathit{loop}(m\text{'}, r)$ (induction hypothesis MUTEX-LOCK-LOOP)

Figure 13.4: Proof outline for `Mutex::lock`.

jumps to *loop*. Another way to view this is that the precondition of this lemma becomes the loop invariant, and we have to re-establish that invariant before jumping back to the beginning of the loop.

The first and main operation inside the loop is the **CAS** on *m'*. To be able to verify this step, we need access to $m' \mapsto \_$, which we obtain by opening the atomic borrow (again using Lalive-sem to get hold of the appropriate lifetime token). **CAS** is an atomic operation, so we are permitted to open the atomic borrow around it, but we have to close the borrow again immediately after the operation completed. After opening the atomic borrow, we have two possible cases: either *m'* already points to **true**, so the **CAS** fails and nothing changes (someone else holds the lock and we will retry later), or else *m'* points to **false** and the **CAS** will succeed (we just acquired the lock). In the latter case, the atomic borrow also grants us ownership of $\&^{\kappa'}_{\mathsf{full}}\big(\exists \bar{v}.\, (m' + 1) \mapsto \bar{v} * [\![\tau]\!].\textsc{own}(t, \bar{v})\big)$, the full borrow managing the "content" of the lock. Either way we can use lrust-cas-int (page 133) to justify this operation, and afterwards we have $m' \mapsto \mathbf{true}$.[28] We use that ownership to close the atomic borrow again (so, we pick the left disjunct of $M_\tau(\kappa', \ell, t)$ when re-establishing the borrowed proposition). This means that if we previously acquired ownership of the lock content, we do *not* have to give it back! The value of b tells us if the **CAS** succeeded ($b = \mathbf{true}$) or not, and if it did, we additionally own the lock content.[29]

After completing the **CAS**, the code inspects the result b. If the **CAS** failed, we simply go around the loop once more, for which we make use of the "induction hypothesis" granted by hoare-rec. If the **CAS** succeeded, we own the lock content, which together with our remaining ownership is sufficient to establish that r satisfies the ownership predicate for **mutex_guard**: we have $[\![r \lhd \mathsf{own}\ \mathsf{mutex\_guard}(\alpha, \tau)]\!](t)$. This means we can safely return.

Verifying `MutexGuard::deref_mut`*.* Of course, acquiring the lock is only the first step in interacting with it. Next, the program will typically want to *access* the lock. This is done via `MutexGuard::deref_mut`, which returns a mutable reference to the lock content.[30]

The λ_{Rust} code for this method looks as follows:

```
funrec deref_mut(g) ret ret :=
  let g' = *g in
  delete(1, g);
  let m = *g' in
  let r = new(1) in
  r := m.1 ;
  jump ret(r)
```

And its type is:

deref_mut :

$$\forall \alpha, \beta.\, \mathbf{fn}(\digamma : \digamma \sqsubseteq_{\mathsf{e}} \beta, \beta \sqsubseteq_{\mathsf{e}} \alpha; \&^{\beta}_{\mathsf{mut}}\, \mathsf{mutex\_guard}(\alpha, \tau)) \to \&^{\beta}_{\mathsf{mut}}\, \tau$$

[28] This means immediately after the **CAS**, the lock is definitely held by *someone*, but not necessarily by us.

[29] This is an example of *conditional ownership transfer*: if the **CAS** succeeded, this operation transferred ownership of the lock content from the atomic borrow to us (reflecting that we now hold the lock); if the **CAS** failed, nothing got transferred.

[30] There is also `MutexGuard::deref` which turns `&'b MutexGuard<'a, T>` into `&'b T`. We have verified it in Coq as well, but do not consider it here because it is conceptually very similar to `deref_mut`, except for complications caused by delayed sharing.

In this type, $\beta \sqsubseteq_{\mathsf{e}} \alpha$ reflects the implicit well-formedness condition enforced by the Rust compiler that the referent type of a reference must outlive the lifetime of the reference.

The code for $\mathit{deref\_mut}$ is obscured mostly by the fact that there are so many pointer indirections going on. The argument g is a *pointer to* a $\&^{\beta}_{\mathsf{mut}}\, \textbf{mutex\_guard}(\alpha, \tau)$ (because all arguments get passed with an extra pointer indirection), which is a *pointer to* a **mutex_guard**, which is a *pointer to* a **mutex**. There are three indirections. We start by unfolding two of them, so m is a pointer to a **mutex**. The actual content of the lock is stored at offset 1,[31] so we compute that pointer via pointer arithmetic ($m.1$) and then return it in a freshly allocated r. We also have to clean up the memory at g that was used to pass our argument.

Following S-fn and sem-fn-body as usual, we have to show the following:

$$\left\{\begin{array}{l}[\![\digamma \sqsubseteq_{\mathsf{e}} \beta, \beta \sqsubseteq_{\mathsf{e}} \alpha]\!] * [\![\digamma \sqsubseteq_{\mathsf{l}} [\,]]\!](q) * \\ [\![\mathit{ret} \lhd \textbf{cont}(\digamma \sqsubseteq_{\mathsf{l}} [\,]; r.\, r \lhd \textbf{own}\, \&^{\beta}_{\mathsf{mut}}\, \tau)]\!](t) * \\ [\![g \lhd \textbf{own}\, \&^{\beta}_{\mathsf{mut}}\, \textbf{mutex\_guard}(\alpha, \tau)]\!](t) * [\mathsf{NaInv} : t.\top]\end{array}\right\}$$

$$\texttt{call}\ \mathit{deref\_mut}(g)\ \texttt{ret}\ \mathit{ret}$$

$$\{\mathsf{True}\}$$

(mutexguard-deref-mut)

The proof outline for this is shown in Figure 13.5.

The heart of the proof is about re-shaping the ownership predicate of a mutable reference to $\textbf{mutex\_guard}(\alpha, \tau)$ (mutexguard-own) into the ownership predicate of a mutable reference to τ. Unfolding the former, what we get is

$$\&^{\beta}_{\mathsf{full}}\Big(\exists g.\, g\text{'} \mapsto m * \exists \kappa'.\, \alpha \sqsubseteq \kappa' * \&^{\alpha/\mathcal{N}_{\mathsf{rust}}}_{\mathsf{at}}\, M_{\tau}(\kappa', m, t) * \\ \&^{\kappa'}_{\mathsf{full}}\big(\exists \bar{v}.\, (m+1) \mapsto \bar{v} * [\![\tau]\!].\textsc{own}(t, \bar{v})\big)\Big)$$

In the second line we can already see the mutable reference that we are looking for (this matches mut-ref-own exactly), but it is (a) nested within another borrow, and (b) at the wrong lifetime. Intuitively, a borrowed `MutexGuard<'_, T>` is like a doubly-borrowed `T`; what we have to do in this proof is flatten this nesting and show that the lifetimes work out.

For the first problem, we start by using LftL-bor-exists and LftL-bor-split (page 142) to push the borrow down over existential quantifications and separating conjunctions. In the end, we arrive at

$$\&^{\beta}_{\mathsf{full}}\, \&^{\kappa'}_{\mathsf{full}}\Big(\exists \bar{v}.\, (m+1) \mapsto \bar{v} * [\![\tau]\!].\textsc{own}(t, \bar{v})\Big)$$

(and some other propositions that we ignore here). Now we can use LftL-bor-unnest (page 155) to turn the nested borrow into a flat borrow at the intersected lifetime $\beta \sqcap \kappa'$. All that is left is to show $\beta \sqsubseteq \beta \sqcap \kappa'$, which follows from $\beta \sqsubseteq \alpha$ (a precondition of `MutexGuard::deref_mut`) and $\alpha \sqsubseteq \kappa'$ (part of the ownership predicate of $\textbf{mutex\_guard}(\alpha, \tau)$).

[31] Remember that at offset 0 we keep the Boolean flag controlling the spinlock.

Proof outline for MUTEXGUARD-DEREF-MUT.

$$\left\{\begin{array}{l}\llbracket \mathsf{F} \sqsubseteq_{\mathsf{e}} \beta, \beta \sqsubseteq_{\mathsf{e}} \alpha \rrbracket * \llbracket \mathsf{F} \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * \Big((\mathbf{1})\ \llbracket \mathit{ret} \lhd \mathbf{cont}(\mathsf{F} \sqsubseteq_{\mathsf{l}} []; r.\, r \lhd \mathbf{own}\ \&^{\beta}_{\mathbf{mut}}\, \tau) \rrbracket(t) \Big) * \\ \llbracket g \lhd \mathbf{own}\ \&^{\beta}_{\mathbf{mut}}\, \mathbf{mutex\_guard}(\alpha, \tau) \rrbracket(t) * [\mathsf{NaInv} : t.\top] \end{array}\right\}_{\top}$$

Unfold $\llbracket \mathbf{own}\ \_ \rrbracket$.own (own-ptr-own, page 135).

Unfold $\llbracket \&^{\beta}_{\mathbf{mut}}\ \_ \rrbracket$.own (mut-ref-own).

$$\left\{\begin{array}{l}\llbracket \mathsf{F} \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * g \mapsto \ell' * \big((\mathbf{2})\ \mathsf{Dealloc}(g, 1, 1) \big) * \\ \triangleright \&^{\beta}_{\mathsf{full}} \big(\exists \overline{w}.\, \ell' \mapsto \overline{w} * \llbracket \mathbf{mutex\_guard}(\alpha, \tau) \rrbracket.\mathrm{OWN}(t, \overline{w}) \big) * [\mathsf{NaInv} : t.\top] \end{array}\right\}_{\top}$$

$\mathtt{let}\ g' = {}^{*}g\ \mathtt{in}$ (lrust-deref-na, ℓ' becomes g')

$$\Big\{ \llbracket \mathsf{F} \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * g \mapsto g' * (2) * \&^{\beta}_{\mathsf{full}} \big(\exists \overline{w}.\, g' \mapsto \overline{w} * \llbracket \mathbf{mutex\_guard}(\alpha, \tau) \rrbracket.\mathrm{OWN}(t, \overline{w}) \big) * [\mathsf{NaInv} : t.\top] \Big\}_{\top}$$

$\mathtt{delete}(1, g);$ (lrust-delete, consumes $g \mapsto g' * (2)$)

$$\Big\{ \llbracket \mathsf{F} \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * \&^{\beta}_{\mathsf{full}} \big(\exists \overline{w}.\, g' \mapsto \overline{w} * \llbracket \mathbf{mutex\_guard}(\alpha, \tau) \rrbracket.\mathrm{OWN}(t, \overline{w}) \big) * [\mathsf{NaInv} : t.\top] \Big\}_{\top}$$

Use LftL-bor-exists (page 142) and then LftL-bor-split.

$$\Big\{ \llbracket \mathsf{F} \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * \&^{\beta}_{\mathsf{full}}\, g' \mapsto \bar{v}_{g'} * \&^{\beta}_{\mathsf{full}} \llbracket \mathbf{mutex\_guard}(\alpha, \tau) \rrbracket.\mathrm{OWN}(t, \bar{v}_{g'}) * [\mathsf{NaInv} : t.\top] \Big\}_{\top}$$

Unfold $\llbracket \mathbf{mutex\_guard}(\alpha, \tau) \rrbracket$.own (mutexguard-own).

Again use LftL-bor-exists and then repeatedly LftL-bor-split.

Throw away $\&^{\beta}_{\mathsf{full}}\, \&^{\alpha/\mathcal{N}_{\mathsf{rust}}}_{\mathsf{at}}\, M_{\tau}(\kappa', \ell'', t)$.

$$\left\{\begin{array}{l}\llbracket \mathsf{F} \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * \Big((\mathbf{3})\ \&^{\beta}_{\mathsf{full}}\, g' \mapsto \ell'' \Big) * \Big((\mathbf{4})\ \&^{\beta}_{\mathsf{full}}\, \alpha \sqsubseteq \kappa' \Big) * \\ \&^{\beta}_{\mathsf{full}}\, \&^{\kappa'}_{\mathsf{full}} \Big(\exists \bar{v}.\, (\ell + 1) \mapsto \bar{v} * \llbracket \tau \rrbracket.\mathrm{OWN}(t, \bar{v}) \Big) * [\mathsf{NaInv} : t.\top] \end{array}\right\}_{\top}$$

Use $\llbracket \mathsf{F} \sqsubseteq_{\mathsf{l}} [] \rrbracket(q)$ with Lalive-sem to get $[\beta]_{q'}$, closing view shift: $(\mathbf{5})\ [\beta]_{q'} \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \llbracket \mathsf{F} \sqsubseteq_{\mathsf{l}} [] \rrbracket(q)$.

Open (4) with $[\beta]_{q'}$ (LftL-bor-acc), copy $\triangleright \alpha \sqsubseteq \kappa'$, close it again (and throw the borrow away).

Open (3) with $[\beta]_{q'}$, get closing view shift: $(\mathbf{6})\ \triangleright g' \mapsto \ell'' \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \&^{\beta}_{\mathsf{full}}\, g' \mapsto \ell'' * [\beta]_{q'}$.

$$\Big\{ (1) * g' \mapsto \ell'' * \triangleright \alpha \sqsubseteq \kappa' * \&^{\beta}_{\mathsf{full}}\, \&^{\kappa'}_{\mathsf{full}} \Big(\exists \bar{v}.\, (\ell'' + 1) \mapsto \bar{v} * \llbracket \tau \rrbracket.\mathrm{OWN}(t, \bar{v}) \Big) * (5) * (6) * [\mathsf{NaInv} : t.\top] \Big\}_{\top}$$

$\mathtt{let}\ m = {}^{*}g'\ \mathtt{in}$ (lrust-deref-na, ℓ'' becomes m)

Close borrow (6), then lifetime context (5). Throw away $\&^{\beta}_{\mathsf{full}}\, g' \mapsto \ell''$.

$$\Big\{ \llbracket \mathsf{F} \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * \alpha \sqsubseteq \kappa' * \&^{\beta}_{\mathsf{full}}\, \&^{\kappa'}_{\mathsf{full}} \Big(\exists \bar{v}.\, (m + 1) \mapsto \bar{v} * \llbracket \tau \rrbracket.\mathrm{OWN}(t, \bar{v}) \Big) * [\mathsf{NaInv} : t.\top] \Big\}_{\top}$$

Use LftL-bor-unnest with wp-step over the next step of computation.

$\mathtt{let}\ r = \mathtt{new}(1)\ \mathtt{in}$ (lrust-new)

$$\Big\{ \llbracket \mathsf{F} \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * r \mapsto v_r * \mathsf{Dealloc}(r, 1, 1) * \&^{\beta \sqcap \kappa'}_{\mathsf{full}} \Big(\exists \bar{v}.\, (m + 1) \mapsto \bar{v} * \llbracket \tau \rrbracket.\mathrm{OWN}(t, \bar{v}) \Big) * [\mathsf{NaInv} : t.\top] \Big\}_{\top}$$

From $\llbracket \mathsf{F} \sqsubseteq_{\mathsf{e}} \beta, \beta \sqsubseteq_{\mathsf{e}} \alpha \rrbracket$ we have $\beta \sqsubseteq \alpha$, so with $\alpha \sqsubseteq \kappa'$ we have $\beta \sqsubseteq \kappa'$. Thus $\beta \sqsubseteq \beta \sqcap \kappa'$.

Use LftL-bor-shorten to change lifetime of full borrow to β.

$$\Big\{ \llbracket \mathsf{F} \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * r \mapsto v_r * \mathsf{Dealloc}(r, 1, 1) * \&^{\beta}_{\mathsf{full}} \Big(\exists \bar{v}.\, (m + 1) \mapsto \bar{v} * \llbracket \tau \rrbracket.\mathrm{OWN}(t, \bar{v}) \Big) * [\mathsf{NaInv} : t.\top] \Big\}_{\top}$$

$r := m.1\,;$ (lrust-assign-na)

Let $\ell_r := m + 1$.

$$\Big\{ \llbracket \mathsf{F} \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * r \mapsto \ell_r * \mathsf{Dealloc}(r, 1, 1) * \&^{\beta}_{\mathsf{full}} \Big(\exists \bar{v}.\, \ell_r \mapsto \bar{v} * \llbracket \tau \rrbracket.\mathrm{OWN}(t, \bar{v}) \Big) * [\mathsf{NaInv} : t.\top] \Big\}_{\top}$$

Fold $r \mapsto \ell_r * \mathsf{Dealloc}(r, 1, 1) * \&^{\beta}_{\mathsf{full}} \Big(\exists \bar{v}.\, \ell_r \mapsto \bar{v} * \llbracket \tau \rrbracket.\mathrm{OWN}(t, \bar{v}) \Big)$ into $\llbracket r \lhd \mathbf{own}\ \&^{\beta}_{\mathbf{mut}}\, \tau \rrbracket(t)$ (mut-ref-own).

$$\Big\{ \llbracket \mathsf{F} \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * \llbracket r \lhd \mathbf{own}\ \&^{\beta}_{\mathbf{mut}}\, \tau \rrbracket(t) * [\mathsf{NaInv} : t.\top] \Big\}_{\top}$$

$\mathtt{jump}\ \mathit{ret}(r)$ (by (1), using up all the other resources)

Figure 13.5: Proof outline for `MutexGuard::deref_mut`.

Proof outline for MUTEXGUARD-DROP.

$$\left\{\begin{matrix}\llbracket \digamma \sqsubseteq_{\mathsf{e}} \alpha \rrbracket * \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * \Big((\mathbf{1})\ \llbracket \mathit{ret} \lhd \mathbf{cont}(\digamma \sqsubseteq_{\mathsf{l}} []; r.\, r \lhd \mathbf{own}\ ()) \rrbracket(t) * \\ \llbracket g \lhd \mathbf{own\ mutex\_guard}(\alpha, \tau) \rrbracket(t) * [\mathsf{NaInv} : t.\top] \end{matrix}\right\}_{\top}$$

Unfold $\llbracket \mathbf{own}\ \_ \rrbracket$.OWN (OWN-PTR-OWN, page 135).

$\{\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * g \mapsto \ell' * \mathsf{Dealloc}(g, 1, 1) * \triangleright \llbracket \mathbf{mutex\_guard}(\alpha, \tau) \rrbracket.\textsc{own}(t, \ell') * [\mathsf{NaInv} : t.\top]\}_{\top}$

$\mathtt{let}\ m = {}^{*}g\ \mathtt{in}$ (LRUST-DEREF-NA, ℓ' becomes m)

$\{\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * g \mapsto m * \mathsf{Dealloc}(g, 1, 1) * \llbracket \mathbf{mutex\_guard}(\alpha, \tau) \rrbracket.\textsc{own}(t, m) * [\mathsf{NaInv} : t.\top]\}_{\top}$

$\mathtt{delete}(1, g);$ (LRUST-DELETE, consumes $g \mapsto m * \mathsf{Dealloc}(g, 1, 1)$)

$\{\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * \llbracket \mathbf{mutex\_guard}(\alpha, \tau) \rrbracket.\textsc{own}(t, m) * [\mathsf{NaInv} : t.\top]\}_{\top}$

Unfold $\llbracket \mathbf{mutex\_guard}(\alpha, \tau) \rrbracket$.OWN (MUTEXGUARD-OWN), throw away $\alpha \sqsubseteq \kappa'$.

$\Big\{\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * \Big((\mathbf{2})\ \&_{\mathsf{at}}^{\alpha/\mathcal{N}_{\mathsf{rust}}} M_{\tau}(\kappa', m, t)\Big) * \Big((\mathbf{3})\ \&_{\mathsf{full}}^{\kappa'}(\exists \bar{v}.\, (m+1) \mapsto \bar{v} * \llbracket \tau \rrbracket.\textsc{own}(t, \bar{v}))\Big) * [\mathsf{NaInv} : t.\top]\Big\}_{\top}$

Use $\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q)$ with LALIVE-SEM to get $[\alpha]_{q'}$, closing view shift: $(\mathbf{4})\ [\alpha]_{q'} \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q)$.

With this token, open atomic borrow (2) via LFTL-AT-ACC-TOK,

get closing view shift: $(\mathbf{5})\ \triangleright M_{\tau}(\kappa', m, t)\ {}^{\top\setminus\mathcal{N}_{\mathsf{rust}}}\!\Rrightarrow\!\!\ast^{\top}\ [\alpha]_{q'}$. Unfold $M_{\tau}(\kappa', m, t)$ (MUTEX-INV).

  $\Big\{(1) * \Big(m \mapsto \mathtt{true} \vee m \mapsto \mathtt{false} * \triangleright(3)\Big) * (3) * (4) * (5) * [\mathsf{NaInv} : t.\top]\Big\}_{\top\setminus\mathcal{N}_{\mathsf{lft}}}$

  $m :=_{\mathsf{sc}} \mathtt{false};$ (LRUST-ASSIGN-NA)

  $\Big\{(1) * \alpha \sqsubseteq \kappa' * \Big(m \mapsto \mathtt{false} \vee m \mapsto \mathtt{false} * (3)\Big) * (3) * (4) * (5) * [\mathsf{NaInv} : t.\top]\Big\}_{\top\setminus\mathcal{N}_{\mathsf{lft}}}$

  In both disjuncts, we have $m \mapsto \mathtt{false}$. Throw away the extra (3) if we are in the right disjunct.

  $\{(1) * m \mapsto \mathtt{false} * (3) * (4) * (5) * [\mathsf{NaInv} : t.\top]\}_{\top\setminus\mathcal{N}_{\mathsf{lft}}}$

  Fold $M_{\tau}(\kappa', m, t)$ with right disjunct (MUTEX-INV).

  $\{(1) * M_{\tau}(\kappa', m, t) * (4) * (5) * [\mathsf{NaInv} : t.\top]\}_{\top\setminus\mathcal{N}_{\mathsf{lft}}}$

  Close borrow (5), then lifetime context (4).

$\{\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * [\mathsf{NaInv} : t.\top]\}_{\top}$

$\mathtt{let}\ r = \mathtt{new}(0)\ \mathtt{in}$ (LRUST-NEW)

$\{\llbracket \digamma \sqsubseteq_{\mathsf{l}} [] \rrbracket(q) * (1) * r \mapsto \epsilon * \mathsf{Dealloc}(r, 0, 0) * [\mathsf{NaInv} : t.\top]\}_{\top}$

$\mathtt{jump}\ \mathit{ret}(r)$ (by (1), using up all the other resources)

Figure 13.6: Proof outline for `MutexGuard::drop`.

Verifying `MutexGuard::drop`. When the user does not need the lock any more, it has to be released again. This typically happens implicitly when the `MutexGuard` goes out of scope, at which point Rust calls `MutexGuard::drop`.

The λ_{Rust} code for this method looks as follows:

```
funrec drop(g) ret ret :=
  let m = *g in
  delete(1, g);
  m :=_sc false;
  let r = new(0) in
  jump ret(r)
```

And its type is:

$$\mathit{drop} : \forall \alpha.\, \mathbf{fn}(\digamma : \digamma \sqsubseteq_{\mathsf{e}} \alpha; \mathbf{mutex\_guard}(\alpha, \tau)) \to ()$$

The only interesting operation in this function is $m :=_{\mathsf{sc}} \mathtt{false}$, which sets the lock flag to **false** in a thread-safe way (*i.e.*, using an atomic access). The return type () has size 0, so we “allocate” it by asking for a zero-sized block of memory. Back in §9.2, we defined **new** to work correctly for all sizes, including 0.

Following S-fn and sem-fn-body, we have to show:

$$\left\{\begin{array}{l}[\![\digamma \sqsubseteq_{\mathsf{e}} \alpha]\!] * [\![\digamma \sqsubseteq_{\mathsf{l}} []]\!](q) * [\![\mathit{ret} \lhd \mathbf{cont}(\digamma \sqsubseteq_{\mathsf{l}} []; r.\, r \lhd \mathbf{own}\ ())]\!](t) * \\ [\![g \lhd \mathbf{own}\ \mathbf{mutex\_guard}(\alpha, \tau)]\!](t) * [\mathsf{NaInv} : t.\top]\end{array}\right\}$$

$$\mathtt{call}\ \mathit{drop}(g)\ \mathtt{ret}\ \mathit{ret}$$

$$\{\mathsf{True}\}$$

(mutexguard-drop)

The proof outline for this is depicted in Figure 13.6.

To verify the atomic release of the lock, we make use of the atomic borrow $\&_{\mathsf{at}}^{\alpha/\mathcal{N}_{\mathsf{rust}}} M_\tau(\kappa', m, t)$ that is part of the ownership predicate for **mutex_guard**(α, τ). After opening the borrow, we know that m stores either **`true`** (lock currently held) or **`false`** (lock currently not held).[32] We do not actually care which case we are in: either way, we can write **`false`** into m. To close the atomic borrow again, we need to reestablish $M_\tau(\kappa', m, t)$, and since $m \mapsto$ **`false`** this means we need to also provide ownership of the lock content. (Of course, this is exactly what one would expect when releasing a lock.) This ownership is part of $[\![$**mutex_guard**$(\alpha, \tau)]\!]$.own, so we can easily close the borrow and finish the proof.

This completes the interesting part of the correctness proof for `Mutex` and `MutexGuard`. There are a few more methods (such as the constructor `Mutex::new`), but their proofs do not demonstrate anything new. The dedicated reader can find them in our Coq development.[33]

[32] Given that we are holding the lock, it may seem strange that we cannot exclude the case $m \mapsto$ **`false`** here, which should be impossible. Indeed we could extend the mutex invariant M with a token to be able to prove that the lock must be held whenever we own a **mutex_guard**, but doing so would not actually make the proof any easier.

[33] `https://gitlab.mpi-sws.org/iris/lambda-rust`

Chapter 14

Related work

We discuss related work in two broad categories: other substructural type systems, and other work on Rust.

14.1 Substructural type systems for state

Over the past decades, various languages and type systems have been developed that use linear types,[1] ownership,[2] and/or regions[3] to guarantee safety of heap-manipulating programs. These include Cyclone,[4] Vault,[5] and Alms.[6] Much of this work has influenced the design of Rust, but a detailed discussion of that influence is beyond the scope of this dissertation.

In a related but very different line of work, systems like Ynot,[7] FCSL,[8] and F*[9] integrate variants of separation logic into dependent type theory. These systems are aimed at full functional verification of low-level imperative code and thus require a significant amount of manual proof and/or type annotations compared to Rust.

In the following, we focus on two key differences that set Rust and our proof apart from prior substructural languages and their soundness proofs: `unsafe` and `&mut T`.

- Most prior systems are *closed-world*, meaning that they are defined by a fixed set of rules and are proven sound using *syntactic* techniques.[10] As explained in §1.2, Rust's use of `unsafe` code to extend the scope of the type system fundamentally does not fit into this paradigm.
- Another unique feature of Rust are its mutable references (`&'a mut T`, or `&mut T` for short): mutable references are *unique* pointers associated with a *lifetime* (or *region*[11] as they are typically called in the literature) and they support *reborrowing*. Prior work typically permits all region pointers to be duplicated. As we have seen in §8.5, with reborrowing we can actually have multiple mutable references to the same target in scope, but only one of them may be used at any given point in the code.

Remarkably, while mutable references being unique is a core feature of Rust today, this was actually not the case for much of the early development of Rust (before the 1.0 release in 2015). The first steps towards uniqueness of mutable references were made in the seminal blog post "Imagine never hearing the phrase 'aliasable, mutable' again".[12] Only about a year before the release of Rust 1.0, mutable references became fully unique pointers by excluding read accesses through possible aliases.[13]

[1] Wadler, "Linear types can change the world!", 1990 [Wad90].

[2] Clarke, Potter, and Noble, "Ownership types for flexible alias protection", 1998 [CPN98].

[3] Fluet, Morrisett, and Ahmed, "Linear regions are all you need", 2006 [FMA06].

[4] Jim *et al.*, "Cyclone: A safe dialect of C", 2002 [Jim+02].

[5] DeLine and Fähndrich, "Enforcing high-level protocols in low-level software", 2001 [DF01].

[6] Tov and Pucella, "Practical affine types", 2011 [TP11].

[7] Nanevski *et al.*, "Ynot: Dependent types for imperative programs", 2008 [Nan+08].

[8] Nanevski *et al.*, "Communicating state transition systems for fine-grained concurrent resources", 2014 [Nan+14].

[9] Swamy *et al.*, "Dependent types and multi-monadic effects in F*", 2016 [Swa+16].

[10] Wright and Felleisen, "A syntactic approach to type soundness", 1994 [WF94].

[11] Fähndrich and DeLine, "Adoption and focus: Practical linear types for imperative programming", 2002 [FD02].

[12] Matsakis, "Imagine never hearing the phrase 'aliasable, mutable' again", 2012 [Mat12].

[13] Matsakis, "Rust RFC: Stronger guarantees for mutable borrows", 2014 [Mat14].

The work by Fluet, Morrisett, and Ahmed[14] on linear regions bears many similarities to the lifetime logic, with region capabilities corresponding to lifetime tokens and references corresponding to borrows. However, their approach permits combining mutation with aliasing; there is nothing like `&mut T`. This is not a problem because they consider neither interior pointers (where writing to one pointer can invalidate an aliasing pointer) nor concurrency. We believe that, to extend linear regions to handle Rust's unique borrows, one would end up needing something akin to our lifetime logic.

Semantic substructural soundness proofs. We are only aware of a few substructural type systems for which soundness has been proven semantically (using logical relations). These include $\mathbf{L}^3$,[15] λ^{URAL},[16] and the "superficially substructural" type system of Krishnaswami *et al.*[17] Ahmed *et al.*'s motivations for doing semantic soundness proofs were somewhat different from ours. One of their motivations was to build a foundation for substructural extensions to the Foundational Proof-Carrying Code project.[18] Another was to make it possible to modularly extend soundness proofs when building up the features of a language incrementally (although it is worth noting that Balabonski *et al.*[19] achieved similarly modular proofs for Mezzo using only syntactic methods). In contrast, following Krishnaswami *et al.*, we are focused on building a soundness proof that is "extensible" along a different axis, namely the ability to verify soundness of libraries that extend Rust's core type system through their use of unsafe features. Lastly, all of the prior semantic soundness proofs were done directly using set-theoretic step-indexed models, whereas in the present work, in order to model the complexities of Rust's lifetimes and borrowing, we found it essential to work at the higher level of abstraction afforded by Iris and our lifetime logic.

Gordon *et al.*[20] describe a type system for safe parallelism, based on qualifying references with attributes including `isolated` which describes a unique pointer. Similar to Rust (and unlike most other approaches), reference permissions apply transitively: when loading a reference through a reference, both permissions get combined to obtain the permission of the new reference (akin to S-deref-bor-own and S-deref-bor-mut on page 128). The authors specifically avoid the use of regions to make typechecking less dependent on lexical scopes.[21] This keeps the type system significantly simpler, and they can still support temporarily weakening uniqueness through *recovery rules*. However, recovery is weaker than reborrowing: accessing `isolated` fields requires destructive reads—incidentally, a pattern which was also widely used in Rust before mutable references were made unique. The soundness of their system is proven semantically in the Views framework,[22] but they do not explicitly discuss anything like `unsafe` code.

Cyclone. Cyclone[23] is a safe dialect of C, with a focus on maintaining programmer control over memory management. It uses regions as models of dynamic memory allocators ("arenas"), unlike in Rust where lifetimes are an entirely fictional concept of the type system. While Cyclone

[14] Fluet, Morrisett, and Ahmed, "Linear regions are all you need", 2006 [FMA06].

[15] Ahmed, Fluet, and Morrisett, "$\mathbf{L}^3$: A linear language with locations", 2007 [AFM07].

[16] Ahmed, Fluet, and Morrisett, "A step-indexed model of substructural state", 2005 [AFM05].

[17] Krishnaswami *et al.*, "Superficially substructural types", 2012 [Kri+12].

[18] Appel, "Foundational proof-carrying code", 2001 [App01].

[19] Balabonski, Pottier, and Protzenko, "The design and formalization of Mezzo, a permission-based programming language", 2016 [BPP16].

[20] Gordon *et al.*, "Uniqueness and reference immutability for safe parallelism", 2012 [Gor+12].

[21] Indeed the first generation of the Rust borrow checker regularly required tweaking the scope of a variable to please the lifetime analysis. However, things have gotten much better with *non-lexical lifetimes* [Mat16b], where the lifetime of a variable can end before it goes out of scope.

[22] Dinsdale-Young *et al.*, "Views: Compositional reasoning for concurrent programs", 2013 [Din+13].

[23] Grossman *et al.*, "Region-based memory management in Cyclone", 2002 [Gro+02]; Swamy *et al.*, "Safe manual memory management in Cyclone", 2006 [Swa+06].

also supports "ghost" regions (called `alias`), their purpose is to permit temporary aliasing of otherwise linear/unique pointers. Cyclone has no equivalent to `&mut T` in Rust, which provides temporary (controlled by a "ghost" region) *exclusive* access to `T`.

Cogent. Cogent[24] is a purely functional, linearly typed language designed to implement file systems and verify their functional correctness. Its linear type system permits efficient compilation to machine code using in-place updates, while the purely functional semantics enables equational reasoning. Its design is such that missing functionality can be implemented in C functions (much like `unsafe` code in Rust), which are given types to enforce correct usage in the Cogent program. These C functions are then manually verified to implement an equational specification and to follow the guarantees of the type system. However, the language and the type system are much simpler than Rust's (*e.g.*, there is no support for recursion, iteration, borrowing, or mutable state).

Mezzo. Mezzo[25] can be placed somewhere between the syntactic and the semantic approach. It comes with a substructural type system whose expressivity parallels that of a separation logic. Its soundness proof is modular in the sense that the authors start by verifying a core type system, and then add various extensions. This relies on an abstract notion of resources called *monotonic separation algebras*.[26] Nevertheless, Mezzo's handling of types remains entirely syntactic (*e.g.*, based on the grammar of types); there is no semantic account for types that would permit "adding" new types without revisiting the proofs.

In comparison with Rust, while Mezzo's type system decouples ownership from data, granting much more flexibility than Rust does, there are also some key patterns of Rust that are hard or impossible to represent in Mezzo. The authors describe a design pattern they call "borrowing" that is indeed serving a purpose very similar to Rust's mutable reference (`&mut T`). However, borrowing in Mezzo is *encoded* in the type system and requires the programmer to wrangle with "magic wands" encoded as closures. It is also unclear how to encode some more powerful uses of borrowing that Rust provides (such as mutable iteration over a `Vec`, which dynamically hands out mutable references to each element of the vector) in Mezzo. And finally, sharing works very differently in the two languages: in Mezzo, sharing requires irreversibly making the data immutable. In contrast, sharing in Rust is done using shared references. This means sharing is *temporary*, as controlled by the lifetime of that reference. This is a huge win for ergonomics, as it means that all methods which work on shared read-only data (such as `Vec::len` to determine the length of a vector) can also be used on a mutable reference or a fully owned vector.

14.2 Rust verification

Patina[27] is a formalization of the Rust type system, with accompanying partial proofs of progress and preservation. Being syntactic, these proofs do not scale to account for `unsafe` code. To keep our formalization

[24] O'Connor *et al.*, "Refinement through restraint: Bringing down the cost of verification", 2016 [OCo+16]; Amani *et al.*, "Cogent: Verifying high-assurance file system implementations", 2016 [Ama+16].

[25] Balabonski, Pottier, and Protzenko, "The design and formalization of Mezzo, a permission-based programming language", 2016 [BPP16].

[26] See §7.2 for a comparison to resource algebras in Iris.

[27] Reed, "Patina: A formalization of the Rust programming language", 2015 [Ree15].

feasible, we did not reuse the syntax and type system of Patina, but rather designed λ_{Rust} from scratch in a way that better fits Iris.

CRUST[28] is a bounded model checker designed to verify the safety of Rust libraries implemented using `unsafe` code. It checks that all clients calling up to n library methods do not trigger memory-safety faults. This provides an easy-to-use, automated way of checking `unsafe` code before attempting a full formal proof.[29] Their approach has successfully rediscovered some soundness bugs that had already been fixed in Rust's standard library. However, by only considering one library at a time, it cannot find bugs that arise from the interaction of multiple libraries.[30]

Lindner, Aparicius, and Lindgren[31] use symbolic execution in KLEE to automatically verify that safe libraries do not cause panics (an exception-like mechanism that Rust uses to signal fatal unexpected error conditions). Their tool can also detect precondition violations of some `unsafe` intrinsics, such as unchecked division (where division by zero causes undefined behavior), and thus help verify safe encapsulation of `unsafe` code.

An entirely separate line of work is concerned with exploiting Rust's strong type system to simplify formal verification of systems code. Prusti[32] translates Rust functions with user-defined annotations into Viper,[33] which is used as the underlying verification engine. Crucially, the translation exploits the ownership information encoded in the Rust type system so that the user just has to specify the values behind mutable or shared references, but does not need to be concerned with points-to facts. Ullrich[34] shows how to monadically embed a subset of safe Rust programs into the purely functional language of the Lean theorem prover with the goal of verifying functional correctness. RustHorn[35] translates Rust programs to constrained Horn clauses, also relying on the ownership system for this translation. The authors propose a novel treatment of mutable references in terms of prophecy variables,[36] entirely unlike what we have done with the lifetime logic in RustBelt. It will be interesting to explore a possible relation between the two approaches.

[28] Toman, Pernsteiner, and Torlak, "CRUST: A bounded verifier for Rust", 2015 [TPT15].

[29] However, their work contains a flaw where they skip verification of entirely safe methods: "Functions without unsafe code are ignored because their memory safety follows from the memory safety of the functions they call and the assumed correctness of Rust's memory safety analyses." Even entirely safe methods can violate invariants relied upon by other `unsafe` methods in the same module, so skipping them can lead to bugs being missed.

[30] Ben-Yehuda, "std::thread::JoinGuard (and scoped) are unsound because of reference cycles", 2015 [Ben15].

[31] Lindner, Aparicius, and Lindgren, "No panic! Verification of Rust programs by symbolic execution", 2018 [LAL18].

[32] Astrauskas *et al.*, "Leveraging Rust types for modular specification and verification", 2019 [Ast+19].

[33] Müller, Schwerhoff, and Summers, "Viper: A verification infrastructure for permission-based reasoning", 2016 [MSS16].

[34] Ullrich, "Simple verification of rust programs via functional purification", 2016 [Ull16].

[35] Matsushita, Tsukada, and Kobayashi, "RustHorn: CHC-based verification for Rust programs", 2020 [MTK20].

[36] Abadi and Lamport, "The existence of refinement mappings", 1991 [AL91].

Part III

Stacked Borrows

Stacked Borrows is an aliasing model for Rust which enables compilers to perform strong optimizations based solely on local (intraprocedural) type information. For background, examples, and key challenges, see the introduction in §1.3.

In the third and final part of this dissertation, we explain in detail how Stacked Borrows works, and we present the results of our evaluation of Stacked Borrows with Miri, where we check real Rust code for conformance with the model. This part assumes some knowledge of Rust; all necessary background is established in §8.

We begin in §15 by describing a basic version of Stacked Borrows for mutable and shared references (but without interior mutability). In §16, we show how to expand this model to support more advanced optimizations and interior mutability. We finish the description of the model in §17 by giving a formal definition. In §18, we report on our evaluation of Stacked Borrows, and in §19 we discuss related work.

Chapter 15

Uniqueness and immutability

To briefly recall what we discussed in §1.3, the goal of Stacked Borrows is to enable the Rust compiler to exploit the strong alias information that is encoded in the Rust type system: depending on the type of a reference, it either guarantees that *no aliases* to this reference exist (for `&mut T`), or that all aliases that could exist are *read-only* (for `&T`). This information is useful not only to ensure memory and thread safety, but also to inform the optimizer and support static analysis of Rust code. The problem is that `unsafe` code can, in principle, violate these guarantees—and due to the importance of `unsafe` code in the Rust ecosystem, it is crucial for optimizations to be correct even in the presence of `unsafe` code. To enable sound type-based alias analysis of all Rust code, we propose a set of rules that restrict aliasing in `unsafe` code: *Stacked Borrows.* Proof sketches establish that these rules are sufficient to enable interesting optimizations.[1] At the same time the rules are compatible with real-world `unsafe` Rust code, which is demonstrated by running that code in an interpreter equipped with an implementation of Stacked Borrows (we will get to this in §18).

The key idea of Stacked Borrows is to take the static analysis that the borrow checker performs, which uses lifetimes (§8.5), and develop from it a *dynamic analysis* that does *not use lifetimes.* Then we can require that even programs using `unsafe` code must satisfy this dynamic analysis, which gives the compiler license to assume that as a fact during optimization. Safe programs should trivially satisfy the dynamic analysis, because it is meant to be strictly more liberal than the borrow checker.

Avoiding the use of lifetimes in Stacked Borrows was a deliberate choice. Lifetimes are in many cases *inferred* by the compiler, even in `unsafe` code, and whether that code is considered well-behaved should not depend on the mercurial details of the compiler's lifetime inference. In particular, this inference has recently undergone significant change—with the switch from the old AST-based borrow checker to non-lexical lifetimes ("NLL")[2]—and it is planned to change again as part of an ongoing project called "Polonius".[3] Such changes should not affect the way people have to think about their `unsafe` code. Moreover, lifetimes are erased from the program during the Rust compilation process immediately after borrow checking is done, so the optimizer does not even have access to that information. Thus practically speaking, making Stacked Borrows depend on lifetimes would not be useful for analyses performed by the compiler.

[1] Like all formal results presented in this dissertation, these proof sketches have been formalized in Coq.

[2] Klock, "Breaking news: Non-lexical lifetimes arrives for everyone", 2019 [Klo19].

[3] Matsakis, "An alias-based formulation of the borrow checker", 2018 [Mat18].

We will build up the Stacked Borrows semantics incrementally, beginning in §15.1 with the simplest possible version: suppose Rust had neither shared references nor raw pointers,[4] only mutable references. We discuss the operational semantics of such a restricted form of Stacked Borrows in §15.2. In §15.3, we will add raw pointers to show how our operational semantics rules out misbehaving uses of raw pointers—as we demonstrate in §15.4, where we establish the correctness of the example optimization from the introduction (§1.3). Finally, we extend the model to also handle shared references (§15.5), and we present an optimization enabled by this (§15.6).

15.1 Mutable references in a stack

In this simplified language with only mutable references, what does a dynamic version of the borrow checker look like? To reiterate, the borrow checker ensures

1. that a reference and all references derived from it can only be used during its lifetime, and
2. that the original referent is not used until the lifetime of the loan has expired.

We can re-phrase this property without referring to lifetimes, by saying that *every use of the reference (and everything derived from it) must precede the next use of the original referent.* For reasons that will become clear shortly, we call this the *stack principle.* This is equivalent to what the borrow checker does, if we think of the lifetime as ending some time between the last use of the reference and the next use of the referent.

So, let us consider the following example:

```
1  let mut local = 0;
2  let x = &mut local;
3  let y = &mut *x; // Reborrow x to y.
4  *x = 1; // Use x again.
5  *y = 2; // Error! y used after x got used.
```

This program violates the stack principle because a use of the reference y occurs in line 5 *after* the next use of referent x in line 4 (“next” after y has been created). It hence gets rejected by the borrow checker.[5]

If we look at the usage patterns of variables here, we can see that we start by creating x, then we create y, then we use x again—and thereafter, we use y again, which is the point where the code violates the stack principle. This demonstrates that the stack principle enforces a *well-nested* usage of references: the use of the derived reference y must be “nested between” other uses of x and cannot be arbitrarily interleaved. The “XYXY” sequence of usages violates this idea of nesting. As always when things are well-nested, that indicates a *stack discipline*, and indeed we will think of these aliasing references as being organized on a stack.

At a high level, the way Stacked Borrows rejects this program is by tracking a stack of references that are allowed to access local. The stack tracks which borrows of local may still be used, so we also call it the *borrow stack* to disambiguate it from, *e.g.*, the call stack. Newly created references are pushed on the top of the borrow stack, and we enforce that

[4] As explained at the end of §8.5, raw pointers are unchecked pointers that can only be used in unsafe code.

[5] For simple integer types rejecting this program might seem silly, but the analysis is supposed to work for all types, and in §8 we have seen how programs can go wrong if we use more complex types such as Vec<i32> where a mutation can invalidate existing pointers.

$$
\begin{aligned}
t \in \mathit{Tag} &:= \mathbb{N} & \mathit{Scalar} &:= \mathsf{Pointer}(\ell, t) \mid z \qquad \text{where } z \in \mathbb{Z} \\
\mathit{Item} &:= \mathsf{Unique}(t) \mid \ldots & \mathit{Mem} &:= \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Scalar} \times \mathit{Stack} \\
\mathit{Stack} &:= \mathit{List}(\mathit{Item})
\end{aligned}
$$

Figure 15.1: Stacked Borrows domains (preliminary).

after a reference gets used, it is again at the top of the stack. To this end, all references above it are popped, which permanently invalidates them—references not in the stack may not be used at all. So, line 3 pushes `y` on top of `x` in that stack, but line 4 removes `y` again because we used `x`. Thus, in line 5, `y` is no longer in the stack, and hence its use is found to be a violation of the stack principle, and therefore undefined behavior.

15.2 An operational model of the borrow checker

To make this idea more precise, we first have to be able to distinguish different references that point to the same memory. `x` and `y` in the previous example program point to the same location, but to explain what is going on we have to tell them apart in our semantics. So we assume that every reference is *tagged* by some unique "pointer ID" t that gets picked when a reference is created, and is preserved as the reference is copied around. Formally (see Figure 15.1), we say that a *pointer value* $\mathsf{Pointer}(\ell, t)$ consists of a location ℓ in memory that the pointer points to, and a tag t identifying the pointer/reference. Both references and raw pointers are represented as pointer values at run-time.

In memory, we then store for every location ℓ a *borrow stack* of tags identifying the references that are allowed to access this location. To prepare for future extensions of the model in the following sections, we call the elements of the stack *items*; so far, $\mathsf{Unique}(t)$ holding a tag t is the only kind of item. Of course we also store the value that the memory holds at this location; in our case, such primitive values (integers and pointers) are called *scalars*.

The operational semantics acts on the tags and borrow stacks as follows:

Rule (new-mutable-ref)**.** Any time a new mutable reference is created (via `&mut expr`) from some existing pointer value $\mathsf{Pointer}(\ell, t)$ (the referent), first of all this is considered a *use* of that pointer value (so we follow use-1 below). Then we pick some fresh tag t'. The new reference has value $\mathsf{Pointer}(\ell, t')$, and we push $\mathsf{Unique}(t')$ on top of the stack for ℓ.

Rule (use-1)**.** Any time a pointer value $\mathsf{Pointer}(\ell, t)$ is *used*, an item with tag t must be in the stack for ℓ. If there are other tags above it, we pop them, so that the item with tag t is on top of the stack afterwards.[6] If $\mathsf{Unique}(t)$ is not in the stack at all, this program has undefined behavior.

These rules reflect the stack principle: a newly created reference may only be used as long as its item is in the stack, which is until the next time the referent it got *created from* is used.

We can see these rules in action in the following annotated version of the previous example, where we spell out which reference has which

[6] Later it will be possible for a tag to occur multiple times in the stack; in that case we use the topmost item, *i.e.*, we pop as few other items as possible.

pointer value at run-time, and how the borrow stack of the memory storing `local` evolves over time. (Borrow stacks are shown bottom-to-top, *i.e.*, the left end is the bottom of the stack.) We assume that `local` is allocated at address ℓ, and $h \in \mathit{Mem}$ refers to the current memory at the given program point. Direct accesses to a local variable also use a tag; we assume that tag is 0 here.

```
let mut local = 42; // Stored at location ℓ, and with tag 0.
// The initial stack: h(ℓ) = (42, [Unique(0)])
let x = &mut local; // = Pointer(ℓ, 1)
// Use local, push tag of x (1) on the stack: h(ℓ) = (42, [Unique(0), Unique(1)]). (NEW-MUTABLE-REF)
let y = &mut *x; // = Pointer(ℓ, 2)
// Use x, push tag of y (2): h(ℓ) = (42, [Unique(0), Unique(1), Unique(2)]). (NEW-MUTABLE-REF)
*x += 1;
// Pop tag of y (2) to bring tag of x (1) to the top: h(ℓ) = (43, [Unique(0), Unique(1)]). (USE-1)
*y = 2;
// Undefined behavior! Stack principle violated: tag of y (2) is not in the stack. (USE-1)
```

15.3 Accounting for raw pointers

We are almost ready to come back to the example from the introduction (§1.3):

```
fn example1(x: &mut i32, y: &mut i32) -> i32 {
  *x = 42;
  *y = 13;
  return *x; // Has to read 42, because x and y cannot alias!
}

fn main() {
  let mut local = 5;
  let raw_pointer = &mut local as *mut i32;
  let result = unsafe {
    example1(&mut *raw_pointer, &mut *raw_pointer)
  };
  println!("{}", result); // Prints "13".
}
```

Remember, our goal is to explain how and where that program violates the "dynamic borrow checker" that is Stacked Borrows. We need this program to be a violation of Stacked Borrows, because it would otherwise constitute a counterexample to the desired optimization of making `example1` always return `42`.

So far, however, our model only deals with mutable references. We have to explain what happens when raw pointers are created via a cast (`expr as *mut T`). Just like the borrow checker does not do any tracking for raw pointers, Stacked Borrows also makes no attempt to distinguish different raw pointers pointing to the same thing: raw pointers are *untagged*. So we extend our set *Tag* of tags with a $\bot$ value to represent untagged pointers.

Raw pointers are added to the borrow stack in the same way as mutable references. The idea is that an "XYXY" is allowed when *both* "X" and "Y" are raw pointers, but if either one of them is a mutable reference, we want that to still be a violation of the stack principle. This way, when performing analyses for compiler optimizations, we can be sure that mutable references are never part of an "XYXY" pattern of memory accesses.

To track raw pointers in the borrow stack, we add a second kind of items that can live in the stack: SharedRW (short for "shared read-write") items indicate that the location has been "shared" and is accessible to all raw (untagged) pointers for reading and writing. Overall, we now have:

$$t \in \mathit{Tag} := \mathbb{N} \cup \{\bot\} \qquad \mathit{Item} := \mathsf{Unique}(t) \mid \mathsf{SharedRW} \mid \ldots$$

We amend the operational semantics as follows, where USE-2 replaces USE-1:[7]

Rule (NEW-MUTABLE-RAW-1)**.** Any time a mutable raw pointer is created by casting (`expr as *mut T`) a mutable reference (`expr: &mut T`) with value $\mathsf{Pointer}(\ell, t)$, first of all this is considered a *use* of that mutable reference (see USE-2). Then the new raw pointer has value $\mathsf{Pointer}(\ell, \bot)$, and we push SharedRW on top of the borrow stack for ℓ.

Rule (USE-2)**.** Any time a pointer value $\mathsf{Pointer}(\ell, t)$ is used, if t is $\bot$ then SharedRW must be in the stack for ℓ; otherwise $\mathsf{Unique}(t)$ must be in the stack. If there are other tags above that item, we pop them. In case of ambiguity (when there are several items in the stack that match t), use the topmost item, *i.e.*, pop as few items as possible. If the desired item is not in the stack at all, we found a violation of the stack principle.

Note how for tagged pointer values (*i.e.*, mutable references), USE-2 is the same as USE-1.

With this, the example program from the introduction executes as follows (we reordered the functions so that this can be read top-to-bottom):

```
 1  fn main() {
 2    let mut local = 5; // Stored at location ℓ, and with tag 0, h(ℓ) = (5, [Unique(0)]).
 3    let raw_pointer = &mut local as *mut i32; // = Pointer(ℓ, ⊥)
 4    // The temporary reference gets tag 1, and is pushed: h(ℓ) = (5, [Unique(0), Unique(1)]).
 5    // Then the raw pointer is pushed: h(ℓ) = (5, [Unique(0), Unique(1), SharedRW]).
 6    // (NEW-MUTABLE-REF, NEW-MUTABLE-RAW-1)
 7    let result = unsafe { example1(
 8      &mut *raw_pointer, // = Pointer(ℓ, 2)
 9      // Reference is pushed on top of the raw pointer: h(ℓ) = (5, [. . . , SharedRW, Unique(2)]).
10      // This uses the raw pointer! (NEW-MUTABLE-REF)
11      &mut *raw_pointer // = Pointer(ℓ, 3)
12      // Using raw_pointer here pops the first reference off the stack: h(ℓ) = (5, [. . . , SharedRW, Unique(3)]).
13      // This uses the raw pointer! (NEW-MUTABLE-REF)
14    ) }; // Next: jump to example1 (line 17).
15    println!("{}", result); // Prints "13".
16  }
17  fn example1(x: &mut i32, y: &mut i32) -> i32 {
18    // x = Pointer(ℓ, 2), y = Pointer(ℓ, 3), h(ℓ) = (5, [Unique(0), Unique(1), SharedRW, Unique(3)])
19    *x = 42;
20    // Analysis error! Tag of x (which is 2) is not in the stack. Program has undefined behavior.
21    *y = 13;
22    return *x; // We want to optimize this to return 42.
23  }
```

[7] For now, SharedRW behaves the same as $\mathsf{Unique}(\bot)$; this will change later when we add support for shared references.

The key point in this execution is when the two references that get passed to `example1` are created: each time we execute `&mut *raw_pointer`, NEW-MUTABLE-REF says this is *using* `raw_pointer` and as such we make sure that SharedRW is at the top of the borrow stack (USE-2). When creating the second reference (later called `y`), we have to pop off Unique(2), rendering the first reference (`x`) unusable! When `x` *does* get used in line 19, this is detected as a violation of the stack principle.

It may seem strange that a reborrow of a reference (as opposed to actually accessing memory) counts as a "use" of the old reference (the operand of the reborrow), but that is important: if we did not do so, after the second `&mut *raw_pointer`, we would end up with the following stack for ℓ:

$$[\mathsf{Unique}(0), \mathsf{Unique}(1), \mathsf{SharedRW}, \mathsf{Unique}(2), \mathsf{Unique}(3)]$$

This borrow stack encodes that the reference tagged 3 is somehow "nested within" the reference tagged 2, and 3 may only be used until 2 is used the next time—which are the rules we want when 3 is reborrowed from 2, but that was not the case! To make sure that the stack adequately reflects which pointer values were created from which other pointer values, we make merely creating a reference "count" as a write access of the old reference (the operand). Thus, the stack will instead be $[\dots, \mathsf{SharedRW}, \mathsf{Unique}(3)]$, which encodes the fact that 3 was created from a raw pointer and not from 2. In future work, we would like to explore other models that track the precise pointer inheritance information in a *tree*, instead of a stack.

15.4 Retagging, and a proof sketch for the optimization on mutable references

The next step in developing Stacked Borrows would be to try and convince ourselves that we have not just ruled out *one* counterexample for the desired optimization in `example1`, but *all possible* counterexamples. However, it turns out that this would be a doomed enterprise: the semantics is not yet quite right!

In particular, if `example1` is called with two pointer values *carrying the same tag*, then the dynamic analysis as described so far does not have any problem with both of them aliasing. When `x` and `y` have the same tag, the function behaves as if all three accesses were done using `x`, and there is no Stacked Borrows violation. Such duplicate tags are possible because unsafe code can make copies of any data,[8] including mutable references.

The problem is that, when we analyze `example1` in an unknown context, the tags of `x` and `y` are provided by the caller and thus untrusted. To be able to reason based on tags and borrow stacks, we need to be sure that both references have unique tags that are not used by any other pointer value in the program.[9] This is achieved by inserting *retagging* instructions. `retag` is an administrative instruction that makes sure that references have a fresh tag. It is automatically inserted by the compiler—in particular, all arguments of reference type are retagged immediately when the function starts executing:

[8] This can be achieved, for example, using `transmute_copy`, or by reading with a raw pointer.

[9] Unsafe code can *duplicate* tags but it cannot *forge* them—there is no language operation for that.

```
1  fn example1(x: &mut i32, y: &mut i32) -> i32 {
2    retag x; // equivalent to: ‘x = &mut *x;‘
3    retag y; // equivalent to: ‘y = &mut *y;‘
4    *x = 42;
5    *y = 13;
6    return *x; // We want to optimize this to return 42.
7  }
```

As indicated in the comments, retagging achieves the desired effect by basically performing a reborrow. In the fragment of Rust we have considered so far, `retag x` behaves exactly like `x = &mut *x`, which means `x` still points to the same location, but we follow the usual Stacked Borrows steps for both using (the old value of) `x` and creating a new reference (use-2, new-mutable-ref).

Now we are finally ready to give a proof sketch for why performing the desired optimization in line 6 of this program is correct. If the program does not conform to the Stacked Borrows discipline, then there is nothing to show because the program is considered to have undefined behavior. So we proceed under the assumption that the program does conform to Stacked Borrows:

1. Let us say that after the `retag` in line 2, `x` has value $\mathsf{Pointer}(\ell, t)$. We know that no other pointer value has tag t, and that this tag is at the top of ℓ's borrow stack.
2. `x` is not used until line 4, so *if* any code in between has any effect on the value or the stack of ℓ, it will do that through a pointer value with a different tag. That tag must be below t in the stack, because we established in (1) that t is at the top. This means that using a pointer value with a different tag will pop t off the stack. However, for line 4 to pass the analysis, t must still be in the stack. Thus, ℓ could *not* have been accessed in between line 2 and line 4, and t must still be at the top of ℓ's stack. After executing line 4, the stack remains unchanged; but we now know that the value stored at ℓ is 42.
3. Finally, in line 6, we can repeat the same argument again to show that *if* t is still in ℓ's stack, no access to ℓ could have occurred in the meantime. Hence, we can conclude that ℓ still contains value 42, and we can perform the desired optimization.

What is interesting about this argument is that we never argued explicitly about whether `x` and `y` are aliasing or not! As a result, this particular optimization can be generalized to an entire optimization *pattern*, where the access of `y` is replaced by *any* code that does not mention `x`:

```
1  fn example1(x: &mut i32, /* more arguments */) -> i32 {
2    retag x;
3    /* replace this line with any code not using x */
4    *x = 42;
5    /* replace this line with any code not using x */
6    return *x; // We want to optimize this to return 42.
7  }
```

This pattern demonstrates that `x` truly is a “unique pointer”: code not using `x` cannot possibly have an effect on the memory `x` points to.

In particular, the optimization also applies when the code not using x is a call to an *unknown function*. That is, Stacked Borrows allows us to do something that is unthinkable in the typical C/C++ compiler: we have a reference, x, that was passed in by the environment,[10] and still, we can call a foreign function f(), and *as long as we do not pass x as an argument to f*, we can assume that f neither reads from nor writes to x. Furthermore, we can make this assumption without doing any inlining, using only intraprocedural reasoning—a "holy grail" of alias analysis.

When to retag? The **retag** in the previous example was crucial because it allowed the compiler to assume that x actually had a unique tag. This optimization, as well as all of the ones we are going to see, can only be performed on references that have been retagged "locally" (within the function under analysis), because only then we can make the necessary assumptions about the tag of the reference and the borrow stack of the location it points to. So where exactly **retag** is inserted becomes an important knob in Stacked Borrows that determines for which references optimizations like the one above can be performed.

For this version of Stacked Borrows, we perform retagging any time a reference is passed in as an argument, returned from a function, or read from a pointer. Basically, any time a reference "enters" our scope, it is retagged:

```
fn retag_demo(x: &mut i32, y: &&i32) {
  retag x; // All function arguments of reference type...
  retag y; // ...are retagged.
  let z = *y;
  retag z; // We also retag references read from a pointer...
  let incoming_ref = some_function_returning_a_ref();
  retag incoming_ref; // ...and references returned to us.
}
```

Reordering instructions. To catalog the various transformations that Stacked Borrows enables, it will be helpful to phrase them as *reordering* certain instructions. For example, the optimization we just considered is equivalent to reordering a load from a mutable reference up across the unknown code (which we represent here as a call to some externally defined functions f and g):

```
1  retag x;                    1  retag x;
2  g();                        2  g();
3  *x = 42;                    3  *x = 42;
4  f();                   ⇒    4  let retval = *x;
5  let retval = *x;            5  f();
6  return retval;              6  return retval;
```

The only difference between the left-hand side and the right-hand side is that we swapped lines 4 and 5. It is now easy to argue that the load in line 4 immediately follows a store of 42 to the same location, so retval will necessarily be 42.[11]

[10] This is relevant because it means the pointer value is "escaped" in the sense that it is known to the environment, and unknown code could *in principle* (but not with Stacked Borrows) read from or even write to it.

[11] In a concurrent language (which our demo language is not, but Rust is), this assumes that there is no data race. But data races are undefined behavior in Rust, so this assumption is justified.

15.5 Shared references

So far, we have only considered mutable references and raw pointers. We have seen that Stacked Borrows enforces a form of uniqueness for mutable references, enough to justify reordering a memory access around unknown code. Next, we are going to look at shared references. The goal is to enforce that they are read-only, so that we can reorder loads from shared references around unknown code *even if that code has access to the reference* (which, remember, the unknown code in the case of mutable references crucially does *not* have).

Just as we did for mutable references, we arrive at the stack principle for shared references by rephrasing what the borrow checker enforces about them in a way that avoids mentioning lifetimes: *every use of the reference (and everything derived from it) must occur before the next* mutating *use of the referent (after the reference got created), and moreover the reference must not be used for mutation.*

To see how this plays out, let us again consider a simple example involving references to integers:

```
1  let mut local = 42;
2  let x = &mut local;
3  let shared1 = &*x; // Derive two shared references...
4  let shared2 = &*x; // ...from the same mutable reference, x.
5  let val = *x; // Use all *three* references...
6  let val = *shared1; // ...interchangeably...
7  let val = *shared2; // ...for read accesses.
8  *x += 17; // Use x again for a write access.
9  let val = *shared1; // Error! shared1 used after mutating x.
```

Despite the fact that `x`, `shared1` and `shared2` alias, this program is fine until line 8—but in line 9, the stack principle for shared references is violated: a use of the reference `shared1` in line 9 occurs after a mutating use of the referent in line 8.

To model this in Stacked Borrows, we introduce another kind of item that can exist in the borrow stack:

$$\mathit{Item} := \mathsf{Unique}(t) \mid \mathsf{SharedRO}(t) \mid \mathsf{SharedRW}(t) \mid \ldots$$

The new item $\mathsf{SharedRO}(t)$ ("shared read-only") indicates that references tagged with t are allowed to read from but not write to the location associated with this stack. We also equip $\mathsf{SharedRW}$ with a tag. This means we can now speak about the "tag of an item", which will be useful in READ-1. In NEW-MUTABLE-RAW-1, we just push $\mathsf{SharedRW}(\bot)$ instead of $\mathsf{SharedRW}$.[12] That will make the rules for pointer uses easier to state.

We amend the existing Stacked Borrows rules as follows:

Rule (NEW-SHARED-REF-1)**.** Any time a new shared reference is created (`&expr`) from some existing pointer value $\mathsf{Pointer}(\ell, t)$, first of all this is considered a *read access* to that pointer value (so we follow READ-1 below). Then we pick some fresh tag t', use $\mathsf{Pointer}(\ell, t')$ as the value for the shared reference, and add $\mathsf{SharedRO}(t')$ to the top of the stack for ℓ.

Rule (READ-1)**.** Any time a pointer with value $\mathsf{Pointer}(\ell, t)$ is *read from*, an item with tag t (*i.e.*, $\mathsf{Unique}(t)$, $\mathsf{SharedRO}(t)$ or $\mathsf{SharedRW}(t)$) must

[12] For now, all $\mathsf{SharedRW}$ items will have tag $\bot$, but that will change in §16.4.

exist in the stack for ℓ. Pop items off the stack until all the items above the item with tag t are SharedRO(_). If no such item exists in the stack, the program violates the stack principle. (This rule trumps the existing use-2, which only gets used for writes now.)

Notice that we leave the rules for writing unchanged, which means that even if SharedRO(t) *is* in the stack, a reference tagged t cannot be used to write, as that requires a Unique(t) or SharedRW(t). This is the sense in which SharedRO is *read-only*.

The key point in read-1 (and the key difference from use-2) is that after reading with Pointer(ℓ, t), we do *not* necessarily end up with an item with tag t being on top of the stack! There may be some SharedRO above it. This reflects the fact that two shared references can be used for reading without "disturbing" each other; they do not pop the other reference's item off the stack. In contrast, write accesses (which are still governed by use-2) require that the item with the tag of the pointer used for the access becomes the top item on the stack.

Consequently, a key invariant that this system maintains is that if there are any SharedRO's in the stack, they are all adjacent at the top. Notice how all operations that would push another kind of item (creating a mutable reference or a raw pointer) count as a write access, so they would first make some Unique(_) or SharedRW(_) the top of the stack by popping off all SharedRO's above them. *Never is a* Unique(_) *or* SharedRW(_) *pushed on top of a* SharedRO(_).

With these rules, the example program executes as follows:

```
let mut local = 42; // Stored at location ℓ, with tag 0.
let x = &mut local; // = Pointer(ℓ, 1)
// Push tag of x on the stack: h(ℓ) = (42, [Unique(0), Unique(1)]). (new-mutable-ref)
let shared1 = &*x; // = Pointer(ℓ, 2)
let shared2 = &*x; // = Pointer(ℓ, 3)
// New tags are pushed: h(ℓ) = (42, [Unique(0), Unique(1), SharedRO(2), SharedRO(3)]). (new-shared-ref-1)
let val = *x;
// Check: Unique(1) in the stack, all items above are SharedRO(_). (read-1)
let val = *shared1;
// Check: SharedRO(2) in the stack, all items above are SharedRO(_). (read-1)
let val = *shared2;
// Check: SharedRO(3) in the stack, all items above are SharedRO(_). (read-1)
*x += 17;
// Pop until Unique(1) is at the top: h(ℓ) = (59, [Unique(0), Unique(1)]). (use-2)
let val = *shared1;
// Analysis error! The tag of shared1 is not in the stack. (read-1)
```

Observe how the rule for reading references allows `x`, `shared1`, and `shared2` to happily coexist (even the "XYXY" pattern is allowed), but the moment we write to `x`, the SharedRO(_) items are removed from the stack and the corresponding shared references may no longer be used.

15.6 An optimization exploiting read-only shared references

To see how Stacked Borrows' treatment of shared references is helpful, let us consider a function that could benefit from an optimization exploiting that shared references are read-only:

```
1  fn example2(x: &i32, f: impl FnOnce(&i32)) -> i32 {
2    retag x;
3    let val = *x / 3;
4    f(x);
5    return *x / 3; // We want to optimize this to return val.
6  }
```

We use a closure f to reflect the idea that arbitrary code can run in between line 3, where we read x for the first time, and line 5, where we want to optimize away the recomputation of *x / 3. Unlike in the case of mutable references, we even give that unknown code access to our reference x! It is a read-only reference, though, so f should not be able to write through it.

Again we can craft a counterexample that prohibits this optimization under a naive semantics:

```
1  fn main() {
2    let mut local = 6;
3    let x = &local;
4    let result = example2(x, |inner_x| {
5      retag inner_x;
6      let raw_pointer: *mut i32 =
7        unsafe { mem::transmute(inner_x) };
8      unsafe { *raw_pointer = 15; }
9    });
10   println!("{}", result); // Prints "5" (aka 15/3).
11 }
```

The interesting part is in lines 7 and 8, which are the body of the closure f that we pass to example2. There, we circumvent the restriction that one cannot write to a shared reference by calling transmute, which is Rust's unchecked cast operation. This lets us turn the (read-only) shared reference into a (writable) raw pointer—and then we write to it.

However, under Stacked Borrows, this program has undefined behavior (as we would hope). The transmute affects neither the tag t of the pointer nor the borrow stack. Thus, in line 8, when we follow USE-2, we fail to find a $\mathsf{Unique}(t)$ or $\mathsf{SharedRW}(t)$ on the borrow stack. The only item with tag t is the $\mathsf{SharedRO}(t)$ that was added by the **retag** in line 5.[13]

In more detail, here is the step-by-step execution of our counterexample for the optimization of example2 (the closure makes the control flow a bit more complicated—the numbers in [brackets] indicate the order of execution):

```
1  fn main() {
2    let mut local = 6; // Stored at location ℓ, with tag 0.
3    let x = &local; // = Pointer(ℓ, 1)
4    // [0] h(ℓ) = (6, [Unique(0), SharedRO(1)]) (NEW-SHARED-REF-1)
5    // Next: jump to example2 (line 17).
6    let result = example2(x, |inner_x| {
7      retag inner_x;
8      // [3] inner_x = Pointer(ℓ, 3), h(ℓ) = (6, [. . . , SharedRO(1), SharedRO(2), SharedRO(3)])
9      let raw_pointer: *mut i32 = unsafe { mem::transmute(inner_x) };
10     // [4] raw_pointer = Pointer(ℓ, 3), h(ℓ) = (6, [. . . , SharedRO(1), SharedRO(2), SharedRO(3)])
11     unsafe { *raw_pointer = 15; }
12     // [5] Analysis error! No Unique(3) in the stack for write access. (USE-2)
13   });
```

[13] The counterexample would behave the same without the **retag**. We added the **retag** for consistency, but the soundness proof does not expect unknown code to properly use **retag**.

```
14    println!("{}", result); // Prints "5" (aka 15/3).
15  }
16
17  fn example2(x: &i32, f: impl FnOnce(&i32)) -> i32 {
18    retag x;
19    // [1] x = Pointer(ℓ, 2), h(ℓ) = (6, [Unique(0), SharedRO(1), SharedRO(2)])
20    let val = *x / 3;
21    // [2] Check: SharedRO(2) is in the stack, all items above are SharedRO(_). (read-1)
22    f(x); // Next: jump to closure body (line 6).
23    return *x / 3; // We want to optimize this to return val.
24  }
```

15.7 A proof sketch for the optimization on shared references

We have ruled out one particular counterexample to the desired optimization, but what we really need to do is to argue that the optimization is correct in any possible context. Here is the relevant code again:

```
1  fn example2(x: &i32, f: impl FnOnce(&i32)) -> i32 {
2    retag x;
3    let val = *x / 3;
4    f(x);
5    return *x / 3; // We want to optimize this to return val.
6  }
```

The argument now goes as follows:

1. Assume that after the `retag` in line 2, `x` has value Pointer(ℓ, t). We know that SharedRO(t) is at the top of the borrow stack for ℓ.[14] Let us call the current scalar value stored in that location s. The goal is to show that in line 5, ℓ *still* stores the value s.
2. While `f` executes, we know that *any* write access will pop off *all* SharedRO's from the stack. This relies on the invariant that all SharedRO's in the stack are sitting on top. So we conclude that as long as SharedRO(t) is in the stack, the value stored at ℓ remains s.
3. Finally, in line 5, we may assume that SharedRO(t) is still in the stack, because otherwise the program would violate Stacked Borrows (there exists no other item with that tag). As a consequence, ℓ still stores s, which justifies the optimization.

Reordering. We can write the above optimization as a reordering as follows:

```
1  retag x;                        1  retag x;
2  let val = *x / 3;               2  let val = *x / 3;
3  f();                        ⇒   3  let retval = *x / 3;
4  let retval = *x / 3;            4  f();
5  return retval;                  5  return retval;
```

The only change is that we swapped lines 3 and 4, *i.e.*, we moved a load from a shared reference up across some unknown code *with access to that variable*. Now it is easy to justify that doing `*x / 3` twice in a row will produce the same result.[15]

This is another example of a transformation that a C/C++ compiler cannot even hope to perform.

[14] Without the `retag`, the borrow stack could contain SharedRW(t) or Unique(t) instead of SharedRO(t), and the proof would not go through.

[15] Again this relies on there being no data races in a well-defined Rust program.

Chapter 16
Protectors and interior mutability

So far, we have seen two optimizations that exploit the assumption that even unsafe code has to conform to Stacked Borrows, a dynamic analysis that mirrors the static analysis performed by the Rust borrow checker. In this chapter, we will extend Stacked Borrows on two axes: in §16.1–§16.3 we will show how to support transformations that reorder instructions *down* across unknown code (giving the compiler more freedom in its optimizations), and in §16.4 we will show how to support *interior mutability* (§8.6) in Stacked Borrows.

16.1 Reordering memory accesses down instead of up

Both of the optimizations we discussed so far followed a similar pattern: some reference got retagged (step (1) in the proof sketches), then we made some change / observation (we wrote to the mutable reference / read the shared reference), then some unknown code got executed (2), and finally we used our original reference again (3). The retag makes our reference the "topmost" one in the stack, and the final use asserts that it is still there. This works well for reordering instructions *up* across unknown code.

However, in some cases it is also interesting to be able to move a memory access *down* across unknown code.[1] For example, consider the following function:

```
1  // Moving a read down across f().
2  fn example2_down(x: &i32, f: impl FnOnce(&i32)) -> i32 {
3    retag x;
4    let val = *x;
5    f(x);
6    return val; // Can return *x instead.
7  }
```

We might want to move the read of `x` down from line 4 to line 6, across the call to `f`. In the context of some more complex code, this could reduce register pressure because we do not need to remember `val` around the call to `f`.

For mutable references, we can similarly move reads down across code that does not need them, but the more interesting case is moving a *write* access down:

[1] This is harder, because it *extends* the liveness range of a reference.

```
1  // Moving a write down across f().
2  fn example3_down(x: &mut i32, f: impl FnOnce()) {
3    retag x;
4    *x = 42; // This write is redundant.
5    f();
6    *x = 13;
7  }
```

Removing the redundant write to `x` here boils down to moving it *down* from line 4 to line 6. Then we end up with two adjacent writes to the same reference, and the first one can be removed. However, this means that unlike in the optimizations we considered so far—in fact, unlike in the vast majority of optimizations that are typically performed—`f` will be called with a different value stored in `x` than in the original program! We will see in §16.3 how this can work.

16.2 Protectors

It turns out that Stacked Borrows as we have presented it so far does not permit these optimizations that move operations down. Here is a counterexample for `example2_down`:

```
 1  fn main() {
 2    let mut local = 42; // Stored at location ℓ, with tag 0.
 3    let raw_pointer = &mut local as *mut i32;
 4    let val = example2_down(
 5      unsafe { &*raw_pointer }, // = Pointer(ℓ, 2)
 6      |x_inner| unsafe {
 7        retag x_inner; *raw_pointer = 17; // Changes *x.
 8      },
 9    );
10    println!("{}", val); // Prints 42.
11  }
```

The closure that we pass as `f` changes the value stored in `x` by writing to `raw_pointer`, an alias of `x`. This is allowed by Stacked Borrows. Immediately before the write, the borrow stack for ℓ looks as follows:

$$[\mathsf{Unique}(0), \mathsf{Unique}(1), \mathsf{SharedRW}(\bot), \mathsf{SharedRO}(2), \mathsf{SharedRO}(3)]$$

Here, 1 is the tag of the temporary mutable reference created in line 3, 2 is the tag of the shared reference created in line 5, and 3 is the tag of `x` inside `example2_down` (retagging assigns a new tag there). In this situation, we are allowed to perform a write with a raw pointer thanks to the $\mathsf{SharedRW}(\bot)$ in the stack, and the resulting stack is $[\mathsf{Unique}(0), \mathsf{Unique}(1), \mathsf{SharedRW}(\bot)]$. This removes the tag of `x` from the stack, but since `x` does not get used again, that is not a violation of the rules we have set so far.[2]

However, in Rust, a reference passed to a function must *outlive* the function call—*i.e.*, its lifetime must last at least for the duration of the call. Intuitively, since the lifetime of a reference relates to how long the item for that reference's tag is in the stack, the problem in the above counterexample is that the lifetime of `x` ends while `example2_down` is still running, when $\mathsf{SharedRO}(3)$ gets popped off the stack. To enable

[2] This is why extending the liveness range of a reference is hard: when a reference is used somewhere, that lets the compiler make certain assumptions. When adding a new use, there are no use-based assumptions the compiler can make.

the desired optimization, we thus want to treat the counterexample as incurring undefined behavior, by reflecting Rust's "outlives" rule into Stacked Borrows: $\mathsf{SharedRO}(3)$ must not be popped until the execution of `example2_down` is finished.

To prevent $\mathsf{SharedRO}(3)$ from being popped off while `example2_down` still runs, we introduce the notion of a *protector*: an item in the stack can be *protected by a function call*, which means that while that function call is still ongoing, if the item is popped off the stack, that is a Stacked Borrows violation and hence undefined behavior.[3]

Formally, we extend the items in the borrow stack with an optional *call ID*:

$$\mathit{CallId} := \mathbb{N} \qquad \mathit{Item} := \mathsf{Unique}(t, c) \mid \mathsf{SharedRO}(t, c) \mid \mathsf{SharedRW}(t, c)$$

Here, c is an element of $\mathit{CallId}^? := \mathit{CallId} \cup \{\bot\}$. Every call ID represents a function call (we can imagine it being associated with the stack frame). We also assume that the semantics keeps track of the set of call IDs which correspond to function calls that have not returned yet. We use $\mathsf{Unique}(t)$ as notation for $\mathsf{Unique}(t, \bot)$, and similarly for the other kinds of items.

So, when do newly pushed items get protectors? Remember that we have **`retag`** operations every time a reference is passed in as an argument, read via a pointer, or returned from another function. The idea is that we make the first case, the **`retag`** of the arguments, special in the sense that this retag will protect any new items that it adds with the call ID of the current function call. This will exactly capture the references passed to a function, *i.e.*, the ones that we know have to outlive the function call. The protector ensures that the corresponding items will not be popped during said function call. Syntactically, we will write `retag[fn] x` to indicate that `x` is being retagged in the prologue of a function, and thus its items get protected.

Now we add the following to our rules for Stacked Borrows:

Rule (retag-fn)**.** When pushing new items to a location's borrow stack as part of a `retag[fn]` (in new-mutable-ref or new-shared-ref-1), these items have their protector set to the call ID of the current function call.

Rule (protector)**.** Any time an item is popped (by use-2 or read-1), check if it has a protector ($c \neq \bot$). If it does, and if that call ID corresponds to a function call that is still ongoing (that is, the corresponding stack frame is still in the call stack), we say that the protector is "active", and popping an active protector is undefined behavior.

The code of `example2_down` changes a bit to reflect that we need the new kind of retag for `x`:

```
// Moving a read down across f().
fn example2_down(x: &i32, f: impl FnOnce(&i32)) -> i32 {
  retag[fn] x;
  let val = *x;
  f(x);
  return val; // Can return *x instead.
}
```

[3] This gives functions a special status that other code blocks do not enjoy; that status is lost when a function is inlined. We consider this reasonable because within a function, lifetimes in Rust are subject to inference (and inference can and will change as Rust evolves), but the rule that the lifetime of a reference passed to a function outlives the function call is set in stone. In the future, we would like to explore a more explicit representation that enables this information to be preserved during inlining.

If we now consider the previous counterexample again, this time the borrow stack of ℓ at the beginning of the execution of the closure is:

$$[\textsf{Unique}(0), \textsf{Unique}(1), \textsf{SharedRW}(\bot), \textsf{SharedRO}(2), \textsf{SharedRO}(3, c)]$$

Here, c is the ID of the call to `example2_down`. But this means when the closure executes and performs its raw pointer write, it will hit PROTECTOR: a raw pointer write has to make $\textsf{SharedRW}(\bot)$ the top of the stack, which requires popping off $\textsf{SharedRO}(3, c)$, but that is not allowed because the protector c is still active.

16.3 Proof sketches for the optimizations

We have seen that protectors successfully rule out what used to be a counterexample against optimizing `example2_down`. In fact, protectors enable us to validate the transformation:

1. Assume that after the `retag[fn]` in line 3, `x` has value $\textsf{Pointer}(\ell, t)$. We know that the top item of the borrow stack for location ℓ is $\textsf{SharedRO}(t, c)$. Here, c is the call ID of `example2_down`. We remember the current scalar value stored at ℓ as s.
2. While `f` executes, we know that any write access to ℓ would pop all $\textsf{SharedRO}(\_, \_)$ off the stack. That would immediately be a Stacked Borrows violation because $\textsf{SharedRO}(t, c)$ has an active protector c. Thus no write access can happen.
3. As a consequence, when `f` returns (if it ever returns), ℓ still stores s. This justifies the transformation.

Almost the same argument works for `example3_down`, which moves a *write* down across a function call. Again, `x`'s tag is at the top of the borrow stack when `f` gets called, and it is protected. Any attempt by `f` to access `x` (reading or writing) will pop off that item, which is not allowed due to the protector. Thus `f` cannot observe the value stored in `x`, and we are free to do the write later.[4]

What is remarkable about this transformation is that the compiler ends up calling `f` with *a different value stored in* `x`, but we can show that this cannot affect the behavior of programs complying with Stacked Borrows: any access to `x` by `f` would immediately be undefined behavior. This applies even if `f` runs into an infinite loop and we never reach step (3).

Completing the picture. To complete the matrix of moving both read and write accesses to both shared and mutable references both up and down across unknown code, we have also verified the following transformations:

```
// Moving a read down across f().
fn example1_down(x: &mut i32, f: impl FnOnce()) {
  retag[fn] x;
  let val = *x;
  f();
  return val; // Can return *x instead.
}
```

[4] We assume that `f` cannot return in other ways, such as through an exception or via unwinding. Unlike our formal model, Rust supports unwinding. The same transformation is still possible, but the write has to be pushed down into both the normal continuation and the unwinding continuation.

```
// Moving a write up across f().
fn example3(x: &mut i32, f: impl FnOnce()) {
  retag[fn] x;
  *x = 42; // We can change this to write 13...
  f();
  *x = 13; // ...and remove this write.
}
```

The correctness arguments proceed just like above. Unlike the transformations that move down *read* accesses, `example3` requires a `retag[fn]` because when `f` is called, the memory in the transformed and the original program differs: if `f` observed that difference and then entered an infinite loop, that would constitute a counterexample to the optimization.[5] So we need protectors to make it impossible for `f` to observe the value stored behind `x`.

This shows that reordering in both directions is possible for all reference accesses.

16.4 Interior mutability

We have seen that Stacked Borrows supports all of the transformations that it was designed to enable: moving uses of shared and mutable references up and down across unknown code. This gives the compiler the necessary freedom to exploit reference types for its alias analysis. But to have Stacked Borrows adopted, we additionally need to make sure that the large body of existing Rust code is actually compatible with the rules of Stacked Borrows. Unfortunately, there is one feature of Rust that is in direct contradiction to our treatment of shared references: *interior mutability*. As explained in §8.6, and contrary to the exclusion principle, Rust in fact *does* permit mutation of shared references under some controlled circumstances. For example, Rust provides a type called `Cell` that permits mutation through a `set` method that works with a shared reference:

```
fn cells(x: &Cell<i32>, y: &Cell<i32>) -> i32 {
  x.set(13);
  y.set(42);
  return x.get(); // Can return 13 or 42.
}
```

The return value of this function depends on whether `x` and `y` alias or not. A `&Cell<i32>` is essentially a safe version of an `int*` in C: it can be read from and written to by many parties at any time.[6] In §8.6, we discussed why this does not break type safety.

Of course, allowing mutation of aliased data makes alias analysis as hard as it is in C. However, the Rust designers were aware that interior mutability would wreak havoc with the otherwise very strong aliasing guarantees that their type system provides, so they decided early on that the programmer must "mark" data that will be subject to interior mutability in a way that can be recognized by the compiler. To this end, the standard library contains the "magic" type `UnsafeCell<T>`. An `UnsafeCell<T>` is basically the same as a `T`; however, the type is specially recognized by the compiler. This is exploited in the rule for mutation of

[5] In contrast, if the same happens with a moved-down read access, the optimization is still correct. If `f` never returns, it does not matter what it does with `x`.

[6] In fact, after inlining, the function `cells` above looks just like C code writing to and reading from `int*`.

shared data, which says that shared references may not be used to perform mutation *except* when the mutation happens "inside" an `UnsafeCell`. For example, `Cell<T>` is a newtype built around `UnsafeCell<T>` with a safe API.

We exploit this information in Stacked Borrows and exempt the memory inside an `UnsafeCell<T>` from the usual read-only restriction for shared references. To track references into memory inside an `UnsafeCell<T>`, we make use of the $\mathsf{SharedRW}$ items that we have already seen for raw pointers: raw pointers and interior mutable shared references both permit mutable aliasing, so it makes sense to treat them similarly.

"Partial" interior mutability. What makes this complicated is that, in general, a shared reference can have "partial" interior mutability. So far, we have pretended that a reference points to a single location and affects a single stack, but of course in reality a reference like `&i32` spans 4 locations. All the rules that we have seen (both for creating references/raw pointers and for extra actions to be performed on memory accesses) apply to the stack of *every location* that is covered by the reference (with the size determined by the type `T`). However, with `&(i32, Cell<i32>)`, the first 4 bytes are *not* inside an `UnsafeCell` and are thus subject to the full set of Stacked Borrows rules for shared references, but the second 4 bytes *are* inside an `UnsafeCell`, and Stacked Borrows will permit mutation of shared data for those 4 bytes only.

Consequently, when creating a shared reference with "partial" interior mutability, we *cannot* perform the same actions for all locations that the reference covers. We first pick a new tag t, and then we must find the `UnsafeCell`s in the memory region that the reference points to. This is basically a type-based traversal of the memory the reference covers.[7] For all locations outside of the `UnsafeCell`s, we proceed as before and push a new *read-only* $\mathsf{SharedRO}(t, c)$ on top of the stack, after following READ-1 (remember that creating a shared reference counts as a read access). For all locations *inside* an `UnsafeCell`, we instead add to the stack the *read-write* item $\mathsf{SharedRW}(t, c)$, permitting the same aliasable mutation that mutable raw pointers enjoy.[8]

Creating a reference is not always an access. The other difference from what we have seen so far is that, for the part inside the `UnsafeCell<T>`, creating a shared reference does *not* count as a read access, and the new item does *not* get added at the top of the stack, but instead in the middle.

To explain why we cannot treat creating a shared reference as a read access when interior mutability is involved, we have to briefly discuss `RefCell<T>`. `RefCell<T>` is a Rust standard library type that is somewhat similar to `Cell<T>`, but it *does* allow interior pointers.[9] It achieves safety by tracking at run-time how many mutable and shared references to the data exist, and making sure that there always is either exactly one mutable reference or an arbitrary number of (read-only) shared references. However, this means that it is possible to call the following function from safe code *in a way that the two references `shared` and `mutable` alias*:

[7] Rust supports polymorphism but performs monomorphization during compilation, so in the dynamic semantics, we only have to deal with concrete types.

[8] For simplicity, this type-based traversal does *not* inspect `enum`s, which are sum types in Rust. If any variant of the `enum` contains an `UnsafeCell`, we consider the entire `enum` as being subject to interior mutability. Determining the active variant of an `enum` requires accessing memory, so suddenly the Stacked Borrows rules themselves would be subject to the same restrictions as all other memory accesses—this is a complexity that we wanted to avoid. The loss in optimization potential is limited as compilers could only optimize accesses to such `enum`s if they can predict which variant they will be in.

[9] Remember that `Cell<T>` cannot allow interior pointers to ensure that mutating the `Cell<T>` does not invalidate possible aliases. `RefCell<T>` uses dynamic tracking to ensure that no interior pointers exist when the data is mutated.

```
1  fn nasty_ref_cell(shared: &RefCell<i32>, mutable: &mut i32) {
2    retag[fn] shared; retag[fn] mutable;
3    let more_shared = &*shared;
4    *mutable = 23;
5  }
```

We can call this function with a shared reference to some `RefCell` and a mutable reference *to its contents*, in which case the memory ranges they point to overlap.

If creating a shared reference (with the reborrow in line 3, and in fact already for the `retag` in line 2) counted as a read access, that would violate uniqueness of `mutable`! After all, one important property of mutable references is that there are no other accesses (read or write) to that memory while the reference is used. We also cannot add a new item for `more_shared` on top of the stack; that would violate the invariant that the tag of `mutable` is at the top of all the stacks it points to (after retagging)—an invariant which was crucial in our proof sketches. However, we have to accept `nasty_ref_cell` and its caller, because they consist solely of safe code. Rejecting this example would mean declaring `RefCell<T>` unsound, which is not an option.

Consequently, we need to create a shared reference without popping things off the stack. However, we still want the item to be "next to" the one it is derived from. So, when creating a new shared reference from Pointer(ℓ, t), we add the new SharedRW just above where t is in the stack.

Rule (NEW-SHARED-REF-2)**.** When creating a new shared reference from some existing pointer value Pointer(ℓ, t), we pick some fresh tag t' and use Pointer(ℓ, t') as the value for the shared reference.

For all locations covered by this reference that are inside an `UnsafeCell`, we find the item for t in the borrow stack, and we add SharedRW(t') just above that item. (If there is no such item, the program has undefined behavior.) For the remaining locations, this counts as a read access with the old tag t (see READ-1). Then we push SharedRO(t') to the top of the stack.

To keep the rules for shared references with interior mutability and mutable raw pointers consistent (both use SharedRW, after all), we adjust the latter to not count as an access either:

Rule (NEW-MUTABLE-RAW-2)**.** When a mutable raw pointer is created via a cast (`expr as *mut T`) from some mutable reference (`&mut T`) with value Pointer(ℓ, t), we find the item for t in the borrow stack, and we add SharedRW$(\bot)$ just above that item. (If there is no such item, the program has undefined behavior.) Then the new pointer has value Pointer$(\ell, \bot)$.

With these rules, reborrowing `shared` will just add some SharedRW in the middle of the stack—so even if `mutable` aliases, we maintain the property that the *top item* of the stack is a Unique with the same tag as `mutable`. That is enough to keep our proofs working, and it also permits a program calling `nasty_ref_cell` with aliasing pointers.

This means, however, that we can end up with many SharedRW next to each other in the stack, *e.g.*, when creating a bunch of shared references with interior mutability from the same mutable reference. Just as we considered the adjacent SharedRO on top of the stack as one "group" of items that can all be used without removing each other from the stack, we want to do the same with such a "group" of adjacent SharedRW items. Thus we need to adjust the rules for (read and write) accesses in such a way that if any of an adjacent group of SharedRW is used, the others in that group all remain in the stack.

For writes, this looks as follows (also incorporating PROTECTOR):

Rule (WRITE-1)**.** On any write access using pointer value $\mathsf{Pointer}(\_, t)$, do the following for each location ℓ affected by the access: pop the stack until either the top item is $\mathsf{Unique}(t, \_)$, or a $\mathsf{SharedRW}(t, \_)$ exists in the top "group" of adjacent SharedRW in the stack (*i.e.*, $\mathsf{SharedRW}(t, \_)$ is in the stack, and there are only SharedRW above it). If this is not possible, or if this pops off an item with an active protector, the program has undefined behavior.

Read accesses and disabled items. Read accesses, too, need to account for the fact that we now have *tagged* SharedRW items. We want to be able to read from a SharedRW item without invalidating the other SharedRW or SharedRO items that may be adjacent to it.[10]

However, we still need to make sure that there are never any Unique items left above the item that justifies the access, and there is a slight twist to how we do this. For reasons we will explain shortly, instead of popping items off the stack, we keep all the items where they are but mark the Unique ones as "disabled". This means in particular that reading from a SharedRW will maintain validity of *all* SharedRW items above it on the stack (not just the ones that are adjacent to it).

To this end, we introduce a new kind of item:

$$\mathit{Item} := \mathsf{Unique}(t, c) \mid \mathsf{SharedRO}(t, c) \mid \mathsf{SharedRW}(t, c) \mid \mathsf{Disabled}$$

And we change the read rule as follows:

Rule (READ-2)**.** On any read access using pointer value $\mathsf{Pointer}(\_, t)$, do the following for each location ℓ affected by the access: find the topmost (non-Disabled) item with tag t in the stack (there can be several if $t = \bot$). Replace all the $\mathsf{Unique}(\_, \_)$ above it by Disabled. If any of these $\mathsf{Unique}(\_, \_)$ has an active protector, the program has undefined behavior.

The reason for this quirk in READ-2 is that testing with some real Rust code quickly revealed the following pattern:

```
fn make_raw(y: &mut i32) -> *mut i32 { retag[fn] y; y as *mut i32 }

fn bad_pattern(x: &mut i32) {
  retag[fn] x; // Assume x = Pointer(ℓ, t_x).
  let raw_ptr = make_raw(x); // Desugars to `make_raw(&mut *x)`.
  // h(ℓ) = [..., Unique(t_x), Unique(t_tmp), Unique(t_y), SharedRW(⊥)]
  let val1 = *x;
  // h(ℓ) = [..., Unique(t_x)]
  let val2 = unsafe { *raw_ptr }; // Fails because SharedRW(⊥) is no longer in the stack!
}
```

[10] This parallels what we already did for reading SharedRO items in READ-1 and writing SharedRW items in WRITE-1.

This pattern is widely used (and thus should *not* induce undefined behavior), but it *would* induce undefined behavior without the “disabled” approach. Right before line 7, the borrow stack for ℓ contains items for the intermediate mutable references that were created “between” x and raw_ptr: the temporary variable that is introduced because passing a reference to a function in Rust is implicitly reborrowing, and the new tag that gets picked when y is retagged in make_raw.

The trouble is that reading from x has to invalidate the Unique items above it: the example3_down transformation to move a write down across a function call relies on the fact that other code cannot read from memory covered by a mutable reference without causing a violation. But if we are forced to adhere to the stack discipline, we can only invalidate the Unique by first popping off the SharedRW($\bot$), which makes line 9 illegal because SharedRW($\bot$) is no longer in the stack.

To fix this[11], we deviate from the stack discipline in the way we handle removal of tags on a read access. Namely, instead of popping items off the stack until there are no more Unique above the accessed item (which would pop off some SharedRW as well), we just *disable* the Unique that are above the accessed item while leaving the SharedRW items alone. That way, the SharedRW items do not suffer any “collateral damage” from the popping of Unique items, and the above programming pattern is permitted.

This is the last adjustment we need. Let us now take a step back and consider the semantics in its entirety—formally.

[11] Note that in this example we could replace the dereference *raw_ptr in line 9 with *x because make_raw is the identity function. However, in general make_raw will actually do some work that we do not want to re-do.

Chapter 17
Formal operational semantics

The purpose of this chapter is to formally define what we have so far described informally in the preceding two chapters.

17.1 High-level structure

At a high level, we are defining Stacked Borrows as a labeled transition system where the labels are *events* (the most important ones being read and write accesses and retagging) and the state is described by the following record (some of the remaining relevant domains are shown in Figure 17.1):

$$\begin{aligned}
\varsigma \in \mathit{SState} &:= \left\{ \begin{array}{r@{}l} \textsc{stacks} : {} & \mathit{Stacks}, \\ \textsc{calls} : {} & \mathit{List}(\mathit{CallId}), \\ \textsc{nextptr} : {} & \mathit{PtrId}, \\ \textsc{nextcall} : {} & \mathit{CallId} \end{array} \right\} \\
\xi \in \mathit{Stacks} &:= \mathit{Loc} \xrightharpoonup{\mathsf{fin}} \mathit{Stack}
\end{aligned}$$

Here, STACKS tracks the stack for each location, CALLS tracks the current list of active call IDs (needed to implement PROTECTOR), and NEXTPTR and NEXTCALL are used to generate fresh pointer IDs and call IDs.

This approach decouples Stacked Borrows from the rest of the language: the operational semantics of the language can just include an *SState* and take appropriate transitions on that state whenever a relevant event occurs. Therefore, unlike in the previous chapters, we do not use a single heap $h \in \mathit{Mem}$ for both the value and stack at each location. Only the stack is relevant for Stacked Borrows, and it is tracked by *SState* in its STACKS field; the actual values stored in the heap are assumed to be handled by the rest of the semantics. In Figure 17.1, we also re-define *Item* as a triple so that the common structure (a permission, a tag, and an optional protector) is reflected in the structure of the data.

The possible events and corresponding transitions are likewise defined in Figure 17.1. These events include all memory accesses (reads, writes, allocations, and deallocations), as well as initiating and ending a function call[1] and, of course, retagging. Most of the parameters in these events are "input parameters", in the sense that they represent information that comes from the outside world into the Stacked Borrows subsystem. Only the call ID c in the call events and the new tag t_{new} in retag events are

[1] This is relevant for the tracking of call IDs to make protectors work.

Domains.

$$
\begin{aligned}
\mathit{PtrId} &:= \mathbb{N} & \iota \in \mathit{Item} &:= \mathit{Permission} \times \mathit{Tag} \times \mathit{CallId}^{?} \\
t \in \mathit{Tag} &:= \mathit{PtrId}^{?} & p \in \mathit{Permission} &:= \mathsf{Unique} \mid \mathsf{SharedRW} \mid \mathsf{SharedRO} \mid \mathsf{Disabled} \\
c \in \mathit{CallId} &:= \mathbb{N} & S \in \mathit{Stack} &:= \mathit{List}(\mathit{Item})
\end{aligned}
$$

Events.

$$
\begin{aligned}
\mathit{AccessType} &:= \mathsf{AccessRead} \mid \mathsf{AccessWrite} \qquad m \in \mathit{Mutability} := \mathsf{Mutable} \mid \mathsf{Immutable} \\
\mathit{RetagKind} &:= \mathsf{Default} \mid \mathsf{Raw} \mid \mathsf{FnEntry} \qquad \mathit{PtrKind} := \mathsf{Ref}(m) \mid \mathsf{Raw}(m) \mid \mathsf{Box} \\
\tau \in \mathit{Type} &:= \mathsf{FixedSize}(n) \mid \mathsf{Ptr}(k, \tau) \qquad \text{where } n \in \mathbb{N},\ k \in \mathit{PtrKind} \\
&\mid \mathsf{UnsafeCell}(\tau) \mid \mathsf{Union}(\tau^*) \mid \mathsf{Prod}(\tau^*) \mid \mathsf{Sum}(\tau^*) \\
\varepsilon \in \mathit{Event} &:= \mathsf{EAccess}(a, \mathsf{Pointer}(\ell, t), \tau) \qquad \text{where } a \in \mathit{AccessType},\ \tau \in \mathit{Type} \\
&\mid \mathsf{ERetag}(\mathsf{Pointer}(\ell, t_{old}), t_{new}, \tau, k, k') \qquad \text{where } k \in \mathit{PtrKind},\ k' \in \mathit{RetagKind} \\
&\mid \mathsf{EAlloc}(\mathsf{Pointer}(\ell, t), \tau) \mid \mathsf{EDealloc}(\mathsf{Pointer}(\ell, t), \tau) \\
&\mid \mathsf{EInitCall}(c) \mid \mathsf{EEndCall}(c) \qquad \text{where } c \in \mathit{CallId}
\end{aligned}
$$

Transitions.

$$
\frac{\begin{array}{c}\textsc{OS-access} \\ \mathsf{MemAccessed}(\varsigma.\textsc{stacks}, \varsigma.\textsc{calls}, a, \mathsf{Pointer}(\ell, t), |\tau|) = \xi' \qquad \varsigma' = \varsigma \ \textbf{with}\ [\textsc{stacks} := \xi']\end{array}}{\varsigma \xrightarrow{\mathsf{EAccess}(a, \mathsf{Pointer}(\ell,t), \tau)} \varsigma'}
$$

$$
\frac{\begin{array}{c}\textsc{OS-retag} \\ \mathsf{Retag}(\varsigma.\textsc{stacks}, \varsigma.\textsc{nextptr}, \varsigma.\textsc{calls}, \mathsf{Pointer}(\ell, t_{old}), \tau, k, k') = (t_{new}, \xi', n') \\ \varsigma' = \varsigma \ \textbf{with}\ [\textsc{stacks} := \xi', \textsc{nextptr} := n']\end{array}}{\varsigma \xrightarrow{\mathsf{ERetag}(\mathsf{Pointer}(\ell,t_{old}), t_{new}, \tau, k, k')} \varsigma'}
$$

$$
\frac{\begin{array}{c}\textsc{OS-alloc} \\ \forall \ell' \in [\ell, \ell + |\tau|).\ \ell' \notin \mathsf{dom}(\varsigma.\textsc{stacks}) \qquad t = \varsigma.\textsc{nextptr} \\ \xi' = \varsigma.\textsc{stacks}\ \textbf{with}_{\ \ell' \in [\ell, \ell+|\tau|)} [\ell' := [(\mathsf{Unique}, t, \bot)]] \\ \varsigma' = \varsigma \ \textbf{with}\ [\textsc{stacks} := \xi', \textsc{nextptr} := \varsigma.\textsc{nextptr} + 1]\end{array}}{\varsigma \xrightarrow{\mathsf{EAlloc}(\mathsf{Pointer}(\ell,t), \tau)} \varsigma'}
$$

$$
\frac{\begin{array}{c}\textsc{OS-dealloc} \\ \mathsf{MemDeallocated}(\varsigma.\textsc{stacks}, \varsigma.\textsc{calls}, \mathsf{Pointer}(\ell, t), |\tau|) = \xi' \qquad \varsigma' = \varsigma \ \textbf{with}\ [\textsc{stacks} := \xi']\end{array}}{\varsigma \xrightarrow{\mathsf{EDealloc}(\mathsf{Pointer}(\ell,t), \tau)} \varsigma'}
$$

$$
\frac{\begin{array}{c}\textsc{OS-init-call} \\ c = \varsigma.\textsc{nextcall} \qquad \varsigma' = \varsigma \ \textbf{with}\ [\textsc{calls} := \varsigma.\textsc{calls} \mathbin{+\!\!+} [c], \textsc{nextcall} := c + 1]\end{array}}{\varsigma \xrightarrow{\mathsf{EInitCall}(c)} \varsigma'}
$$

$$
\frac{\begin{array}{c}\textsc{OS-end-call} \\ \varsigma.\textsc{calls} = C \mathbin{+\!\!+} [c] \qquad \varsigma' = \varsigma \ \textbf{with}\ [\textsc{calls} := C]\end{array}}{\varsigma \xrightarrow{\mathsf{EEndCall}(c)} \varsigma'}
$$

Figure 17.1: Stacked Borrows domains and transitions.

"output parameters", representing information that is returned by Stacked Borrows to the outside world.[2]

The effect of every event on the Stacked Borrows state ς is described by a function that *computes* the next state given the previous one. We use ς **with** [FIELD := e] as notation for updating a single field of a record (and, as we will see later, also for updating elements of a finite partial map). Considering the very algorithmic nature of Stacked Borrows, we feel that it lends itself more to this computational style than a relational one.

The two rules OS-INIT-CALL and OS-END-CALL implement the tracking of active calls IDs for protector enforcement: OS-INIT-CALL picks a fresh call ID c and adds it to the call stack CALLS, and OS-END-CALL ensures that c is the topmost call ID in CALLS and removes it from the call stack.[3]

The rules that actually interact with memory are somewhat more complicated. In the following, we first discuss accesses (reads and writes) and (de)allocation in §17.2, and then retagging in §17.3.

17.2 Memory accesses

In Figure 17.2, we formalize READ-2 and WRITE-1 by implementing MemAccessed, the function defining what happens on EAccess events. The definition of the function itself is simple: it iterates over all locations affected by the access and calls the helper function Access to compute their new stacks. To determine the set of affected locations, we need to know the size of the access; this is computed based on the type τ which is part of the event.[4] Access is a partial function: $\perp$ is used as return value to express that the access is illegal. If that is the case, our **with** operator propagates the failure, so MemAccessed also returns $\perp$ and the transition system (Figure 17.1) gets stuck.

For both reads and writes, Access (Figure 17.2) starts by finding the "granting item", defined by the Grants function. Grants(p, a) determines if an item with given permission p may be used to justify a memory access, with a indicating whether this is a read or a write access. For a write, only Unique and SharedRW are allowed (as in WRITE-1); for a read we also accept SharedRO (excluding only Disabled, as in READ-2). Access uses FindGranting to search the stack S top-to-bottom to find the topmost item which matches the given tag t and grants the current access. Implicitly, in the base case of the recursion where the list is empty, FindGranting returns $\perp$ (as none of its patterns match). This only happens when there is no item that grants the desired access.

In Access, we use **bind** to denote the bind operation of the partiality monad: if FindGranting returns $\perp$, then that is propagated by Access; otherwise we proceed with the remaining computation. Having determined the granting item, Access implements the rest of READ-2 and WRITE-1, respectively, as indicated by the comments in the definition.

Allocation and deallocation. Allocation, as defined in OS-DEALLOC in Figure 17.1, picks a fresh tag t and assigns that tag Unique permission

[2] In other words, for the input parameters, Stacked Borrows is *receptive* except when the particular parameters chosen cause undefined behavior. For output parameters, Stacked Borrows expects the other transition system it is coupled with to be receptive.

[3] For the formal proofs, in particular when handling calls to unknown code, it turns out to be beneficial to enforce a stack discipline on call IDs, and not just track the *set* of active call IDs.

[4] Types reflect just as much of the Rust type system as is needed for Stacked Borrows. This is a very coarse-grained abstraction of the types of λ_{Rust} that we introduced in §9.

```
(* Defines whether $p$ can be used to justify accesses of type $a$.  *)
$\mathsf{Grants}(p : \mathit{Permission}, a : \mathit{AccessType}) : \mathbb{B}$
  $\mathsf{Grants}(\mathsf{Disabled}, \_)$                 $:=$  $\textbf{false}$
  $\mathsf{Grants}(\mathsf{SharedRO}, \mathsf{AccessWrite})$  $:=$  $\textbf{false}$
  $\mathsf{Grants}(\_, \_)$                        $:=$  $\textbf{true}$

(* Finds the topmost item in $S$ that grants access $a$ to $t$.
   Returns the index in the stack ($0$ = bottom) and the permission of the granting item.  *)
$\mathsf{FindGranting}(S : \mathit{Stack}, a : \mathit{AccessType}, t : \mathit{Tag}^?) : (\mathbb{N} \times \mathit{Permission})^?$
  $\mathsf{FindGranting}(S \mathbin{+\!\!+} [\iota], a, t)$  $:=$  $\textbf{if}\ (\iota.\textsc{tag} = t \land \mathsf{Grants}(\iota.\textsc{perm}, a))$
                                    $\textbf{then}\ (|S|, \iota.\textsc{perm})\ \textbf{else}\ \mathsf{FindGranting}(S, a, t)$

(* Finds the bottommost item above $i$ that is incompatible with a write justified by $p$.  *)
$\mathsf{FindFirstWIncompat}(S : \mathit{Stack}, i : \mathbb{N}, p : \mathit{Permission}) : \mathbb{N}^?$
  (* Writes to $\mathsf{Unique}$ are incompatible with everything above.  *)
  $\mathsf{FindFirstWIncompat}(S, i, \mathsf{Unique})$     $:=$  $i + 1$
  (* Writes to $\mathsf{SharedRW}$ are compatible with adjacent $\mathsf{SharedRW}$, and nothing else.
     So if the next item up is $\mathsf{SharedRW}$ then go on searching, otherwise stop.  *)
  $\mathsf{FindFirstWIncompat}(S, i, \mathsf{SharedRW})$  $:=$  $\textbf{if}\ i + 1 < |S| \ \land\ S(i+1).\textsc{perm} = \mathsf{SharedRW}$
                                    $\textbf{then}\ \mathsf{FindFirstWIncompat}(S, i+1, \mathsf{SharedRW})\ \textbf{else}\ i + 1$

(* Computes the new stack after an access of type $a$ with tag $t$.
   Also depends on the active calls tracked by $C$.  *)
$\mathsf{Access}(a : \mathit{AccessType}, S : \mathit{Stack}, t : \mathit{Tag}^?, C : \mathit{List}(\mathit{CallId})) : \mathit{Stack}^?$
  $\mathsf{Access}(\mathsf{AccessRead}, S, t, C) :=$
    $\textbf{bind}\ (i, \_) = \mathsf{FindGranting}(S, \mathsf{AccessRead}, t)\ \textbf{in}$
    (* Disable all $\mathsf{Unique}$ above $i$; error out if any of them is protected.  *)
    $\textbf{if}\ \{S(j).\textsc{protector} \mid j \in (i, |S|) \land S(j).\textsc{perm} = \mathsf{Unique}\} \cap C = \emptyset$
    $\textbf{then}\ S\ \textbf{with}_{\ j \in (i,|S|) \land S(j).\textsc{perm} = \mathsf{Unique}}[j := (S(j)\ \textbf{with}\ [\textsc{perm} := \mathsf{Disabled}])]\ \textbf{else}\ \bot$
  $\mathsf{Access}(\mathsf{AccessWrite}, S, t, C) :=$
    $\textbf{bind}\ (i, p) = \mathsf{FindGranting}(S, \mathsf{AccessWrite}, t)\ \textbf{in}$
    $\textbf{bind}\ j = \mathsf{FindFirstWIncompat}(S, i, p)\ \textbf{in}$
    (* Remove items at $j$ and above; error out if any of them is protected.  *)
    $\textbf{if}\ \{S(i').\textsc{protector} \mid i' \in [j, |S|)\} \cap C = \emptyset$
    $\textbf{then}\ S|_{[0,j)}\ \textbf{else}\ \bot$


(* Read and write accesses are just lifted pointwise for each location.  *)
$\mathsf{MemAccessed}(\xi : \mathit{Stacks}, C : \mathit{List}(\mathit{CallId}), a, \mathsf{Pointer}(\ell, t), n : \mathbb{N}) : \mathit{Stacks}^? :=$
  $\xi\ \textbf{with}_{\ \ell' \in [\ell, \ell+n)}[\ell' := \mathsf{Access}(a, \xi(\ell'), t, C)]$


(* Deallocation is like a write, but also errors out if any item is still protected.  *)
$\mathsf{Dealloc}(S : \mathit{Stack}, t : \mathit{Tag}^?, C : \mathit{List}(\mathit{CallId})) : 1^? :=$
  $\textbf{bind}\ \_ = \mathsf{FindGranting}(S, \mathsf{AccessWrite}, t)\ \textbf{in}$
  $\textbf{if}\ \{\iota.\textsc{protector} \mid \iota \in S\} \cap C = \emptyset\ \textbf{then}\ ()\ \textbf{else}\ \bot$
$\mathsf{MemDeallocated}(\xi : \mathit{Stacks}, C : \mathit{List}(\mathit{CallId}), \mathsf{Pointer}(\ell, t), n : \mathbb{N}) : \mathit{Stacks}^? :=$
  $\xi\ \textbf{with}_{\ \ell' \in [\ell, \ell+n)}[\ell' := (\textbf{bind}\ \_ = \mathsf{Dealloc}(\xi(\ell'), t, C)\ \textbf{in}\ \bot)]$
```

Figure 17.2: Stacked Borrows access semantics.

for every location in the new memory block. Deallocation is very similar to a write access, except that it is an error to deallocate memory while *any* item of a deallocated location is protected.

17.3 Retagging

In §15 and §16, Stacked Borrows took some extra actions any time a reference was created (NEW-MUTABLE-REF and NEW-SHARED-REF-2) or cast to a raw pointer (NEW-MUTABLE-RAW-2). We then explained `retag` as basically sugar for one of these operations. It turns out that we can simplify implementation and formalization by swapping things around: an assignment like `let x = &mut expr` in Rust becomes `let x = &mut expr; retag x` in our language with explicit retagging, and `retag` takes care of assigning a new tag (following NEW-MUTABLE-REF, in this case). The `&mut expr` itself just keeps the old tag. This is also the case for creating shared references and casting references to raw pointers. For the latter, we insert a `retag[raw]` to let retagging know that this is part of a reference-to-raw-pointer cast. Altogether, this means that accesses and `retag` cover everything we discussed in the previous chapters, without the need to instrument reference creation or casts.[5] To summarize, retagging happens for all references passed as function arguments or received as return value, for each assignment of reference type, and for casts from reference to raw pointers.

So, let us take a closer look at retagging. Or rather, let us first look at its core helper function, *reborrowing*, defined as Reborrow in Figure 17.3. Reborrowing takes as parameters the tag t_{old} and location ℓ of the *old* pointer (remember that retagging always starts with an existing pointer and updates its tag), the type τ the pointer points to, the tag t_{new} and pointer kind k of the *new* pointer, and *prot*, an optional call ID the new items will be protected by. The pointer kind k here determines if the new pointer is a reference, raw pointer, or `Box`; this affects which new permission p_{new} will be used for the items we are about to add.

Reborrowing proceeds as follows for each location ℓ' covered by the pointer (that list of locations is computed by $\mathsf{FrozenIter}(\ell, \tau)$):

1. Compute the permission p_{new} we are going to grant to t_{new} for ℓ' (done by NewPerm):
   - If `x` is a mutable reference (`&mut T`), $p_{new} :=$ Unique.
   - If `x` is a mutable raw pointer (`*mut T`), $p_{new} :=$ SharedRW.
   - If `x` is a shared reference (`&T`) or a constant raw pointer (`*const T`)[6], we check if ℓ' is *frozen* (definitely outside an `UnsafeCell`) or not. This is determined by FrozenIter based on the type τ. For frozen locations we use $p_{new} :=$ SharedRO; otherwise $p_{new} :=$ SharedRW.
2. The new item we want to add is $\iota_{new} := (p_{new}, t_{new}, \mathit{prot})$. Proceed in GrantTag to actually add that item to the stack.
3. The "access" a that this operation corresponds to is a write access if p_{new} permits writing, and a read access otherwise.
4. Find the granting item for this access with tag t_{old} in $\varsigma.\textsc{stacks}(\ell')$.

[5] It also further emphasizes that where we insert `retag` instructions is a key knob to decide how restrictive Stacked Borrows is (as described at the end of §15.4).

[6] Rust actually has two kinds of raw pointers, mutable and constant. Constant raw pointers correspond to `const` pointers in C. We have not discussed constant raw pointers before to keep the discussion more focused. They behave basically like untagged shared references.

```
(* Inserts $\iota$ into $S$ at index $i$.  *)
$\mathsf{InsertAt}(S : \mathit{Stack}, \iota : \mathit{Item}, i : \mathbb{N}) := S|_{[0,i)} +\!\!+ [\iota] +\!\!+ S|_{[i,|S|)}$

(* Computes the new stack after inserting new item $\iota_{new}$ derived from old tag $t_{old}$.
   Also depends on the list of active calls $C$ (used by Access).  *)
$\mathsf{GrantTag}(S : \mathit{Stack}, t_{old} : \mathit{Tag}^?, \iota_{new} : \mathit{Item}, C : \mathit{List}(\mathit{CallId})) : \mathit{Stack}^? :=$
  (* Determine the "access" this operation corresponds to. Step (3).  *)
  let $a = (\textbf{if } \mathsf{Grants}(\iota_{new}.\textsc{perm}, \mathsf{AccessWrite}) \textbf{ then } \mathsf{AccessWrite} \textbf{ else } \mathsf{AccessRead})$ in
  (* Find the item matching the old tag. Step (4).  *)
  bind $(i, p) = \mathsf{FindGranting}(S, a, t_{old})$ in
  if $\iota_{new}.\textsc{perm} = \mathsf{SharedRW}$ then
    (* A SharedRW just gets inserted next to the granting item. Step (5).  *)
    bind $j = \mathsf{FindFirstWIncompat}(S, i, p)$ in $\mathsf{InsertAt}(S, \iota_{new}, j)$
  else
    (* Otherwise, perform the effects of an access and add item at the top. Step (5).  *)
    bind $S' = \mathsf{Access}(a, S, t_{old}, C)$ in $\mathsf{InsertAt}(S', \iota_{new}, |S'|)$

(* Lists all locations covered by a value of type $\tau$ stored at location $\ell$, and indicate for each location
   whether it is frozen (outside an UnsafeCell) or not.  *)
$\mathsf{FrozenIter}(\ell : \mathit{Loc}, \tau : \mathit{Type}) : \mathit{List}(\mathit{Loc} \times \mathbb{B}) :=$
    $\big[(\ell', b) \mid \ell' \in [\ell, \ell + |\tau|) \land b = (\ell' \text{ is } \mathit{outside} \text{ of an } \texttt{UnsafeCell})\big]$

(* Computes the new permission granted to a reborrow with pointer kind $k$; $\mathit{fr}$ indicates if this location
   is frozen or not. Step (1).  *)
$\mathsf{NewPerm}(k : \mathit{PtrKind}, \mathit{fr} : \mathbb{B}) : \mathit{Permission}$
   (* Mutable references and boxes get Unique permission.  *)
   $\mathsf{NewPerm}(\mathsf{Ref}(\mathsf{Mutable}) \mid \mathsf{Box}, \_)$                      $:=$  $\mathsf{Unique}$
   (* Mutable raw pointers get SharedRW permission.  *)
   $\mathsf{NewPerm}(\mathsf{Raw}(\mathsf{Mutable}), \_)$                            $:=$  $\mathsf{SharedRW}$
   (* Shared references and const raw pointers depend on whether the location is frozen.  *)
   $\mathsf{NewPerm}(\mathsf{Ref}(\mathsf{Immutable}) \mid \mathsf{Raw}(\mathsf{Immutable}), \mathit{fr})$  $:=$  if $\mathit{fr}$ then $\mathsf{SharedRO}$ else $\mathsf{SharedRW}$

(* Reborrow the memory pointed to by $\mathsf{Pointer}(\ell, t_{old})$ for pointer kind $k$ and pointee type $\tau$. $\mathit{prot}$
   indicates if the item should be protected.  *)
$\mathsf{Reborrow}(\xi : \mathit{Stacks}, C : \mathit{List}(\mathit{CallId}), \mathsf{Pointer}(\ell, t_{old}), \tau : \mathit{Type}, k : \mathit{PtrKind}, t_{new} : \mathit{Tag}^?, \mathit{prot} : \mathit{CallId}^?)$
    $: \mathit{Stacks}^? :=$
  (* For each location, determine the new permission and add a corresponding item.  *)
  $\xi$ with$_{(\ell', \mathit{fr}) \in \mathsf{FrozenIter}(\ell, \tau)}$ [ $\ell' :=$
      let $p_{new} = \mathsf{NewPerm}(k, \mathit{fr})$ in (* Step (1).  *)
      let $\iota_{new} = (p_{new}, t_{new}, \mathit{prot})$ in (* Step (2).  *)
      $\mathsf{GrantTag}(\xi(\ell'), t_{old}, \iota_{new}, C)$ ]
```

Figure 17.3: Stacked Borrows reborrowing semantics.

5. Check if $p_{new} = \mathsf{SharedRW}$.
   - If yes, add the new item ι_{new} just above the granting item.
   - Otherwise, perform the effects of a read or write access (as determined by a) to ℓ' with t_{old}. Then, push the new item ι_{new} on top of the stack.

The `retag` instruction now just needs to determine the parameters for reborrowing. This is done by the helper function NewTagAndProtector in Figure 17.4. Given the pointer kind k and the kind of retag k' (regular `retag`, `retag[fn]`, or `retag[raw]`), the new tag t_{new} and protector *prot* are computed as follows: raw pointers get new tag $\bot$ without a protector, but only on `retag[raw]` (otherwise they are ignored); `Box` pointers get a fresh tag but no protector; references get a fresh tag and on `retag[fn]` also a protector.

```
(* For a given pointer kind and retag kind, determine the tag and protector used for reborrowing.
   Also returns the new "next tag". *)
$\mathsf{NewTagAndProtector}(n : \mathbb{N}, k : \mathit{PtrKind}, k' : \mathit{RetagKind}, C : \mathit{List}(\mathit{CallId})) : (\mathit{Tag}^? \times \mathit{CallId}^? \times \mathbb{N})^?$
   (* Raw retags are used for reference-to-raw casts: the pointer gets untagged. *)
   $\mathsf{NewTagAndProtector}(n, \mathsf{Raw}(\_), \mathsf{Raw}, \_) := (\bot, \bot, n)$
   (* Boxes get a fresh tag and never a protector. *)
   $\mathsf{NewTagAndProtector}(n, \mathsf{Box}, \_, \_) := (n, \bot, n+1)$
   (* References get a fresh tag and sometimes a protector. *)
   $\mathsf{NewTagAndProtector}(n, \mathsf{Ref}(\_), \_, C) := (n, \textbf{if } k' = \mathsf{FnEntry} \textbf{ then } \mathsf{top}(C) \textbf{ else } \bot, n+1)$

(* Top-level retag operation. Computes the new tag and new state. *)
$\mathsf{Retag}(\varsigma : \mathit{SState}, \mathsf{Pointer}(\ell, t_{old}), \tau : \mathit{Type}, k : \mathit{PtrKind}, k' : \mathit{RetagKind}) : (\mathit{Tag} \times \mathit{SState})^? :=$
  (* If we can compute the next tag and protector, then try reborrowing. *)
  $\textbf{if bind } (t_{new}, \mathit{prot}, n') = \mathsf{NewTagAndProtector}(\varsigma.\textsc{nextptr}, k, k', \varsigma.\textsc{calls}) \textbf{ then}$
    $\textbf{bind } \xi' = \mathsf{Reborrow}(\varsigma.\textsc{stacks}, \varsigma.\textsc{calls}, \mathsf{Pointer}(\ell, t_{old}), \tau, k, t_{new}, \mathit{prot}) \textbf{ in}$
    $(t_{new}, \varsigma \textbf{ with } [\textsc{stacks} := \xi', \textsc{nextptr} := n'])$
  $\textbf{else}$
    (* Otherwise, do nothing. *)
    $(t_{old}, \varsigma)$
```

Figure 17.4: Stacked Borrows retagging semantics.

Chapter 18
Evaluation

As described in the introduction, we have evaluated Stacked Borrows in two ways. First, to ensure that the model actually accepts enough real Rust code to be a realistic option, we implemented this model in an existing Rust interpreter called *Miri*.[1] Secondly, we formalized the informal proof sketches given in previous chapters in Coq.

18.1 Miri

We implemented Stacked Borrows in Miri to be able to test existing bodies of **`unsafe`** Rust code and make sure the model we propose is not completely in contradiction with how real Rust code gets written. Moreover, this also served to test a large body of *safe* code (including code that relies on non-lexical lifetimes), empirically verifying that Stacked Borrows is indeed a dynamic version of the borrow checker and accepts strictly more code than its static counterpart.

Having an implementation of Stacked Borrows also proved extremely valuable during development; it allowed us to quickly iterate with new rules and validate them against a few key test cases, easily discarding ideas that did not have the intended effect. We only started formalization after the model survived our testing in Miri.

Implementation. Miri is an interpreter that operates on the MIR, an intermediate representation in the Rust compiler. MIR is an imperative, non-SSA, control-flow-graph based IR with a fairly small set of operations, which makes it well suited for an interpreter. The interpreter was already able to run basic Rust programs, so all we had to do was add the extra checks imposed by Stacked Borrows to it.

It was relatively straightforward to translate our operational rules into Rust code that runs whenever the interpreted program reads from or writes to memory. More interesting was the handling of **`retag`**; for this we decided to add a new primitive MIR statement and implemented a compiler pass (conceptually part of MIR construction in the compiler) that inserts **`retag`** statements automatically at the appropriate locations.

The implementation is fairly naive; the only optimization worth mentioning is that instead of storing a borrow stack per location, we store one stack for an adjacent range of locations that all share the same stack. Any memory access covering that entire range (say, a 4-byte access where these 4 locations share their stack) just performs the Stacked Borrows

[1] Available online at `https://github.com/rust-lang/miri/`.

read/write action once. The ranges get automatically split and merged as the stack of adjacent locations diverges and re-unifies. For example, memory storing an `i32` that is never subject to byte-wise accesses just needs a single borrow stack instead of 4 of them.

These changes have all been accepted into Miri, so running Miri now by default checks the program for conformity with Stacked Borrows.[2] In particular, all our optimization counterexamples from §15 and §16 are flagged appropriately by Miri.

Testing and results. We ran part of the Rust standard library test suites in Miri. Miri is not very efficient, but it is fast enough for this purpose: the overall slowdown of Miri with Stacked Borrows compared to compiling the code with optimizations and running that is around 1000x; compared to Miri without Stacked Borrows the slowdown is around 2x. We did not run the parts that are mostly concerned with host OS interaction (such as network and file system access) because Miri does not support those operations. The part that we *did* test includes key data structures (*e.g.*, `Vec`, `VecDeque`, `BTreeMap`, `HashMap`), primitives for interior mutability (`Cell`, `RefCell`), string formatting, arrays/slices, and iterator combinators. All of these involve interesting `unsafe` code using plenty of raw pointers.

Across all these tests, we found in total 7 cases of `unsafe` code not following Stacked Borrows. In two of these cases,[3] the code accidentally turned a shared reference into a mutable one. This is a pattern that the Rust designers explicitly forbade since day one; there is no question that such code is illegal, and thus patches to fix it were promptly accepted.

In three cases,[4] code created mutable references that were never used to access memory, but for Stacked Borrows, the mere fact that they exist already makes them in conflict with existing pointers (because creating a mutable reference is considered a write access). These could all be fixed by adapting the code, but one of them required some refactoring to use fewer mutable references and more raw pointers in `BTreeMap`. The fourth case[5] is similar, but this time was only visible to `unsafe` clients of `Vec`. These clients were making assumptions about `Vec` that are explicitly supported by the documentation, but were again violated by `Vec` internally creating (but not accessing) a conflicting mutable reference. We submitted patches to fix all of these cases, and they were all accepted. Still, this indicates that an interesting next step for Stacked Borrows is to be less aggressive about asserting uniqueness of mutable references that are created but never used.

The final case involves protectors.[6] That code passes around references wrapped inside a `struct`, and one function ends up invalidating these references while it is still ongoing, leading to an item with an active protector being popped. When we encountered this issue, instead of changing the code, we decided to adjust our model to accommodate this: we restricted `retag` to only operate on "bare" references, and not perform retagging of references inside compound types such as `struct`s. At that point it was unclear if the code should be considered correct or not, so this choice was helpful for us in order to focus our testing on other aspects of Stacked Borrows. Since then, however, after completing

[2] Miri also checks for many other cases of undefined behavior, such as illegal use of uninitialized memory.

[3] Jung, "Fix futures creating aliasing mutable and shared ref", 2018 [Jun18b]; Jung, "Fix str mutating through a ptr derived from &self", 2019 [Jun19c].

[4] Jung, "VecDeque: fix for stacked borrows", 2018 [Jun18d]; Jung, "Fix LinkedList invalidating mutable references", 2019 [Jun19a]; Jung, "Fix overlapping references in BTree", 2019 [Jun19b].

[5] Jung, "Vec::push invalidates interior references even when it does not reallocate", 2019 [Jun19e].

[6] Jung, "VecDeque's Drain::drop writes to memory that a shared reference points to", 2019 [Jun19f].

implementation and formalization of Stacked Borrows, new evidence has been discovered by the Rust community showing that this code is violating aliasing assumptions currently being made by the compiler. What remains unclear is whether this should be fixed by changing the code, changing the alias analysis, or both (and giving the programmer more control). Miri will be a valuable tool to explore this trade-off in the future.

Miri is also available as a "Tool" on the Rust Playground, where small snippets of code can be tested directly in the browser.[7] Rust developers increasingly use Miri to check their code for undefined behavior, and if they are surprised by the result—*e.g.*, because code they deemed legal violates Stacked Borrows—we will often hear about that through bug reports or one of the various Rust community channels. So far, this has not uncovered any unsafe code patterns that are incompatible with Stacked Borrows beyond the ones discussed above.

We are now running Miri on the aforementioned Rust test suites every night, to continuously monitor the standard library for new cases of undefined behavior and Stacked Borrows violations. Some projects, including `HashMap` (which is a separate library that is also shipped as part of the Rust standard library), have also decided to run Miri as part of their continuous integration for pre-merge testing of pull requests. The Miri documentation contains a growing list of bugs[8] that have been found that way in various parts of the Rust ecosystem.

Overall, we believe that this evaluation demonstrates that Stacked Borrows is a good fit for Rust. It might seem surprising that Rust developers would follow the stack discipline mandated by Stacked Borrows when mixing raw pointers and mutable references. Our hypothesis is that this works well because (a) raw pointers are currently untagged, so as long as *any* raw pointers may be used, *all* of them may be used, and (b) developers are aware that violating uniqueness of mutable references is not allowed, and already try as best they can to avoid it. Nevertheless, they currently do not know exactly what they can and cannot do. The goal of our work is to be able to give them a clearer answer to the questions that frequently arise in this space.

18.2 Coq formalization

In this dissertation, we have given informal proof sketches of several representative optimizations enabled by Stacked Borrows. To further increase confidence in the semantics, we formalized its operational rules (6k lines of Coq, including proofs showing some key properties of the operational semantics) and turned our proof sketches into mechanized correctness proofs of all example transformations mentioned in previous chapters. To reason about transformations in the presence of unknown code, we built a simulation framework (5k lines of Coq) based on open simulations.[9] See the Stacked Borrows paper[10] and its technical appendix for further details.

[7] Available online at `https://play.rust-lang.org/`.

[8] `https://github.com/rust-lang/miri/#bugs-found-by-miri`

[9] Hur *et al.*, "The marriage of bisimulations and Kripke logical relations", 2012 [Hur+12].

[10] Jung *et al.*, "Stacked Borrows: An aliasing model for Rust", 2020 [Jun+20a].

Chapter 19

Related work

In terms of language semantics to enable better alias analysis, the most closely related to Stacked Borrows are C's *strict aliasing rules* and its `restrict` qualifier.

Strict aliasing rules. These rules, broadly speaking, allow the compiler to assume that pointers to different types do not alias. Every object (in memory) has a particular *effective type*, which is used to rule out (most) accesses happening through differently-typed pointers. This is also often referred to as *type-based alias analysis* (TBAA).

The C standard[1] describes the strict aliasing rules in an axiomatic style. However, in particular the interaction of the strict aliasing rules with unions is not very clear in the standard. Under some conditions, the C standard permits "type-punning" through unions, meaning that a read with the "wrong" type is sometimes allowed, but the details are fuzzy.[2]

The first CompCert memory model[3] formalizes a very strong operational version of the strict aliasing rules that entirely disallows type-punning through unions. However, this is not exploited for the purpose of program analyses or transformations. Later versions of CompCert moved to a simpler memory model that does not impose any strict aliasing rules.[4] CompCert also features a formally verified alias analysis,[5] but that analysis does not exploit extra aliasing information provided by the language.

Krebbers[6] gives another operational account of strict aliasing, with rules for type-punning through unions that are based on the GCC developers' interpretation of the C standard. He also shows a basic non-aliasing "soundness" theorem, but no compiler transformations.

Long-standing compiler bugs in both GCC[7] and clang[8] indicate that exploiting strict aliasing rules for optimizations is tricky and easily leads to miscompilations. One issue is that C does allow (or, at least, the standard does not forbid) writes that change the effective type, leading to an inconsistency with the analysis that compilers perform.[9] Moreover, many real-world C idioms are violating the strict aliasing rules, and many programmers consider the strict aliasing rules to be overly restrictive.[10] For these reasons, many large projects, including the Linux kernel, outright disable type-based alias analysis, essentially opting-in to a version of C with less undefined behavior and fewer optimizations.

Moreover, type-based alias analysis is comparatively weak. In particular, it cannot be used to reason about unknown code; the compiler must

[1] ISO Working Group 14, "Programming languages – C", 2018 [ISO18].

[2] The standard says that such type-punning reads are permitted through a union if several conditions all apply, including "that a declaration of the completed type of the union is visible" (§6.5.2.3 p6). It is not clear exactly what that means.

[3] Leroy and Blazy, "Formal verification of a C-like memory model and its uses for verifying program transformations", 2008 [LB08].

[4] Leroy *et al.*, "The CompCert memory model, version 2", 2012 [Ler+12].

[5] Robert and Leroy, "A formally-verified alias analysis", 2012 [RL12].

[6] Krebbers, "Aliasing restrictions of C11 formalized in Coq", 2013 [Kre13].

[7] Hazeghi, "Store motion causes wrong code for union access at -O3", 2013 [Haz13].

[8] De Fraine, "Wrong results with union and strict-aliasing", 2014 [De 14].

[9] Sewell, "Type-changing and effective type", 2019 [Sew19].

[10] Memarian and Sewell, "N2014: What is C in practice? (Cerberus survey v2): Analysis of responses", 2016 [MS16].

know the types of both memory accesses involved to determine if they might alias. In contrast, as we have shown, Stacked Borrows enables optimizations involving unknown code.

Hathhorn, Ellison, and Rosu[11] formalize C in their K framework,[12] including the rules around strict aliasing. Since K formalizations are executable, this also provides a way to check real code for conformance with the model. However, the paper does not give many formal details of how the strict aliasing rules are defined. It remains unclear how the authors decided to resolve the open questions mentioned above, and whether that choice is compatible with real world C code and compilers. There are also no theorems establishing that their model permits the program transformations that motivate these rules.

Another tool for detecting at least some violations of strict aliasing rules is libcrunch,[13] which however is neither sound nor complete.[14]

`restrict`-qualified pointers. Since C99, the C language supports adding the `restrict` qualifier to pointer types, an explicit annotation that can be used by the programmer to give non-aliasing information to the compiler. This qualifier indicates that accesses performed through this pointer and pointers derived from it cannot alias with other accesses. One common application of `restrict` is in tight numerical loops, *e.g.*, matrix multiplication, where assuming that the output matrix does not alias either of the input matrices can make the difference between a fully vectorized loop using SIMD (single instruction multiple data) operations and purely scalar (unvectorized) code.

Conceptually, `restrict` is closely related to Stacked Borrows. In fact, the Rust compiler (which uses LLVM as its backend) aims to emit `noalias`, the LLVM equivalent of `restrict`, as annotations for references in function argument position. LLVM considers pointers to not alias when none of their respective accesses conflict (*i.e.*, when all their accesses can be mutually reordered); this means two read-only pointers never "alias" even if the memory ranges they point to overlap. Rust exploits this to also emit `noalias` for shared reference, except in the presence of `UnsafeCell`.

However, the exact semantics of `noalias` and `restrict` are unclear, in particular when considering general pointers and not just function arguments.[15] Even for function arguments, uncertainty in the semantics led to several LLVM bugs.[16] For this reason, Rust currently does not emit `noalias` annotations for mutable references (however, they are still emitted for shared references without interior mutability).

The aforementioned formalization of C in the K framework[17] also includes some semantics for `restrict`, and a way to check programs for conformance. However, similar to strict aliasing, the details of how the axiomatic, informal standard got interpreted into a formal operational semantics are not covered in the paper. For example, the standard defines a notion of pointer expressions being "based on" an object in terms of whether modifying the object to point elsewhere would change the value of the expression (§6.7.3.1 p3). This definition is rather problematic when also considering that control flow could change as a result of making the object point elsewhere. So instead, the K formalization relies on tracking

[11] Hathhorn, Ellison, and Rosu, "Defining the undefinedness of C", 2015 [HER15].

[12] Rosu and Serbanuta, "An overview of the K semantic framework", 2010 [RS10].

[13] Available online at `https://github.com/stephenrkell/libcrunch`.

[14] Regehr, "Undefined behavior in 2017", 2017 [Reg17].

[15] Function arguments are easier because there is a clear notion of "scope" that one could use to say for *how long* the aliasing guarantee must hold.

[16] Gohman, "Incorrect liveness in DeadStoreElimination", 2015 [Goh15]; Popov, "Loop unrolling incorrectly duplicates noalias metadata", 2018 [Pop18].

[17] Hathhorn, Ellison, and Rosu, "Defining the undefinedness of C", 2015 [HER15].

pointer provenance with “tags”, like Stacked Borrows. But it remains unclear (and the paper does not discuss) how well this interpretation of the standard reflects its intent, or how well it matches the interpretation of the standard by programmers and compilers.

Fortran. Loosely related to C’s `restrict` are the aliasing rules of Fortran,[18] which disallow function parameters to alias (unless the programmer specifically marks them as potentially aliasing). In fact, competing with Fortran compilers was a primary motivation for adding `restrict` to C.[19] Nguyen and Irigoin[20] describe a tool that dynamically checks for aliasing violations in Fortran programs, but they do not verify any program transformations.

Low-level language semantics. There is a large body of work on formalizing the semantics of C or LLVM (as representative examples of highly optimized “low-level” languages) and in particular their handling of pointers and pointer provenance.[21] However, with the exception of what was explained above, this work does not account for strict aliasing rules and the `restrict` qualifier. They, instead, focus on orthogonal complications, such as the handling of casts between integers and pointers, and the use of pointer provenance to prevent pointer arithmetic across object boundaries.

We have specifically designed Stacked Borrows to *not* assume that all pointers have a known provenance, by adding the notion of an “untagged” pointer. This means we should be able to basically take any of the existing approaches to model integer-pointer casts, and equip it with a variant of Stacked Borrows that handles pointers of unknown provenance as untagged.

[18] ANSI, “Programming language FORTRAN”, 1978 [ANS78].

[19] Drepper, “Memory part 5: What programmers can do”, 2007 [Dre07].

[20] Nguyen and Irigoin, “Alias verification for Fortran code optimization”, 2003 [NI03].

[21] Memarian *et al.*, “Exploring C semantics and pointer provenance”, 2019 [Mem+19]; Krebbers, “The C standard formalized in Coq”, 2015 [Kre15]; Kang *et al.*, “A formal C memory model supporting integer-pointer casts”, 2015 [Kan+15]; Lee *et al.*, “Reconciling high-level optimizations and low-level code in LLVM”, 2018 [Lee+18]; Hathhorn, Ellison, and Rosu, “Defining the undefinedness of C”, 2015 [HER15]; Norrish, “C formalised in HOL”, 1998 [Nor98].

Chapter 20

Conclusion

The goal of this dissertation was to demonstrate that a complex real-world language such as Rust can be formally understood and that we can prove meaningful theorems about it, not just on paper but in a machine-checked proof assistant. Moreover, we can use formal methods to further the development of Rust and help its designers as they flesh out the details of the specification.

To support this claim, we have presented three major bodies of work: Iris, RustBelt, and Stacked Borrows.

Iris is a framework for building higher-order concurrent separation logics with a focus on deriving custom reasoning principles from lower-level primitives. It forms the foundation necessary to state and prove the theorems of RustBelt, and to do so in Coq. For this dissertation, we have cut out a slice of the logic that covers everything needed for the semantic model of Rust. But Iris is certainly not specific to RustBelt, and indeed the list of publications that directly make use of Iris[1] is growing rapidly. In each of the last three years (2018–2020), there were *several* papers at POPL that make use of Iris, and in 2018 and 2020 there was a POPL tutorial to teach people how to use the logic. Iris is run as an open-source project, and the number of people participating in discussions in the chat room and submitting improvements to the Coq implementation has been rising as well.

RustBelt consists of λ_{Rust}, a core language calculus, as well as formal models of some key Rust libraries containing `unsafe` code, and a safety proof of both of these components. λ_{Rust} models some key aspects of the Rust type system: ownership, borrowing, and lifetimes. The safety proof is carried out by means of a *semantic model* (or *logical relation*) expressed in Iris, building on the novel *lifetime logic* which equips separation logic with a notion of *borrows* and associated lifetimes. This model for the first time enables the verification of Rust libraries implemented with `unsafe` code, ensuring that their public API soundly encapsulates the unsafety. This work has not only provided some key insights into the structure of the Rust type system and, in particular, interior mutability; we have also been able to fix a bug in the Rust standard library[2] and follow-on work has found and fixed another bug.[3] But the impact of RustBelt on Rust goes further than fixing a few bugs: discussion of a possible formal model[4] also played a role when the concept of *pinning* was introduced to Rust.[5] (Pinning can make an instance of a type "immovable" in memory so that other data structures can safely point to it without borrowing it.)

[1] see `https://iris-project.org/`

[2] Jung, "MutexGuard<Cell<i32>> must not be Sync", 2017 [Jun17].

[3] Jourdan, "Insufficient synchronization in Arc::get_mut", 2018 [Jou18].

[4] Jung, "A formal look at pinning", 2018 [Jun18a]; Jung, "Safe intrusive collections with pinning", 2018 [Jun18c].

[5] withoutboats, "Tracking issue for pin APIs (RFC 2349)", 2018 [wit18].

When extending the API of types that were verified in RustBelt, the Rust teams are sometimes even delaying stabilization of that new API until our soundness proof is extended accordingly.[6]

Stacked Borrows aims not to describe Rust as it exists, but to help evolve the language by giving some precise aliasing rules to Rust's reference types, with the goal of enabling more type-based optimizations. This work was done in close collaboration with the Rust language team, who are wary of introducing more aggressive optimizations into the compiler until there is a better understanding of what `unsafe` code is and is not allowed to do. Stacked Borrows is not the official aliasing model of Rust, but it is by far the most concrete proposal on the table, and so whenever questions of aliasing come up in discussions, Stacked Borrows is used as the *de facto* standard. We intend to continue to work with the Rust community and development teams to adjust Stacked Borrows to their needs, with the goal of eventually making a variant of Stacked Borrows part of the official semantics of Rust.

20.1 Future work

As always, each time a scientific question is answered, at least two more questions pop up in its place. As such, there are a lot of possibilities for future research on top of the results of this dissertation.

Iris. The two most frequently requested extensions for Iris are *linearity* and support for reasoning about *liveness*.

Linearity was already discussed in §6.2. The current state of the art of linear reasoning with Iris is reflected in Iron,[7] a logic that encodes resources representing *obligations* (such as the obligation to free some memory) as predicates in Iris, and then shows that these predicates themselves form a proper separation logic. However, a more direct approach to linearity could be to generalize the resource algebras of Iris to *ordered resource algebras* (oRA). The idea here is to not fix extension order ($\preccurlyeq$) as the relation under which Iris assertions must be closed, but instead to let the oRA pick a suitable order. Affine resources would pick the extension order, linear resources would pick an order that just contains the reflexive pairs (*i.e.*, the diagonal), and conceivably there could be resources that are somewhere "in between" affine and linear.[8] A possible set of axioms for oRAs has been described in §3.2 of the MoSeL paper.[9] But many questions remain open; in particular, so far it is unclear what the right notion of frame-preserving updates is for oRAs and how to obtain principles such as "absence of memory leaks" as instances of something more general, in the usual Iris way.

Liveness is notoriously hard to reason about in step-indexed logics, because the step-index restricts the statements of the logic to only concern traces of finite length. Some limited forms of liveness reasoning are still possible,[10] but that approach falls short of the kind of reasoning performed *e.g.*, in TaDA Live.[11] One possible avenue for progress here could be to extend step-indices from being natural numbers to being (possibly infinite) *ordinals*; this is sometimes called *transfinite step-indexing*.[12] An alternative

[6] Liebow-Feeser, "Tracking issue for Ref/RefMut::map_split", 2018 [Lie18].

[7] Bizjak *et al.*, "Iron: Managing obligations in higher-order concurrent separation logic", 2019 [Biz+19].

[8] Cao, Cuellar, and Appel, "Bringing order to the separation logic jungle", 2017 [CCA17].

[9] Krebbers *et al.*, "MoSeL: A general, extensible modal framework for interactive proofs in separation logic", 2018 [Kre+18].

[10] Tassarotti, Jung, and Harper, "A higher-order logic for concurrent termination-preserving refinement", 2017 [TJH17].

[11] D'Osualdo *et al.*, "TaDA Live: Compositional reasoning for termination of fine-grained concurrent programs", 2019 [DOs+19].

[12] Svendsen, Sieczkowski, and Birkedal, "Transfinite step-indexing: Decoupling concrete and logical steps", 2016 [SSB16].

approach would be to entirely avoid step-indexing, and explore how much of Iris can still be used.

Another, smaller extension worth exploring has been mentioned in §7.2: it looks like the authoritative RA can be generalized further to support a Views-style[13] reification function/relation, which could simplify proofs (removing the need for `to_fheap` in §4.4) and make the authoritative RA more generally applicable, such as when additional side-conditions must be imposed on the authoritative value (*e.g.*, to model a monotonically growing partial bijection, which frequently comes up when considering heap locations in a binary logical relation).

RustBelt. Even without extending the set of language facilities that we can model, there are some interesting `unsafe`ly implemented libraries that we have not yet verified in RustBelt. We already mentioned at the end of §13.1 that there are some operations on `Cell` that our current model does not support. Beyond that, two interesting libraries to verify (with potentially far-reaching consequences for the entire model) are `Pin`, which we already mentioned above, and "sound unchecked indexing" using *generativity of lifetimes*,[14] a Rust adaptation of the "branded types" pattern also used in Haskell and OCaml.[15]

Another avenue for future work is supporting more Rust type system features. First and foremost, while λ_{Rust} can handle many cases of non-lexical lifetimes (NLL) just fine (such as the first two examples from an early blog post on NLL[16]—the third example is not accepted by the final version of NLL), there are still some gaps between the borrow checker in Rust and the one in λ_{Rust}: λ_{Rust} does not support *two-phase borrows*,[17] a special variant of mutable borrowing where the borrow is actually *shared* (and read-only) for some time before becoming "activated" and turning into a proper mutable borrow. The handling of shared references in λ_{Rust} is also more restrictive than that of current Rust. Both of these can in many cases be worked around through a more clever translation, but it is unclear if that is possible in general. And the Rust compiler team is already working on the next generation of borrow checking in the form of *Polonius*,[18] which is even harder to compare to λ_{Rust}: where previous borrow checkers were implemented as subtle dataflow analyses, Polonius is defined by translating Rust functions into a dialect of Datalog. Some programs newly accepted by Polonius, such as "problem case #3" from the NLL blog post, *can* be typechecked in λ_{Rust} but require the somewhat odd F-EQUALIZE rule (see §9.4). Further exploration is needed to determine how the two systems compare on more examples. However, due to its foundation in logic programming, Polonius also opens some exciting new avenues for a formal analysis—it now seems feasible to relate λ_{Rust} to the actual analysis performed by the Rust compiler!

A tricky aspect of the Rust type system that λ_{Rust} glosses over entirely is *traits*, Rust's take on Haskell-style type classes. Rust crucially relies on *coherence* of traits, which means that for a given trait and type, there can be no more than one implementation of the trait for the type. However, there are some tricky corner cases where the current trait system allows coherence to be violated.[19] Even more interestingly, recently a soundness

[13] Dinsdale-Young *et al.*, "Views: Compositional reasoning for concurrent programs", 2013 [Din+13].

[14] Beingessner, "You can't spell trust without Rust", 2015 [Bei15].

[15] Kiselyov and Shan, "Lightweight static capabilities", 2007 [KS07].

[16] Matsakis, "Non-lexical lifetimes: Introduction", 2016 [Mat16b].

[17] Matsakis, "Nested method calls via two-phase borrowing", 2017 [Mat17].

[18] Matsakis, "An alias-based formulation of the borrow checker", 2018 [Mat18].

[19] Ben-Yehuda, "Coherence can be bypassed by an indirect impl for a trait object", 2019 [Ben19].

bug[20] related to Rust's Pin type was found (and fixed) which involved the *interaction* of unsafe code with the trait system. A full soundness proof for Pin will thus involve reasoning about both unsafe code and coherence of trait implementations.

Other unmodeled aspects of the language that would be interesting to explore include the *drop checker*, which makes sure that destructors can be executed correctly even when data structures contain pointer cycles, and *panics*, a mechanism akin to exceptions whereby Rust methods can return "abnormally" but still have resources managed properly via automatic destructors.

An entirely unrelated opportunity for future work is to explore improvements to the lifetime logic. While the lifetime logic proved powerful enough to capture subtle borrowing patterns such as those used by the Rust types Rc and RefCell, it also turned out to often be quite tedious to use. We have to constantly track which fraction of the token for which lifetime is available, and we have to explicitly open and close lifetime inclusion accessors. This is a continuing source of boilerplate in the proofs. A possible improvement here could be to find a way to adopt the approach proposed for *temporary read-only permissions.*[21] The authors introduce a duplicable "read-only" modality with rules that resemble ours for shared references at "simple" types like i32. However, since shared references permit interior mutability, read-only permissions are not suited for directly modeling shared references. Nevertheless, it would be interesting to explore whether this approach can facilitate the tracking of lifetime tokens, just as read-only permissions eliminate the bookkeeping involved in fractional points-to permissions. One challenge here is that λ_{Rust} supports non-lexical lifetimes, whereas read-only permissions are strictly lexical.

Stacked Borrows. The primary goal for future work on Stacked Borrows is to better match the patterns of unsafe code that can be found in the wild. The evaluation with Miri identified two regular patterns of Stacked Borrows violations (conflicting mutable references being created but not used, and references in private fields being guarded by protectors). A more tree-based structure as mentioned in §15.3 could help with the issue for mutable references in particular, but it remains open to what extent this reduces the set of optimizations the compiler can perform. Stacked Borrows also currently lacks good support for *two-phase borrows* (already mentioned above). Two-phase borrows need special treatment in the aliasing model; currently, Stacked Borrows as implemented in Miri treats them as freely aliasiable, thus losing optimizations for such references. We suspect that trees would help to rule out more undesirable behaviors and thus allow proper optimizations for two-phase borrows as well.

Another aspect of Stacked Borrows that could be refined are protectors. Currently, their scope is tied to a function call and thus lost during inlining; a more explicit representation of scopes could help preserve this information. This is closely related to recent proposals in LLVM that aim to improve the interaction of inlining and scoped noalias annotations.

[20] comex, "Unsoundness in Pin", 2019 [com19].

[21] Charguéraud and Pottier, "Temporary read-only permissions for separation logic", 2017 [CP17].

The formal version of Stacked Borrows we presented here works in the context of a sequential language. But of course, Rust is a concurrent language, so Stacked Borrows has to also work well in that setting. Concurrency has recently been implemented in Miri, so we should be able to gather some initial practical experience with Stacked Borrows in concurrent programs soon. It would be interesting to also expand the formal model and its accompanying proofs to consider concurrency. We expect the approach to scale to concurrency as long as all race conditions occur with pointers that have SharedRW permission, and indeed race conditions with other pointers should be data races and thus are not allowed in well-behaved programs.

Finally, it would be great to see Stacked Borrows and RustBelt combined, by showing that the λ_{Rust} type safety and library correctness proofs still hold under this stricter operational semantics—stricter in that it imposes more restrictions on what constitutes well-defined behavior. If we then also relate the λ_{Rust} type system to Polonius, we can complement our optimization correctness proofs by formally verifying that all safe code accepted by the Rust compiler conforms with Stacked Borrows.

Bibliography

[AFM05] Amal Ahmed, Matthew Fluet, and Greg Morrisett. “A step-indexed model of substructural state”. In: *ICFP*. 2005. DOI: `10.1145/1086365.1086376`.

[AFM07] Amal Ahmed, Matthew Fluet, and Greg Morrisett. “$\mathbf{L}^3$: A linear language with locations”. In: *Fundamenta Informaticae* 77.4 (2007).

[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: *TOPLAS* 32.3 (2010). DOI: `10.1145/1709093.1709094`.

[Ahm04] Amal Ahmed. “Semantics of types for mutable state”. PhD thesis. Princeton University, 2004.

[AL91] Martín Abadi and Leslie Lamport. “The existence of refinement mappings”. In: *TCS* 82.2 (1991). DOI: `10.1016/0304-3975(91)90224-P`.

[AM01] Andrew W. Appel and David McAllester. “An indexed model of recursive types for foundational proof-carrying code”. In: *TOPLAS* 23.5 (2001). DOI: `10.1145/504709.504712`.

[Ama+16] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O’Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, *et al.* “Cogent: Verifying high-assurance file system implementations”. In: *ASPLOS*. 2016. DOI: `10.1145/2872362.2872404`.

[ANS78] ANSI. “Programming language FORTRAN”. ANSI X3.9-1978. 1978.

[App+07] Andrew W. Appel, Paul-André Melliès, Christopher Richards, and Jérôme Vouillon. “A very modal model of a modern, major, general type system”. In: *POPL*. 2007. DOI: `10.1145/1190216.1190235`.

[App01] Andrew W. Appel. “Foundational proof-carrying code”. In: *LICS*. 2001. DOI: `10.1109/LICS.2001.932501`.

[App07] Andrew W. Appel. “Compiling with continuations”. Cambridge University Press, 2007.

[App14] Andrew W. Appel. “Program logics – for certified compilers”. Cambridge University Press, 2014. ISBN: 978-1-10-704801-0.

[Ash75] Edward A. Ashcroft. “Proving assertions about parallel programs”. In: *Journal of Computer and System Sciences* 10.1 (1975). DOI: `10.1016/S0022-0000(75)80018-3`.

[Ast+19] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. “Leveraging Rust types for modular specification and verification”. In: *PACMPL* 3.OOPSLA, Article 147 (2019). DOI: `10.1145/3360573`.

[Bat15] Mark John Batty. “The C11 and C++11 concurrency model”. PhD thesis. University of Cambridge, UK, 2015.

[BB17] Aleš Bizjak and Lars Birkedal. “On models of higher-order separation logic”. In: *MFPS*. 2017. DOI: `10.1016/j.entcs.2018.03.016`.

[BCY05] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. “Variables as resource in separation logic”. In: *MFPS*. 2005. DOI: `10.1016/j.entcs.2005.11.059`.

[Bei15] Alexis Beingessner. “You can’t spell trust without Rust”. MA thesis. Carleton University, Ottawa, Ontario, Canada, 2015.

[Ben15] Ariel Ben-Yehuda. “std::thread::JoinGuard (and scoped) are unsound because of reference cycles”. Rust issue #24292. 2015. URL: `https://github.com/rust-lang/rust/issues/24292`.

[Ben19] Ariel Ben-Yehuda. “Coherence can be bypassed by an indirect impl for a trait object”. Rust issue #57893. 2019. URL: `https://github.com/rust-lang/rust/issues/57893`.

[Bie06] Kevin Bierhoff. “Iterator specification with typestates”. In: *SAVCBS*. Portland, Oregon, 2006. ISBN: 1-59593-586-X. DOI: `10.1145/1181195.1181212`.

[Bio17] Christophe Biocca. “std::vec::IntoIter::as_mut_slice borrows &self, returns &mut of contents”. Rust issue #39465. 2017. URL: `https://github.com/rust-lang/rust/issues/39465`.

[Bir+11] 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: *LICS*. 2011. DOI: `10.1109/LICS.2011.16`.

[Biz+19] Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. “Iron: Managing obligations in higher-order concurrent separation logic”. In: *PACMPL* 3.POPL, Article 65 (2019). DOI: `10.1145/3290378`.

[Bor+05] Richard Bornat, Cristiano Calcagno, Peter W. O’Hearn, and Matthew J. Parkinson. “Permission accounting in separation logic”. In: *POPL*. 2005. DOI: `10.1145/1040305.1040327`.

[Boy03] John Boyland. “Checking interference with fractional permissions”. In: *SAS*. Vol. 2694. LNCS. 2003. DOI: `10.1007/3-540-44898-5_4`.

[BPP16] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. “The design and formalization of Mezzo, a permission-based programming language”. In: *TOPLAS* 38.4 (2016). DOI: `10.1145/2837022`.

[Bro06] Stephen D. Brookes. “Variables as resource for shared-memory programs: Semantics and soundness”. In: *MFPS*. 2006. DOI: `10.1016/j.entcs.2006.04.008`.

[Bro07] Stephen Brookes. “A semantics for concurrent separation logic”. In: *TCS* 375.1–3 (2007). DOI: `10.1016/j.tcs.2006.12.034`.

[Bur19] Adam Burch. “Using Rust in Windows”. Blog post. 2019. URL: `https://msrc-blog.microsoft.com/2019/11/07/using-rust-in-windows/`.

[BW81] M. Broy and M. Wirsing. “On the algebraic specification of nondeterministic programming languages”. In: *CAAP*. 1981. DOI: `10.1007/3-540-10828-9_61`.

[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: *JAR* 61.1-4 (2018). DOI: `10.1007/s10817-018-9457-5`.

[Cao17] Qinxiang Cao. Private communication. Apr. 2017.

[CCA17] Qinxiang Cao, Santiago Cuellar, and Andrew W. Appel. “Bringing order to the separation logic jungle”. In: *APLAS*. Vol. 10695. LNCS. 2017. DOI: `10.1007/978-3-319-71237-6_10`.

[Cha+19] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. “Verifying concurrent, crash-safe systems with Perennial”. In: *SOSP*. ACM, 2019. DOI: `10.1145/3341301.3359632`.

[com19] comex. “Unsoundness in Pin”. Rust internals forum discussion. 2019. URL: `https://internals.rust-lang.org/t/unsoundness-in-pin/11311`.

[Coq20] The Coq Team. “The Coq proof assistant”. 2020. URL: `https://coq.inria.fr/`.

[COY07] Cristiano Calcagno, Peter W. O’Hearn, and Hongseok Yang. “Local action and abstract separation logic”. In: *LICS*. 2007. DOI: `10.1109/LICS.2007.30`.

[CP17] Arthur Charguéraud and François Pottier. "Temporary read-only permissions for separation logic". In: *ESOP*. LNCS. 2017. DOI: 10.1007/978-3-662-54434-1_10.

[CPN98] David G. Clarke, John M. Potter, and James Noble. "Ownership types for flexible alias protection". In: *OOPSLA*. 1998. DOI: 10.1145/286936.286947.

[DAB11] Derek Dreyer, Amal Ahmed, and Lars Birkedal. "Logical step-indexed logical relations". In: *LMCS* 7.2:16 (June 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: *PACMPL* 4.POPL, Article 34 (2020). DOI: 10.1145/3371102.

[dDG14] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. "TaDA: A logic for time and data abstraction". In: *ECOOP*. Vol. 8586. LNCS. 2014. DOI: 10.1007/978-3-662-44202-9_9.

[De 14] Bruno De Fraine. "Wrong results with union and strict-aliasing". LLVM issue #21725. 2014. URL: https://bugs.llvm.org/show_bug.cgi?id=21725.

[DF01] Robert DeLine and Manuel Fähndrich. "Enforcing high-level protocols in low-level software". In: *PLDI*. 2001. DOI: 10.1145/381694.378811.

[DGW10] Thomas Dinsdale-Young, Philippa Gardner, and Mark J. Wheelhouse. "Abstraction and refinement for local reasoning". In: *VSTTE*. Vol. 6217. LNCS. 2010. DOI: 10.1007/978-3-642-15057-9_14.

[DHA09] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. "A fresh look at separation algebras and share accounting". In: *APLAS*. Vol. 5904. LNCS. 2009. DOI: 10.1007/978-3-642-10672-9_13.

[Din+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. "Concurrent abstract predicates". In: *ECOOP*. Vol. 6183. LNCS. 2010. 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*. 2013. DOI: 10.1145/2429069.2429104.

[Dod+09] Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. "Deny-guarantee reasoning". In: *ESOP*. Vol. 5502. LNCS. 2009. DOI: 10.1007/978-3-642-00590-9_26.

[Dod+16] Mike Dodds, Suresh Jagannathan, Matthew J. Parkinson, Kasper Svendsen, and Lars Birkedal. "Verifying custom synchronization constructs using higher-order separation logic". In: *TOPLAS* 38.2, Article 4 (2016). DOI: 10.1145/2818638.

[DOs+19] Emanuele D'Osualdo, Azadeh Farzan, Philippa Gardner, and Julian Sutherland. "TaDA Live: Compositional reasoning for termination of fine-grained concurrent programs". In: *CoRR* abs/1901.05750 (2019). Preprint. URL: http://arxiv.org/abs/1901.05750.

[dPJ20] Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. "Spy game: Verifying a local generic solver in Iris". In: *PACMPL* 4.POPL, Article 33 (2020). DOI: 10.1145/3371101.

[Dre+10] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. "A relational modal logic for higher-order stateful ADTs". In: *POPL*. 2010. DOI: 10.1145/1706299.1706323.

[Dre07] Ulrich Drepper. "Memory part 5: What programmers can do". LWN article. 2007. URL: https://lwn.net/Articles/255364/.

[FD02] Manuel Fähndrich and Robert DeLine. "Adoption and focus: Practical linear types for imperative programming". In: *PLDI*. 2002. DOI: 10.1145/512529.512532.

[Fen09] Xinyu Feng. "Local rely-guarantee reasoning". In: *POPL*. 2009. DOI: 10.1145/1480881.1480922.

[FFS07] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. "On the relationship between concurrent separation logic and assume-guarantee reasoning". In: *ESOP*. Vol. 4421. LNCS. 2007. DOI: 10.1007/978-3-540-71316-6_13.

[FGK19] Dan Frumin, Léon Gondelman, and Robbert Krebbers. “Semi-automated reasoning about non-determinism in C expressions”. In: *ESOP*. Vol. 11423. LNCS. Springer, 2019. DOI: 10.1007/978-3-030-17184-1_3.

[FKB18] Dan Frumin, Robbert Krebbers, and Lars Birkedal. “ReLoC: A mechanised relational logic for fine-grained concurrency”. In: *LICS*. 2018. DOI: 10.1145/3209108.3209174.

[FKB21] Dan Frumin, Robbert Krebbers, and Lars Birkedal. “Compositional non-interference for fine-grained concurrent programs”. In: *SP*. To appear. IEEE, 2021.

[FMA06] Matthew Fluet, Greg Morrisett, and Amal Ahmed. “Linear regions are all you need”. In: *ESOP*. LNCS. 2006. DOI: 10.1007/11693024_2.

[Fu+10] Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. “Reasoning about optimistic concurrency using a program logic for history”. In: *CONCUR*. Vol. 6269. LNCS. 2010. DOI: 10.1007/978-3-642-15375-4_27.

[GBC11] Alexey Gotsman, Josh Berdine, and Byron Cook. “Precision and the conjunction rule in concurrent separation logic”. In: *MFPS*. 2011. DOI: 10.1016/j.entcs.2011.09.021.

[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: *ICFP*. 2020. DOI: 10.1145/3408996.

[GLS01] Rakesh Ghiya, Daniel M. Lavery, and David C. Sehr. “On the importance of points-to analysis and other memory disambiguation methods for C programs”. In: *PLDI*. 2001. DOI: 10.1145/378795.378806.

[Goh15] Dan Gohman. “Incorrect liveness in DeadStoreElimination”. LLVM issue #25422. 2015. URL: https://bugs.llvm.org/show_bug.cgi?id=25422.

[Gor+12] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. “Uniqueness and reference immutability for safe parallelism”. In: *OOPSLA*. ACM, 2012. DOI: 10.1145/2384616.2384619.

[Gro+02] Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. “Region-based memory management in Cyclone”. In: *PLDI*. 2002. DOI: 10.1145/512529.512563.

[GS02] Douglas P. Gregor and Sibylle Schupp. “Making the usage of STL safe”. In: *IFIP TC2/WG2.1 Working Conference on Generic Programming*. July 2002. DOI: 10.1007/978-0-387-35672-3_7.

[Har16] Robert Harper. “Practical foundations for programming languages (second edition)”. New York, NY, USA: Cambridge University Press, 2016.

[Haz13] Dara Hazeghi. “Store motion causes wrong code for union access at -O3”. GCC issue #57359. 2013. URL: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=57359.

[HBK20] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. “Actris: Session-type based reasoning in separation logic”. In: *PACMPL* 4.POPL, Article 6 (2020). DOI: 10.1145/3371074.

[HER15] Chris Hathhorn, Chucky Ellison, and Grigore Rosu. “Defining the undefinedness of C”. In: *PLDI*. 2015. DOI: 10.1145/2737924.2737979.

[Hob08] Aquinas Hobor. “Oracle semantics”. PhD thesis. Princeton University, 2008.

[Hor97] Susan Horwitz. “Precise flow-insensitive may-alias analysis is NP-hard”. In: *TOPLAS* 19.1 (1997). DOI: 10.1145/239912.239913.

[Hur+12] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. “The marriage of bisimulations and Kripke logical relations”. In: *POPL*. 2012. DOI: 10.1145/2103656.2103666.

[HV13] Aquinas Hobor and Jules Villard. “The ramifications of sharing in data structures”. In: *POPL*. ACM, 2013. DOI: `10.1145/2429069.2429131`.

[IO01] Samin S. Ishtiaq and Peter W. O’Hearn. “BI as an assertion language for mutable data structures”. In: *POPL*. 2001. DOI: `10.1145/360204.375719`.

[Iri19] Iris Team. “The Iris 3.2 documentation”. 2019. URL: `https://plv.mpi-sws.org/iris/appendix-3.2.pdf`.

[ISO11] ISO Working Group 21. “Programming languages – C++”. ISO/IEC 14882:2011. 2011.

[ISO18] ISO Working Group 14. “Programming languages – C”. ISO/IEC 9899:2018. 2018.

[JB12] Jonas Braband Jensen and Lars Birkedal. “Fictional separation logic”. In: *ESOP*. Vol. 7211. LNCS. 2012. DOI: `10.1007/978-3-642-28869-2_19`.

[Jef18] Alan Jeffrey. “Rust 1.20 caused pinning to become incorrect”. Rust internals forum discussion. 2018. URL: `https://internals.rust-lang.org/t/rust-1-20-caused-pinning-to-become-incorrect/6695`.

[Jim+02] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. “Cyclone: A safe dialect of C”. In: *USENIX ATC*. 2002. URL: `https://www.usenix.org/legacy/publications/library/proceedings/usenix02/jim.html`.

[Jon10] Cliff B. Jones. “The role of auxiliary variables in the formal development of concurrent programs”. In: *Reflections on the Work of C. A. R. Hoare*. 2010. DOI: `10.1007/978-1-84882-912-1_8`.

[Jou18] Jacques-Henri Jourdan. “Insufficient synchronization in Arc::get_mut”. Rust issue #51780. 2018. URL: `https://github.com/rust-lang/rust/issues/51780`.

[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*. 2015. DOI: `10.1145/2676726.2676980`.

[Jun+16] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. “Higher-order ghost state”. In: *ICFP*. 2016. 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: *PACMPL* 2.POPL, Article 66 (2018). DOI: `10.1145/3158154`.

[Jun+18b] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”. In: *JFP* 28, e20 (Nov. 2018). DOI: `10.1017/S0956796818000151`.

[Jun+20a] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. “Stacked Borrows: An aliasing model for Rust”. In: *PACMPL* 4.POPL, Article 41 (2020). DOI: `10.1145/3371109`.

[Jun+20b] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. “Safe systems programming in Rust: The promise and the challenge”. In: *CACM* (2020). To appear.

[Jun+20c] 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: *PACMPL* 4.POPL, Article 45 (2020). DOI: `10.1145/3371113`.

[Jun17] Ralf Jung. “MutexGuard<Cell<i32>> must not be Sync”. Rust issue #41622. 2017. URL: `https://github.com/rust-lang/rust/issues/41622`.

[Jun18a] Ralf Jung. “A formal look at pinning”. 2018. URL: `https://www.ralfj.de/blog/2018/04/05/a-formal-look-at-pinning.html`.

[Jun18b] Ralf Jung. “Fix futures creating aliasing mutable and shared ref”. Rust pull request #56319. 2018. URL: `https://github.com/rust-lang/rust/pull/56319`.

[Jun18c] Ralf Jung. “Safe intrusive collections with pinning”. 2018. URL: https://www.ralfj.de/blog/2018/04/10/safe-intrusive-collections-with-pinning.html.

[Jun18d] Ralf Jung. “VecDeque: fix for stacked borrows”. Rust pull request #56161. 2018. URL: https://github.com/rust-lang/rust/pull/56161.

[Jun19a] Ralf Jung. “Fix LinkedList invalidating mutable references”. Rust pull request #60072. 2019. URL: https://github.com/rust-lang/rust/pull/60072.

[Jun19b] Ralf Jung. “Fix overlapping references in BTree”. Rust pull request #58431. 2019. URL: https://github.com/rust-lang/rust/pull/58431.

[Jun19c] Ralf Jung. “Fix str mutating through a ptr derived from &self”. Rust pull request #58200. 2019. URL: https://github.com/rust-lang/rust/pull/58200.

[Jun19d] 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.

[Jun19e] Ralf Jung. “Vec::push invalidates interior references even when it does not reallocate”. Rust issue #60847. 2019. URL: https://github.com/rust-lang/rust/issues/60847.

[Jun19f] Ralf Jung. “VecDeque’s Drain::drop writes to memory that a shared reference points to”. Rust issue #60076. 2019. URL: https://github.com/rust-lang/rust/issues/60076.

[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. 2017. DOI: 10.4230/LIPIcs.ECOOP.2017.17.

[Kan+15] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. “A formal C memory model supporting integer-pointer casts”. In: *PLDI*. 2015. DOI: 10.1145/2737924.2738005.

[Kle99] Thomas Kleymann. “Hoare logic and auxiliary variables”. In: *Formal Aspects of Computing* 11.5 (Dec. 1999). DOI: 10.1007/s001650050057.

[Klo19] Felix S. Klock. “Breaking news: Non-lexical lifetimes arrives for everyone”. Blog post. 2019. URL: http://blog.pnkfx.org/blog/2019/06/26/breaking-news-non-lexical-lifetimes-arrives-for-everyone/.

[KPS20] Siddharth Krishna, Nanhey Patel, and Dennis E. Shasha. “Verifying concurrent search structure templates”. In: *PLDI*. 2020. DOI: 10.1145/3385412.3386029.

[Kre+17] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. “The essence of higher-order concurrent separation logic”. In: *ESOP*. Vol. 10201. LNCS. 2017. 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: *PACMPL* 2.ICFP, Article 77 (2018). DOI: 10.1145/3236772.

[Kre13] Robbert Krebbers. “Aliasing restrictions of C11 formalized in Coq”. In: *CPP*. 2013. DOI: 10.1007/978-3-319-03545-1_4.

[Kre15] Robbert Krebbers. “The C standard formalized in Coq”. PhD thesis. Radboud University, 2015.

[Kri+12] Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. “Superficially substructural types”. In: *ICFP*. 2012. DOI: 10.1145/2364527.2364536.

[Kri19] Siddharth Krishna. “Compositional abstractions for verifying concurrent data structures”. PhD thesis. New York University, Sept. 2019.

[Kro+20] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. "Aneris: A mechanised logic for modular reasoning about distributed systems". In: *ESOP*. Vol. 12075. LNCS. Springer, 2020. DOI: `10.1007/978-3-030-44914-8_13`.

[Kro18] Morten Krogh-Jespersen. "Towards modular reasoning for stateful and concurrent programs". PhD thesis. Aarhus University, Sept. 2018.

[KS07] Oleg Kiselyov and Chung-chieh Shan. "Lightweight static capabilities". In: (2007). Proceedings of the Programming Languages meets Program Verification (PLPV 2006). DOI: `10.1016/j.entcs.2006.10.039`.

[KSB17] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. "A relational model of types-and-effects in higher-order concurrent separation logic". In: *POPL*. 2017. DOI: `10.1145/3093333.3009877`.

[KSW18] Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. "Go with the flow: Compositional abstractions for concurrent data structures". In: *PACMPL* 2.POPL, Article 37 (2018). DOI: `10.1145/3158125`.

[KSW20] Siddharth Krishna, Alexander J. Summers, and Thomas Wies. "Local reasoning for global graph properties". In: *ESOP*. 2020. DOI: `10.1007/978-3-030-44914-8_12`.

[KTB17] Robbert Krebbers, Amin Timany, and Lars Birkedal. "Interactive proofs in higher-order concurrent separation logic". In: *POPL*. 2017. DOI: `10.1145/3093333.3009855`.

[LAL18] Marcus Lindner, Jorge Aparicius, and Per Lindgren. "No panic! Verification of Rust programs by symbolic execution". In: *INDIN*. IEEE, 2018. DOI: `10.1109/INDIN.2018.8471992`.

[LB08] Xavier Leroy and Sandrine Blazy. "Formal verification of a C-like memory model and its uses for verifying program transformations". In: *JAR* 41.1 (2008). DOI: `10.1007/s10817-008-9099-0`.

[Lee+17] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. "Taming undefined behavior in LLVM". In: *PLDI*. 2017. DOI: `10.1145/3062341.3062343`.

[Lee+18] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. "Reconciling high-level optimizations and low-level code in LLVM". In: *PACMPL* 2.OOPSLA, Article 125 (Oct. 2018). DOI: `10.1145/3276495`.

[Ler+12] Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. "The CompCert memory model, version 2". Technical Report RR-7987. Inria, 2012. URL: `https://hal.inria.fr/hal-00703441`.

[Lev19] Ryan Levick. "Why Rust for safe systems programming". Blog post. 2019. URL: `https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming/`.

[Lie18] Joshua Liebow-Feeser. "Tracking issue for Ref/RefMut::map_split". Rust issue #51476. 2018. URL: `https://github.com/rust-lang/rust/issues/51476`.

[Mal19] Gregory Malecha. "Two monoid questions". Thread on Iris Club mailing list. 2019. URL: `https://lists.mpi-sws.org/pipermail/iris-club/2019-July/000198.html`.

[Mat12] Nicholas D. Matsakis. "Imagine never hearing the phrase 'aliasable, mutable' again". Blog post. 2012. URL: `https://smallcultfollowing.com/babysteps/blog/2012/11/18/imagine-never-hearing-the-phrase-aliasable/`.

[Mat14] Nicholas D. Matsakis. "Rust RFC: Stronger guarantees for mutable borrows". Blog post. 2014. URL: `https://smallcultfollowing.com/babysteps/blog/2014/02/25/rust-rfc-stronger-guarantees-for-mutable-borrows/`.

[Mat16a] Nicholas D. Matsakis. "Introducing MIR". Blog post. 2016. URL: `https://blog.rust-lang.org/2016/04/19/MIR.html`.

[Mat16b] Nicholas D. Matsakis. "Non-lexical lifetimes: Introduction". Blog post. 2016. URL: http://smallcultfollowing.com/babysteps/blog/2016/04/27/non-lexical-lifetimes-introduction/.

[Mat17] Nicholas D. Matsakis. "Nested method calls via two-phase borrowing". Blog post. 2017. URL: https://smallcultfollowing.com/babysteps/blog/2017/03/01/nested-method-calls-via-two-phase-borrowing/.

[Mat18] Nicholas D. Matsakis. "An alias-based formulation of the borrow checker". Blog post. 2018. URL: https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/.

[Mem+19] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. "Exploring C semantics and pointer provenance". In: *PACMPL* 3.POPL, Article 67 (2019). DOI: 10.1145/3290380.

[Mil78] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3 (1978). DOI: 10.1016/0022-0000(78)90014-4.

[MJP19] Glen Mével, Jacques-Henri Jourdan, and François Pottier. "Time credits and time receipts in Iris". In: *ESOP*. Vol. 11423. LNCS. Springer, 2019. DOI: 10.1007/978-3-030-17184-1_1.

[MS16] Kayvan Memarian and Peter Sewell. "N2014: What is C in practice? (Cerberus survey v2): Analysis of responses". ISO SC22 WG14 N2014. Mar. 2016. URL: http://www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html.

[MSS16] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A verification infrastructure for permission-based reasoning". In: *VMCAI*. Vol. 9583. LNCS. 2016. DOI: 10.1007/978-3-662-49122-5_2.

[MTK20] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. "RustHorn: CHC-based verification for Rust programs". In: *ESOP*. 2020. DOI: 10.1007/978-3-030-44914-8_18.

[Nan+08] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. "Ynot: Dependent types for imperative programs". In: *ICFP*. 2008. DOI: 10.1145/1411204.1411237.

[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. 2014. DOI: 10.1007/978-3-642-54833-8_16.

[NI03] Thi Viet Nga Nguyen and François Irigoin. "Alias verification for Fortran code optimization". In: *J. UCS* 9.3 (2003). DOI: 10.3217/jucs-009-03-0270.

[Nor98] Michael Norrish. "C formalised in HOL". PhD thesis. University of Cambridge, 1998.

[OCo+16] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. "Refinement through restraint: Bringing down the cost of verification". In: *ICFP*. 2016. DOI: 10.1145/2951913.2951940.

[OHe07] Peter W. O'Hearn. "Resources, concurrency, and local reasoning". In: *TCS* 375.1 (2007). DOI: 10.1016/j.tcs.2006.12.035.

[OP99] Peter W. O'Hearn and David J. Pym. "The logic of bunched implications". In: *Bulletin of Symbolic Logic* 5.2 (June 1999). DOI: 10.2307/421090.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local reasoning about programs that alter data structures". In: *CSL*. Vol. 2142. LNCS. 2001. DOI: 10.1007/3-540-44802-0_1.

[Par10] Matthew J. Parkinson. "The next 700 separation logics - (Invited paper)". In: *VSTTE*. Vol. 6217. LNCS. 2010. DOI: 10.1007/978-3-642-15057-9_12.

[PBC06] Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. "Variables as resource in Hoare logics". In: *LICS*. IEEE, 2006. DOI: 10.1109/LICS.2006.52.

[Pie+19] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. “Software foundations (volume 2): Programming language foundations”. 2019. URL: https://softwarefoundations.cis.upenn.edu/plf-current/index.html.

[Plo73] Gordon Plotkin. “Lambda-definability and logical relations”. Unpublished manuscript. 1973. URL: http://homepages.inf.ed.ac.uk/gdp/publications/logical_relations_1973.pdf.

[Pop18] Nikita Popov. “Loop unrolling incorrectly duplicates noalias metadata”. LLVM issue #39282. 2018. URL: https://bugs.llvm.org/show_bug.cgi?id=39282.

[Pot13] François Pottier. “Syntactic soundness proof of a type-and-capability system with hidden state”. In: *JFP* 23.1 (2013). DOI: 10.1017/S0956796812000366.

[Ree15] Eric Reed. “Patina: A formalization of the Rust programming language”. MA thesis. University of Washington, USA, 2015.

[Reg17] John Regehr. “Undefined behavior in 2017”. Blog post. 2017. URL: https://blog.regehr.org/archives/1520.

[Rey02] John C. Reynolds. “Separation logic: A logic for shared mutable data structures”. In: *LICS*. 2002. DOI: 10.1109/LICS.2002.1029817.

[RL12] Valentin Robert and Xavier Leroy. “A formally-verified alias analysis”. In: *CPP*. 2012. DOI: 10.1007/978-3-642-35308-6_5.

[RS10] Grigore Rosu and Traian-Florin Serbanuta. “An overview of the K semantic framework”. In: *Journal of Logic and Algebraic Programming* 79.6 (2010). DOI: 10.1016/j.jlap.2010.03.012.

[Rus20] The Rust teams. “Rust Programming Language”. 2020. URL: https://www.rust-lang.org/.

[RVG15] Azalea Raad, Jules Villard, and Philippa Gardner. “CoLoSL: Concurrent local subjective logic”. In: *ESOP*. 2015. DOI: 10.1007/978-3-662-46669-8_29.

[Sam+20] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. “The high-level benefits of low-level sandboxing”. In: *PACMPL* 4.POPL, Article 32 (2020). DOI: 10.1145/3371100.

[SB14] Kasper Svendsen and Lars Birkedal. “Impredicative concurrent abstract predicates”. In: *ESOP*. Vol. 8410. LNCS. 2014. 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. 2013. DOI: 10.1007/978-3-642-37036-6_11.

[Sew19] Peter Sewell. “Type-changing and effective type”. Thread on C Programming Language Memory Object Model mailing list. 2019. URL: https://lists.cam.ac.uk/pipermail/cl-c-memory-object-model/2019-May/msg00093.html.

[SGD17] David Swasey, Deepak Garg, and Derek Dreyer. “Robust and compositional verification of object capability patterns”. In: *PACMPL* 1.OOPSLA, Article 89 (2017). DOI: 10.1145/3133913.

[SM17] Josh Stone and Nicholas D. Matsakis. “The Rayon library”. Rust crate. 2017. URL: https://crates.io/crates/rayon.

[SNB15] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. “Mechanized verification of fine-grained concurrent programs”. In: *PLDI*. 2015. DOI: 10.1145/2737924.2737964. URL: http://doi.acm.org/10.1145/2737924.2737964.

[SSB16] Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. “Transfinite step-indexing: Decoupling concrete and logical steps”. In: *ESOP*. Vol. 9632. LNCS. Springer, 2016. DOI: 10.1007/978-3-662-49498-1_28.

[Str12] Bjarne Stroustrup. “Foundations of C++”. In: *ESOP*. 2012. DOI: 10.1007/978-3-642-28869-2_1.

[Str94] Bjarne Stroustrup. “The design and evolution of C++”. New York, NY, USA: ACM Press/ Addison-Wesley Publishing Co., 1994. ISBN: 0-201-54330-3.

[Swa+06] Nikhil Swamy, Michael W. Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. “Safe manual memory management in Cyclone”. In: *Science of Computer Programming* 62.2 (2006). DOI: 10.1016/j.scico.2006.02.003.

[Swa+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, *et al.* “Dependent types and multi-monadic effects in F*”. In: *POPL*. 2016. DOI: 10.1145/2837614.2837655.

[Tai67] W. W. Tait. “Intensional interpretations of functionals of finite type I”. In: *Journal of Symbolic Logic* 32.2 (1967). DOI: 10.2307/2271658.

[Tas19] Joseph Tassarotti. “Verifying concurrent randomized algorithms”. PhD thesis. Carnegie Mellon University, Jan. 2019.

[TB19] Amin Timany and Lars Birkedal. “Mechanized relational verification of concurrent programs with continuations”. In: *PACMPL* 3.ICFP, Article 105 (2019). DOI: 10.1145/3341709.

[TDB13] Aaron Turon, Derek Dreyer, and Lars Birkedal. “Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency”. In: *ICFP*. 2013. DOI: 10.1145/2500365.2500600.

[TH19] Joseph Tassarotti and Robert Harper. “A separation logic for concurrent randomized programs”. In: *PACMPL* 3.POPL, Article 64 (2019). DOI: 10.1145/3290377.

[The20] The Chromium project. “Chromium security: Memory safety”. 2020. URL: https://www.chromium.org/Home/chromium-security/memory-safety.

[Tho19] Gavin Thomas. “A proactive approach to more secure code”. Blog post. 2019. URL: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/.

[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: *PACMPL* 2.POPL, Article 64 (2018). DOI: 10.1145/3158152.

[Tim18] Amin Timany. “Contributions in programming languages theory: Logical relations and type theory”. PhD thesis. KU Leuven, May 2018.

[TJH17] Joseph Tassarotti, Ralf Jung, and Robert Harper. “A higher-order logic for concurrent termination-preserving refinement”. In: *ESOP*. Vol. 10201. LNCS. 2017. DOI: 10.1007/978-3-662-54434-1_34.

[TP11] Jesse A. Tov and Riccardo Pucella. “Practical affine types”. In: *POPL*. 2011. DOI: 10.1145/1926385.1926436.

[TPT15] John Toman, Stuart Pernsteiner, and Emina Torlak. “CRUST: A bounded verifier for Rust”. In: *ASE*. 2015. DOI: 10.1109/ASE.2015.77.

[Tur+13] Aaron Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. “Logical relations for fine-grained concurrency”. In: *POPL*. 2013. DOI: 10.1145/2429069.2429111.

[TVD14] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. “GPS: Navigating weak memory with ghosts, protocols, and separation”. In: *OOPSLA*. 2014. DOI: 10.1145/2660193.2660243.

[Ull16] Sebastian Andreas Ullrich. “Simple verification of rust programs via functional purification”. MA thesis. Karlsruher Institut für Technologie (KIT), Dec. 2016.

[Vaf11] Viktor Vafeiadis. “Concurrent separation logic and operational semantics”. In: *MFPS*. 2011. DOI: `10.1016/j.entcs.2011.09.029`.

[VP07] Viktor Vafeiadis and Matthew J. Parkinson. “A marriage of rely/guarantee and separation logic”. In: *CONCUR*. Vol. 4703. LNCS. 2007. DOI: `10.1007/978-3-540-74407-8_18`.

[Wad90] Philip Wadler. “Linear types can change the world!” In: *Programming Concepts and Methods*. 1990. URL: `https://homepages.inf.ed.ac.uk/wadler/papers/linear/linear.dvi`.

[Wan+12] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. “Undefined behavior: What happened to my code?” In: *APSYS*. 2012. DOI: `10.1145/2349896.2349905`.

[WF94] Andrew K Wright and Matthias Felleisen. “A syntactic approach to type soundness”. In: *Information and computation* 115.1 (1994). DOI: `10.1006/inco.1994.1093`.

[wit18] withoutboats. “Tracking issue for pin APIs (RFC 2349)”. Rust issue #49150. 2018. URL: `https://github.com/rust-lang/rust/issues/49150`.

[WL95] Robert P. Wilson and Monica S. Lam. “Efficient context-sensitive pointer analysis for C programs”. In: *PLDI*. 1995. DOI: `10.1145/207110.207111`.