Non-Step-Indexed Separation Logic with Invariants and Rust-Style Borrows

（不変条件と Rust 流の借用を扱える 非 Step-Indexed な分離論理）

by

Yusuke Matsushita
松下 祐介

A Doctoral Thesis
博士論文

Submitted to
the Graduate School of the University of Tokyo
on December 6, 2023
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Information Science and Technology
in Computer Science

Thesis Supervisor: Naoki Kobayashi　小林 直樹
Professor of Computer Science

ABSTRACT

Today, computer software is playing a remarkable role, and improving its quality and robustness has become vital for the development and safety of society. For this purpose, software science and engineering have developed mathematically rigorous methods, or formal methods. Particularly fundamental is general-purpose *program logic* that can precisely and flexibly *verify* programs, i.e., reason about the behavior of software described by programs.

A hard problem in program verification is sound and scalable reasoning about *mutable state*, such as data stored in heap memory. The basic tool for this is *ownership*, the exclusive right to update mutable state. For each fragment of mutable state, only the computational agent with its ownership is allowed to update it, so that the latest information can be accurately captured. *Separation logic* is a logic that can reason about ownership and verify programs with mutable state in a modular way, which has been actively and fruitfully studied in this century. Modern separation logics, going beyond naive ownership, support advanced mechanisms for reasoning about mutable state under *propositional sharing*, i.e., sharing among multiple computational agents based on contracts described by separation logic propositions, such as *shared invariants* from Iris by Jung et al. (2015) and *Rust-style borrows* from RustBelt's lifetime logic by Jung et al. (2018). However, to soundly support such mechanisms for propositional sharing, existing approaches weaken the access to the shared content by the *later modality*, which is ill-behaved, and employ *step-indexed* program logic. This has fundamental problems especially in verifying *liveness properties* such as program termination.

In this dissertation, we propose a novel, general framework, *Nola*, for achieving *non-step-indexed separation logic* that supports advanced mechanisms for propositional sharing of mutable state such as invariants and borrows. Using our framework, one can verify programs involving shared mutable state in non-step-indexed program logic that can guarantee liveness properties, being free from the later modality. Our key idea is to *isolate* the *syntactic* data type for the separation logic propositions to be shared from its *semantic* interpretation. Our proof rules for propositional sharing are generalized over the choice of the proposition data type and its interpretation, which one can freely instantiate for one's purpose. The condition that the interpretation of the syntax is well-defined restricts the class of propositions that a mechanism for propositional sharing can store, which naturally avoids the paradox of later-free invariants. Still, the framework allows one to freely nest the logical connectives for propositional sharing to reason about nested shared variable references. Moreover, we have discovered a novel, general technique for *semantic alteration* of the content propositions of the logic connectives for propositional sharing. We have fully mechanized our framework on top of the Iris separation logic framework in the Coq Proof Assistant, making it easy to combine it with existing developments. We have demonstrated the expressivity and verification power of our framework with non-trivial examples, including verification of strong normalization of functional programs under a hierarchical type system that supports higher-order reference types, and construction of a general later-free mechanism that refines the approach of RustHornBelt by Matsushita et al. (2022) to functional verification about borrows with RustHorn-style prophecies.

論文要旨

今日、コンピュータ・ソフトウェアは目覚ましい活躍をしており、その品質や堅牢性を高めることは、社会の発展と安全のために不可欠となっている。このために、ソフトウェア科学・工学は、数学的に厳格な諸手法、形式手法を発展させてきた。特に基盤となるのが、正確かつ柔軟にプログラムを**検証**できる、すなわちプログラムで記述されたソフトウェアの動作について推論できる、汎用的な**プログラム論理**である。

プログラム検証における難問が、ヒープメモリに格納されたデータに代表される**可変状態**について、健全かつスケーラブルな推論をすることである。そのための基本的な道具が、可変状態を更新するための独占的な権利、**所有権**である。可変状態の各断片について、その所有権をもつ計算主体のみに更新を許すことで、最新の情報が正確に把握できる。**分離論理**は所有権について推論し、可変状態を扱うプログラムをモジュラーな形で検証できる論理であり、今世紀において活発に、実りの多い形で研究されてきた。現代的な分離論理は、素朴な所有権にとどまらず、**命題的共有**、すなわち分離論理命題で記述された契約に基づく複数の計算主体間の共有のもとにある可変状態について検証するための高度な仕組みである、Jung ら (2015) の Iris による**共有不変条件**や Jung ら (2018) の RustBelt のライフタイム論理による *Rust* **流の借用**などを提供している。しかし、そうした命題的共有のための仕組みを健全に扱うために、既存手法では、共有された中身へのアクセスを振る舞いの悪い *later* **様相**によって弱め、*step-indexed* なプログラム論理を用いる。これは特に、プログラムの停止性などの *liveness* **性質**を検証するうえで、根本的な問題を抱えている。

この学位論文では、不変条件や借用といった可変状態の**命題的共有**のための高度な仕組みを扱える**非** *step-indexed* **な分離論理**を実現するための、新しい汎用的なフレームワーク、*Nola* を提案する。私たちのフレームワークを使えば、later 様相から解放されて、liveness 性質を保証できる非 step-indexed なプログラム論理のうえで、共有された可変状態を扱うプログラムを検証することができる。私たちの鍵となるアイデアは、共有される分離論理命題を表す**構文**的なデータ型とその**意味論**的な解釈を**切り離す**ことである。私たちが提供する命題的共有のための証明規則は、命題のデータ型とその解釈の選択について一般化されており、これらは用途に応じて自由に具体化することができる。構文の解釈が有効な形で定義されるという条件は、命題的共有の機構が格納できる命題のクラスを制限しており、これにより later 様相のない不変条件に関するパラドクスを自然に回避している。その一方で、このフレームワークでは、命題的共有のための論理結合子を自由にネストして、特にネストした共有可変参照について推論することができる。さらに私たちは、命題的共有のための論理結合子の中身の命題を**意味論的に変える**ための、新しい汎用的な手法を発見した。私たちはこのフレームワークを Coq 定理証明支援系で Iris 分離論理フレームワークの上で完全に機械化し、既存のコード資産と容易に組み合わせられるようにした。私たちはこのフレームワークの表現力および検証能力を、高階参照型を扱える階層的な型システムのもとでの関数型プログラムの強正規化性の検証、および松下ら (2022) の RustHornBelt の手法を洗練させた、借用に関する RustHorn 流の預言による機能検証のための later 様相のない一般的機構の構成を含む、非自明な例で実証した。

Acknowledgments

First of all, I would like to express my unreserved gratitude to my supervisor, Professor Naoki Kobayashi. For more than five years since I was an undergraduate, he has perseveringly and significantly nurtured my skills as a researcher, especially in presentation and communication, while generously giving me the freedom, time and environment to explore my own research interests. He is one of the world's leading software scientists, and his keen insight and enthusiasm for academic activities have always inspired me. I feel very fortunate to have him as my supervisor.

I would like to express my sincere gratitude to Professor Takeshi Tsukada. He is my only collaborator on the research project Nola, the main body of this dissertation, and he has generously given his time and effort to help me push this project to a higher level. He is also my close collaborator on my first research project, RustHorn. He carefully guided me especially on how to write papers better, and also taught me the joy of theoretical research. He often complimented me on my sense of research, which greatly encouraged me.

I thank my collaborators in the RustHornBelt project, Professor Derek Dreyer, Professor Jacques-Henri Jourdan, and Dr. Xavier Denis. Although the COVID pandemic unfortunately made it a remote collaboration, the experience of working with them greatly broadened my horizons and enhanced my skills to advance my research. In particular, Derek has an extraordinary ability to capture the essence of things and communicate it to others, which I was fascinated by and learned a lot from. I am also grateful to Professor Ralf Jung, one of the world's leading pioneers in theoretical research on Rust, who assisted and inspired me. I also appreciate the recent projects Creusot, CreuSAT and RusSOL, respectively led by Dr. Xavier Denis, Mr. Sarek Høverstad Skotåm and Mr. Jonáš Fiala, which provided intriguing demonstrations of the effectiveness of RustHorn's approach. Moreover, I thank the members of the Kobayashi Lab, especially Mrs. Yukiko Kimura, Dr. Ken Sakayori, Dr. Ryosuke Sato, Dr. Minchao Wu, Mr. Hiroyuki Katsura, and Mr. Takashi Nakayama. I had a great time interacting with them and my graduate life got more colorful thanks to them.

I express my special thanks to my piano teacher, Mrs. Sonoko Hayashi. For about six years, she energetically cultivated my understanding of classical music and showed me how to convey the music with the body in piano performance. This has nurtured my intellectual sensibilities beyond the realm of music, I believe. Also, I am grateful to the Piano Society of the University of Tokyo, to which I belonged for about nine years since I entered the university. I owe a lot to this club for my continuous engagement in piano playing, and I had a memorable time with the people there. In addition, I thank my ballroom dancing friends. Dancing and talking with them has helped keep my mind and body healthy.

Finally, I would like to express my utmost gratitude to my parents and grandparents. They have always been there for me and supported me with great love. They raised me and taught me the most important things in life. They also understood and supported my decision to pursue a doctoral degree in the field of computer science. It is surely thanks to them that I am what I am today.

Contents

Chapter 1

Introduction

First, Section 1.1 presents the general background of this dissertation. Next, Section 1.2 presents the basic approach to *mutable state*, *ownership* and *separation logic*. After that, Section 1.3 presents the mechanisms for *propositional sharing* that modern separation logics support to reason about *shared mutable state*. Section 1.4 discusses the problem of the *later modality* and *step-indexing* that existing approaches to propositional sharing suffer from. Finally, Section 1.5 presents our solution to the problem, the *Nola* framework. See § 1.5.3 for how this dissertation is organized after this chapter.

1.1 General Background

知者樂水
The wise enjoy water

Confucius, the *Analects*

Today, computer software is playing a remarkable role in our lives. In the cities, traffic light systems are keeping traffic safe, credit cards are making payments only as intended, MRI machines in hospitals are working safely, airplanes are flying accident-free, and so on. With a laptop or smartphone, we are sending messages to friends, searching directions from home to new places, and filing taxes online, all with privacy protected. In the last few decades, human life has come to rely increasingly on software, and this momentum is not about to stop. It has become vital to *improve the quality and robustness of software* for the development and safety of society.

Software science and engineering are devoted to this purpose. Compared to the oldest days when programmers directly typed machine language on punch cards, modern software development is much more sophisticated, thanks to the achievements of these fields. Nowadays, programmers can develop complex large-scale software in programming languages with rich abstractions, can ascertain the correctness of programs by type checking and unit testing, can manage the time evolution of huge code bases by a version control system, and so forth.

Of special scientific interest are mathematically rigorous techniques for solving problems in software development, or *formal methods*. The most widely used formal methods in modern times are *type systems*. Types are labels attached to data, describing its kind and attributes. Automatic checking on the types runs statically, which prevents many bugs before the programs are executed. Beyond simple types like telling integers from strings, modern programmers have come to use richer and stronger types, such as those used in TypeScript and Rust. From a general perspective, programmers are increasingly turning to formal methods as software grows in size and complexity far beyond the grasp of a single person.

Particularly fundamental in this direction is the exploration of general-purpose *program logic* that can precisely and flexibly *verify* programs, i.e., reason about the behavior of software described by programs.

This dissertation aims to build expressive program logic for reasoning about *mutable state* with the power to verify *liveness properties*, including *program termination* and *total correctness*.

1.1.1 Hoare Logic

Central among program logics is *Hoare logic*, which was invented over a half-century ago by Hoare (1969) after the idea of Floyd (1967). Hoare logic and its variants have been actively studied for a long time and have served as the scientific foundation for more downstream, practical formal methods, such as type systems and automated verification platforms.

A key assertion of Hoare logic is the *partial Hoare triple* $\{\,\phi\,\}\; e\; \{\,\psi\,\} \in \mathit{Prop}$, where $e \in \mathit{Expr}$ is the expression (or program fragment) of interest, $\phi \in \mathit{Prop}$ is called the *precondition* and $\psi \colon \mathit{Val} \to \mathit{Prop}$ is called the *postcondition*. The partial Hoare triple $\{\,\phi\,\}\; e\; \{\,\psi\,\}$ says that the expression e can be safely executed under the premise ϕ and whenever e terminates with a return value $v \in \mathit{Val}$, the condition $\psi\, v$ is satisfied.

Example: Sum of Odd Numbers As a warmup example, let us think of the following recursive function:

$$\mathsf{fun}\ \mathsf{oddsum}(n)\ \big\{\ \mathsf{if}\ \ n = 0\ \ \mathsf{then}\ \ 0\ \ \mathsf{else}\ \ 2 \times n\ -\ 1\ +\ \mathsf{oddsum}(n-1)\ \big\}. \tag{1.1}$$

Roughly, the function $\mathsf{oddsum}(n)$ adds up the odd numbers from 1 to $2n-1$. You may expect that its return value is n^2.

Indeed, we can verify the following partial Hoare triple assertion in Hoare logic:

$$\forall n.\ \big\{\ n \in \mathbb{Z}\ \big\}\ \ \mathsf{oddsum}(n)\ \ \big\{\ \lambda v.\ v = n^2\ \big\}. \tag{1.2}$$

This assertion means that, for any integer $n \in \mathbb{Z}$, whenever the function call $\mathsf{oddsum}(n)$ terminates, its return value is equal to n^2. The proof goes by finding the *invariant* (post)condition[1] $\lambda v.\ v = n^2$ on the execution of $\mathsf{oddsum}(n)$ for any $n \in \mathbb{Z}$. The proof that this is an invariant goes as follows. For the case $n = 0$ we are done. Otherwise, we can assume that the recursive call $\mathsf{oddsum}(n-1)$ returns $(n-1)^2$ by the invariant, and thus the return value of $\mathsf{oddsum}(n)$ is $(2n-1) + (n-1)^2 = n^2$, which satisfies the invariant. Technically, this kind of reasoning where we can use the proof goal as an assumption is called *coinduction*.

Semantics of Hoare Logic Let us also see the *semantics* of Hoare logic for a better understanding. The partial Hoare triple $\{\,\phi\,\}\; e\; \{\,\psi\,\}$ is modeled as an entailment:

$$\{\,\phi\,\}\; e\; \{\,\psi\,\} \quad \triangleq \quad \phi \to \mathsf{pwp}\ e\ \{\,\psi\,\} \tag{1.3}$$

The consequent of the entailment is the *(partial) weakest precondition* $\mathsf{pwp}\ e\ \{\,\psi\,\}$ for the expression e and the postcondition ψ, defined as follows:

$$\begin{aligned}\mathsf{pwp}\ e\ \{\,\psi\,\} \ \ \triangleq_{\nu}\ \ & \big(\exists v \in \mathit{Val}\ \text{s.t.}\ \ e = v.\ \ \psi\, v\,\big)\ \lor \\ & \big(\,(\exists e'.\ e \hookrightarrow e')\ \land\ \forall e' \hookleftarrow e.\ \mathsf{pwp}\ e'\ \{\,\psi\,\}\,\big)\end{aligned} \tag{1.4}$$

[1] The notion of the invariant here is related to but quite different from the *shared invariant* mechanism for shared mutable state we explore in this dissertation (§ 1.3.1).

We write $\hookrightarrow$ for the *reduction* relation on expressions (and $\hookleftarrow$ for its inverse). The first disjunct says that the expression has already been reduced to a value $v \in \mathit{Val}$ and satisfies the postcondition ψ. The second disjunct says that the expression is reducible (i.e., the execution is not stuck) and, for any expression e' that e can reduce to, the weakest precondition pwp $e' \{\psi\}$ *coinductively* holds. Here, the definition uses the *greatest fixed point* (marked by $\triangleq_\nu$) to model the *coinductive reasoning* of the partial Hoare triple.

1.1.2 Partial vs. Total Correctness, or Safety vs. Liveness

Problem: Non-Termination A careful reader may have wondered about the case $n < 0$. For the case, the execution of $\mathsf{oddsum}(n)$ never terminates, with the argument n decremented an infinite number of times.

Assertions like (1.2) hold because the *partial* Hoare triple $\{\phi\}\, e\, \{\psi\}$ considered here *does not guarantee termination*. It only guarantees what holds *if* the execution terminates. This type of guarantee is called the *partial correctness*.

For an extreme example, consider the following variant of oddsum:

$$\mathsf{fun}\ \mathsf{oddsum}'(n)\ \big\{\ 2 \times n\ -\ 1\ +\ \mathsf{oddsum}'(n-1)\ \big\}$$

This function oddsum′ is very similar to oddsum (1.1) but omits the part returning 0 for the case $n = 0$. As a result, the function *never terminates* for any n whatsoever. Nonetheless, we can still prove the following for oddsum′ just like (1.2):

$$\forall n.\ \big\{\ n \in \mathbb{Z}\ \big\}\ \mathsf{oddsum}'(n)\ \big\{\ \lambda v.\ v = n^2\ \big\}.$$

The proof goes in a similar way to (1.2), by establishing the invariant $\lambda v.\ v = n^2$ for $\mathsf{oddsum}'(n)$. As seen in the examples above, the partial Hoare triple *cannot prove the absence of bugs of non-termination*, which can be problematic in practice.

Total Hoare Triple The good news is that Hoare logic can also support the *total* Hoare triple $[\phi]\ e\ [\psi]$ that guarantees the total correctness. The *total correctness* means the conjunction of the termination and the partial correctness. The wording partial vs. total here can be understood by an analogy with partial vs. total functions. The total Hoare triple $[\phi]\ e\ [\psi]$ says that, under the precondition ϕ, the execution of e always *terminates* with a return value v satisfying the postcondition $\psi\ v$.

Now as we expect, neither $\forall n.\ [n \in \mathbb{Z}]\ \mathsf{oddsum}(n)\ [\lambda v.\ v = n^2]$ nor $\forall n.\ [n \in \mathbb{Z}]\ \mathsf{oddsum}'(n)\ [\lambda v.\ v = n^2]$ holds, while we can successfully prove the following in Hoare logic:

$$\forall n.\ \big[\ n \in \mathbb{N}\ \big]\ \mathsf{oddsum}(n)\ \big[\ \lambda v.\ v = n^2\ \big]. \tag{1.5}$$

This assertion means that, for any natural number (i.e., non-negative integer) $n \in \mathbb{N}$, the function call $\mathsf{oddsum}(n)$ always *terminates* with a return value equal to n^2. The proof of (1.5) simply goes by mathematical *induction* over $n \in \mathbb{N}$. Technically, the *coinductive* reasoning we used for the partial Hoare triple does not work for the total Hoare triple, and so we employ *inductive* reasoning here. As seen in this example, the total Hoare triple gives stronger guarantees and thus is less likely to overlook bugs than the partial Hoare triple.

Also, if we introduce a primitive expression ndnat that returns a *non-deterministic natural number*, we can prove the following in Hoare logic:

$$\big[\ \top\ \big]\ \mathsf{oddsum}(\mathsf{ndnat})\ \big[\ \lambda v.\ \exists n \in \mathbb{N}.\ v = n^2\ \big]$$

The expression $\mathsf{oddsum}(\mathsf{ndnat})$ calls the function oddsum with a non-deterministic natural number as the argument. This total Hoare triple asserts total correctness, saying that execution of oddsum(ndnat) *always terminates* with a square number as the return value, *regardless of the non-deterministic choice* by ndnat. Remarkably, the number of program steps that the expression oddsum(ndnat) takes is *unbounded*, because the primitive ndnat can return an arbitrarily large natural number.

Semantically, the *total* Hoare triple $\left[\phi\right] e \left[\psi\right]$ is modeled as an entailment $\phi \to \mathsf{twp}\ e\left[\psi\right]$ to the *total weakest precondition* $\mathsf{twp}\ e\left[\psi\right]$, which is defined as follows:

$$\begin{aligned}\mathsf{twp}\ e\left[\psi\right] \ \triangleq_\mu\ & \left(\exists v \in \mathit{Val}\ \text{s.t.}\ e = v.\ \psi\, v\right) \lor \\ & \left((\exists e'.\ e \hookrightarrow e') \land \forall e' \hookleftarrow e.\ \mathsf{twp}\ e'\left[\psi\right]\right)\end{aligned} \tag{1.6}$$

This is quite similar to the definition of the partial weakest precondition $\mathsf{pwp}\ e\left\{\psi\right\}$ (1.4). The only difference is that the definition uses the *least fixed point* (marked by $\triangleq_\mu$) to model the *inductive reasoning* of the total Hoare triple.

Safety vs. Liveness Properties From a more general perspective, partial correctness is classified as a *safety property* and total correctness is classified as a *liveness property*.

Formally, they are defined as follows.

- A safety property only refers to the absence of ‘bad’ *finite* behaviors, i.e., behaviors observed from a finite number of execution steps.
- A liveness property can refer to the absence of ‘bad’ *infinite* behaviors, i.e., behaviors observed only from an infinite sequence of execution steps.

For a simple example, program termination is a liveness property (not safety), because it refers to the absence of any infinite execution.

More roughly speaking, a safety property just states that a ‘bad’ event will never happen in execution, whereas a liveness property states that a ‘good’ event will eventually happen. For example, program termination states that the event of termination will eventually happen in execution.

Exploring program logic that can prove liveness properties is significant, especially because bugs may be overlooked if we only consider safety properties as we saw in the examples of oddsum and oddsum′.

1.2 Ownership and Separation Logic

Διαίρει καὶ βασίλευε
Divide and rule

Philip II of Macedon

So far, we have set aside a hard problem: *mutable state*. After explaining its general difficulties (§ 1.2.1), we introduce the basic tool for tackling it, *ownership* (§ 1.2.2), and then illustrate *separation logic*, the program logic employing ownership to reason about mutable state (§ 1.2.3).

1.2.1 Hard Problem: Mutable State

A hard problem in program verification is sound and scalable reasoning about *mutable state*, i.e., state that can be updated as the program is executed.

A prime example of mutable state is mutable data or objects stored in *heap memory*. Managing and manipulating heap memory in a fine-grained way is often essential for achieving tolerable time and space efficiency of computation, especially in *system software*, or low-level software that other software relies on.

But such mutable state can drastically increase the number of possible situations that we naively have to take into account, making it difficult to reasoning about the behaviors of the programs. In fact, programmers often make a mistake in reasoning about mutable state, especially around heap memory, which sometimes leads to serious bugs and security problems.

Example: Dangling Pointer by Memory Reallocation To take a glance at the hardship involving mutable state, consider the following C++ program:

```
vector<int> v {0, 1, 2};  int* p = &v[0];
v.push_back(3);  *p = 7;  printf("%d\n", v[0]); // 7 or 0?
```

Code 1.1: Dangling Pointer by Memory Reallocation

First, we create a *vector* (or *heap-allocated*, growable array) of integers `v` with the initial elements `0`, `1`, and `2`.[2] Second, we create the *pointer* `p` that has the memory location (a.k.a. address) of the first element of the vector `v`. Third, we push a new element `3` to the end of the vector `v`. Fourth, we write a new value `7` to the location the pointer `p` points to. Finally, we print the first element of the vector `v`.

One may well expect that this program prints the new value `7` by the effect of the memory write `*p = 7`. But in reality, *the program prints the old value* `0`. Why? This is because of *memory reallocation* performed behind the scenes.

When the vector `v` is initialized, it allocates the memory block of three memory cells for its initial elements. The pointer `p` points to the first element of this memory block. But when the new element is pushed, the vector `v` has to *reallocate* the memory block to a new larger one, moving the elements to the new memory block. As a result, the pointer `p` becomes a *dangling pointer*, i.e., a pointer that does not point to a valid object, because the memory block has been *deallocated* (or *freed*) by the vector `v`. The memory write `*p = 7` does not have a valid effect on the vector `v`, making the program print `0`. What is worse, this even causes *undefined behavior*, because it is a memory write to a dangling pointer. This is also an example of *use-after-free*, i.e., manipulation of an object that has been deallocated.

Memory Unsafety in the Real World In fact, many of the software vulnerabilities in the real world are due to *memory unsafety*. Google's Project Zero team reported that about 70% of the zero-day exploit cases that they collected from public sources had memory-corruption issues as the root cause (Project Zero, 2019). Also, Google's Chromium reported that about 70% of their high-severity security bugs are memory unsafety problems, with about 35% being caused by use-after-free (Chromium Projects, 2023), and Microsoft similarly reported that around 70% of their security bugs are due to memory safety issues (Microsoft Security Response Center, 2019).[3] The difficulties in mutable state are a real threat to the security of software around us.

1.2.2 Ownership

A basic known remedy to the difficulties in mutable state is the idea of *ownership*, whose importance has been known for a long time (Dijkstra, 1965; Hogg et al., 1992). The ownership of mutable state is the *exclusive* right to *mutate* (or update) the state, which cannot be shared.

[2] Here we use the initializer list syntax introduced in C++11.

[3] It is an interesting coincidence that all these three different sources show a similar trend that about 70% of the security issues are due to memory unsafety.

The principle of ownership is as follows: for each mutable state, there may be multiple computational agents that can get access to the state, but only *one* of them can have the ownership of the state at a time and thus can mutate the state. Notably, the computational agent with the ownership can safely have the *up-to-date knowledge* about the state because all the mutations on the state are performed by the agent itself. On the other hand, any other agent gives up such up-to-date knowledge because it does not know what mutations are performed by the agent with the ownership.

Generally speaking, ownership helps one to safely manage mutable state in a modular, scalable way.

Managing Ownership Ownership can be managed both dynamically and statically.

A basic, familiar example of *dynamic* ownership management is the *mutex* (short for mutual exclusion) used for concurrent algorithms, proposed by Dijkstra (1965). At each time in program execution, only one computational agent (or thread) can dynamically acquire the *lock* of the mutex to get the *ownership* of the shared mutable state associated with the mutex. The mutex prevents the shared mutable state from being mutated by multiple threads at the same time, which makes concurrent programming safer.

An important approach to *static* ownership management is *ownership type systems*, or type systems that manage ownership. Roughly speaking, an ownership type system handles *static 'locks'* on objects at compile time to ensure memory and thread safety (no data race, no use-after-free, etc.) without runtime overhead. Various forms of ownership type systems have been studied for decades (Wadler, 1990; Tofte and Talpin, 1997; Gay and Aiken, 1998; Clarke et al., 1998; Grossman et al., 2002; Fluet et al., 2006; Clarke et al., 2013; Bernardy et al., 2018; Arvidsson et al., 2023).

Rust Of particular note is the *Rust programming language* (Rust Team, 2023; Matsakis and Klock, 2014), a modern system programming language released in 2015. While Rust supports low-level memory control like C and C++, it also uses a strong *ownership type system*, influenced by Cyclone (Grossman et al., 2002), to provide high-level guarantees of memory and thread safety.

Despite the somewhat exotic nature of Rust's ownership type system, its ability to eliminate real-world bugs has made Rust widely adopted in the software industry. For example, Google's Android has recently adopted Rust, with about 21% of the new native code of Android 13 being written in Rust, drastically reducing the number of memory safety bugs (Stoep, 2022).

Revisiting the Example in Rust To see how an ownership type system prevents memory safety bugs, we revisit the previous example Code 1.1.

We first translate the C++ code to Rust:

```
let mut v = vec![0, 1, 2];  let p = &mut v[0];
v.push(4);  *p = 7;  println!("{}", v[0]);
```

Code 1.2: Rust Version of Code 1.1

The code looks quite the same as C++.

But Rust's compiler *rejects* this code, before executing the program or even generating the machine code. It omits an error message like the following:

```
error[E0499]: cannot borrow ‘v’ as mutable more than once at a time
| let p = &mut v[0];
|               - first mutable borrow occurs here
| v.push(4);
| ^ second mutable borrow occurs here
| *p = 7;
| ------ first borrow later used here
```
Code 1.3: Error Message for Code 1.2

This is because the *static ownership analysis* (nicknamed *borrow check*) of Rust's compiler correctly *rejects* this dangerous, memory-unsafe program. Let us see how the ownership analysis goes.

When we first create the vector by `let mut v = vec![0, 1, 2]`, the object `v` has the *ownership* of the vector. When we take a pointer to the first element by `let p = &mut v[0]`, we allow the pointer (or *mutable reference*) `p` to *temporarily get the ownership* of the whole vector, which is retained until and used for the memory write `*p = 7`. Since the original owner `v` is expected to recover the ownership after `p` is deactivated, this is an example of what is called a *mutable borrow* in Rust (thus it is referred to as the 'first mutable borrow' above). On the other hand, pushing a new element to the vector by `v.push(4)` requires the *ownership* of the vector since it *mutates* the vector. Technically, functions like `push` get the ownership for mutation by a mutable borrow (and so this is referred to as the 'second mutable borrow' above). But since the mutable borrow `p` is still active, this would require two mutable borrows of the vector, or two computational agents with the ownership of the vector, which breaks the principle of ownership.

Note that Rust's ownership type system does accept many memory-safe programs. For example, consider the following variant of Code 1.2, with the third and fourth statements swapped:

```
let mut v = vec![0, 1, 2];  let p = &mut v[0];
*p = 7;  v.push(4);  println!("{}", v[0]);
```

Now this is memory-safe because the vector is not mutated between the creation of `p` and the write to `p`. Rust does accept this program. Because the mutable borrow to `p` is deactivated before the mutation by `push`, the ownership is not overlapping.

1.2.3 Separation Logic

Separation logic (O'Hearn and Pym, 1999) is a logic for reasoning about *ownership*, which can be used for verifying programs with mutable state (Ishtiaq and O'Hearn, 2001; O'Hearn et al., 2001; Reynolds, 2002).[4]

The core feature of separation logic is a logical connective called the *separating conjunction* $P * Q$, which comes from the 'multiplicative conjunction' of *linear logic* (Girard, 1987).[5] Although the logical connective is commutative and associative

$$P * Q \;=\; Q * P \qquad\qquad P * (Q * R) \;=\; (P * Q) * R,$$

it is *not idempotent*

$$P \;\neq\; P * P$$

unlike the usual logical conjunction $P \land Q$.

[4] Historically, The logic was named 'the logic of bunched implications', because it features the 'bunch' structure by the separating conjunction $*$ and the separating 'implication' $-\!\!*$. Some people use the term 'separation logic' for program logic that is built on the logic of bunched implications (i.e., uses the logic of bunched implications for its assertion logic). In this dissertation, we use the term 'separation logic' for both base and program logic, which seems common nowadays.

[5] This dissertation uses P, Q, R for separation logic propositions, Φ, Ψ for separation logic predicates, and ϕ, ψ for *pure* propositions or predicates in the basic logic.

Intuitively, a proposition $P \in \mathit{iProp}$ in separation logic[6] asserts the *ownerhip* of some fragment of *mutable state* and some *up-to-date knowledge* about it. The separating conjunction $P * Q$ asserts the ownership and knowledge of two *disjoint* fragments of mutable state respectively owned by P and Q.

Separation logic has proved exceptionally useful for reasoning about mutable state and has been an active and fruitful subject of research (Brookes and O'Hearn, 2016; O'Hearn, 2019). In particular, the inventors of concurrent separation logic (O'Hearn, 2004; Brookes, 2004), separation logic applied to concurrent programs, were awarded the Gödel Prize, the prestigious award in theoretical computer science (European Association for Theoretical Computer Science, 2016).

Basics: Separation Logic for Heap Memory The most common usage of separation logic is to reason about mutable state allocated in *heap memory*.

The key assertion for that is the *points-to token* $\ell \mapsto v \in \mathit{iProp}$, which has the *ownership* of the memory cell at the location (or memory address) $\ell \in \mathit{Loc}$ and the *up-to-date knowledge* that currently the value $v \in \mathit{Val}$ is stored in the memory cell.

Now we build Hoare logic (§ 1.1.1) that uses separation logic for the assertion logic, employing the Hoare triple $\{P\}\, e\, \{\Psi\}$ for $P \in \mathit{iProp}$ and $\Psi \colon \mathit{Val} \to \mathit{iProp}$.[7] This Hoare logic can reason about programs that manipulate heap memory.

Under the assertion $\ell \mapsto v$, we can safely *read* (a.k.a. *load*) from the location ℓ and know that the value v is obtained by the read:

$$\{\, \ell \mapsto v \,\}\ !\ell\ \{\, \lambda v'.\ v' = v * \ell \mapsto v \,\} \qquad \textsc{phoare-load}$$

On the other hand, we can safely *write* (a.k.a. *store*) any new value $w \in \mathit{Val}$ to the location ℓ as long as we *update* the assertion $\ell \mapsto v$ to $\ell \mapsto w$:

$$\{\, \ell \mapsto v \,\}\ \ell \leftarrow w\ \{\, \lambda\_.\ \ell \mapsto w \,\} \qquad \textsc{phoare-store}$$

Also, we have the following useful rule for ignoring the 'environment' R that is not relevant to the execution of the expression e:

$$\frac{\{P\}\, e\, \{\Psi\}}{\{P * R\}\, e\, \{\lambda v.\ \Psi\, v * R\}} \qquad \textsc{phoare-frame}$$

You can read this rule from the bottom up: to prove the Hoare triple $\{P * R\}\, e\, \{\lambda v.\ \Psi\, v * R\}$, it suffices to prove the Hoare triple $\{P\}\, e\, \{\Psi\}$ that ignores the unchanged 'environment' R. This rule is called the *frame* rule, with the 'environment' R called the *frame*. This is named after the 'frame problem' in the context of artificial intelligence (McCarthy and Hayes, 1981). The frame rule is a key to the *modular, scalable* reasoning about *mutable* state in separation logic, because the rule allows one to reason about each fragment e of a program focusing only on the relevant parts of mutable state manipulated by e.

Also, Hoare triples can be modified with entailment:

$$\frac{P' \vDash P \qquad \{P\}\, e\, \{\Psi\} \qquad \forall v.\ (\Psi\, v \vDash \Psi'\, v)}{\{P'\}\, e\, \{\Psi'\}} \qquad \textsc{phoare-}\vDash$$

[6] The name *iProp* is originally given to the propositions of the Iris separation logic (Jung et al., 2015, 2018b). Because we use Iris throughout our formulation of the Nola framework, for consistency, we use by extension the name *iProp* for the propositions of any kind of separation logic.

[7] We use the metavariables P, Q for propositions in separation logic *iProp* and Φ, Ψ for predicates of separation logic (i.e., functions to *iProp*) $A_1 \to \cdots A_n \to \mathit{iProp}$, while we use ϕ, ψ for *pure* propositions *Prop* (or predicates $A_1 \to \cdots A_n \to \mathit{Prop}$) in the base logic.

Semantics of Separation Logic We give the semantics to the basic separation logic and Hoare logic we built above. First, the domain *iProp* of separation logic propositions is defined as follows:

$$\mathit{iProp} \triangleq \mathit{Heap} \to \mathit{Prop} \qquad \mathit{Heap} \triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val} \tag{1.7}$$

The domain *Heap* represents a *heaplet*, or *fragment* of heap memory. Each heaplet $H \in \mathit{Heap}$ *owns* the memory cells of a finite number of locations $\operatorname{dom} H$ and *knows* that the value of the memory cell at each location $\ell \in \operatorname{dom} H$ is currently $H[\ell] \in \mathit{Val}$. Note that the global state of heap memory is also modeled as an element of *Heap*. Each separation logic proposition $P \in \mathit{iProp}$ owns some heaplet $H \in \mathit{Heap}$ satisfying the condition $P\,H$ asserted by P.

The points-to token $\ell \mapsto v$, the key assertion about heap memory, is modeled as follows:

$$\ell \mapsto v \quad \triangleq \quad \lambda H.\ \ell \in \operatorname{dom} H \ \land\ H[\ell] = v.$$

It says that the heaplet H of the assertion owns the memory cell at the location ℓ and knows that the value of the cell is v.

The separating conjunction $P * Q \in \mathit{iProp}$ is modeled as follows:

$$P * Q \quad \triangleq \quad \lambda H.\ \exists H_1, H_2 \text{ s.t. } H = H_1 + H_2.\ P\,H_1 \ \land\ Q\,H_2. \tag{1.8}$$

The model of the separating conjunction says that the heaplet H of $P * Q$ can be decomposed into two *disjoint* heaplets H_1, H_2 such that P owns H_1 and Q owns H_2. Here, we define the *disjoint union* $f + g$ of partial maps f, g as the union $f \cup g$ that is defined only if their domains are disjoint $\operatorname{dom} f \cap \operatorname{dom} g = \varnothing$.

A pure assertion $\phi \in \mathit{Prop}$ like $v' = v$ can be embedded into *iProp* simply as $\lambda\_.\ \phi$, ignoring the heaplet.

The entailment $P \vDash Q$ in separation logic is modeled as follows, using universal quantification over the heaplet H:

$$P \vDash Q \quad \triangleq \quad \forall H.\ P\,H \to Q\,H.$$

Now we can model the Hoare triple $\{P\}\ e\ \{\Psi\}$ as follows:

$$\{P\}\ e\ \{\Psi\} \quad \triangleq \quad \forall R.\ \ P * R \ \vDash\ \operatorname{pwp} e\ \{\lambda v.\ \Psi\,v\ *\ R\} \tag{1.9}$$

$$\begin{aligned} \operatorname{pwp} e\ \{\Psi\}\ H \quad \triangleq_\nu \quad & (\exists v \in \mathit{Val} \text{ s.t. } e = v.\ \Psi\,v\,H) \ \lor \\ & \big((\exists (e', H').\ (e, H) \hookrightarrow (e', H')) \ \land\ \forall (e', H') \hookleftarrow (e, H).\ \operatorname{pwp} e'\ \{\Psi\}\ H'\big). \end{aligned} \tag{1.10}$$

The overall structure is quite analogous to the definition for the stateless setting (1.3) and (1.4) (§ 1.1.1). The Hoare triple (1.9) is defined with universal quantification over the 'frame' R to admit the frame rule PHOARE-FRAME. The reduction of expressions now takes into account the global state of heap memory, having the form $(e, H) \hookrightarrow (e', H')$. The predicate $\operatorname{pwp} e\ \{\Psi\}\ H \in \mathit{Prop}$ (1.10) is defined quite like (1.4) but takes around the global state of heap memory H in the store-passing style.

Total Hoare Triple in Separation Logic We can also consider the *total* Hoare triple $[P]\ e\ [\Psi]$ instead of the partial one (§ 1.1.2). The total Hoare triple satisfies the rules exactly like the partial Hoare triple:

$$[\,\ell \mapsto v\,]\ \ !\ell\ \ [\,\lambda v'.\ v' = v\ *\ \ell \mapsto v\,] \qquad \text{THOARE-LOAD}$$

$$[\,\ell \mapsto v\,]\ \ \ell \leftarrow w\ \ [\,\lambda\_.\ \ell \mapsto w\,] \qquad \text{THOARE-STORE}$$

$$\frac{\big[P\big]\, e\, \big[\Psi\big]}{\big[P * R\big]\, e\, \big[\lambda v.\ \Psi v * R\big]}\ \textsc{thoare-frame}$$

$$\frac{P' \vDash P \qquad \big[P\big]\, e\, \big[\Psi\big] \qquad \forall v.\ \big(\Psi v \vDash \Psi' v\big)}{\big[P'\big]\, e\, \big[\Psi'\big]}\ \textsc{thoare-}\vDash$$

The only difference is that the total Hoare triple guarantees the termination and thus prohibits coinductive reasoning like the partial one. Note that the total Hoare triple entails the partial one:

$$\frac{\big[P\big]\, e\, \big[\Psi\big]}{\big\{P\big\}\, e\, \big\{\Psi\big\}}\ \textsc{thoare-phoare}$$

The semantics can be constructed in a similar way to the partial setting. The total Hoare triple can be modeled as

$$\big[P\big]\, e\, \big[\Psi\big] \quad \triangleq \quad \forall R.\ P * R \vDash \mathsf{twp}\ e\ \big[\lambda v.\ \Psi v * R\big],$$

just like the partial one (1.9). The total weakest precondition $\mathsf{twp}\ e\ \big[\Psi\big]\ H \in \mathit{Prop}$ is modeled just like the partial one (1.10) but using the least fixed point $\triangleq_\mu$ instead of the greatest fixed point $\triangleq_\nu$.

Example: Singly Linked List For a non-trivial example, we verify an iterative mutation of a singly linked list.

First, the separation logic assertion $\mathsf{olist}_\ell\ \mathit{ns} \in \mathit{iProp}$ of owning a singly linked list for integers $\mathit{ns} \in \mathit{List}\,\mathbb{Z}$ starting at the location $\ell \in \mathit{Loc}$ is defined as follows:

$$\mathsf{olist}_\ell\ [] \ \triangleq\ \ell \mapsto \mathsf{false}$$

$$\mathsf{olist}_\ell\ (n : \mathit{ns}) \ \triangleq\ \ell \mapsto \mathsf{true} * (\ell+1) \mapsto n * \exists \ell'.\ (\ell+2) \mapsto \ell' * \mathsf{olist}_{\ell'}\ \mathit{ns}$$

Here, we use the existential quantifier in *iProp*, which is simply modeled as follows:

$$\exists a.\ P_a \quad \triangleq \quad \lambda H.\ \exists a.\ P_a\, H$$

The assertion olist_ℓ can be described as follows. First, the location ℓ stores the tag of the list, being false for the nil case and true for the cons case. For the cons case for $n : \mathit{ns}$, the second location $\ell + 1$ stores the integer value n and the third location $\ell + 2$ stores the location ℓ' of the tail list.

Consider the following recursive function iterincr that increments every element of a singly linked list:

$$\mathsf{iterincr}(\ell) \quad \triangleq \quad \mathsf{if}\ !\ell\ \mathsf{then}\ (\ell+1) \leftarrow !(\ell+1)+1;\ \mathsf{iterincr}(!(\ell+2))$$

We can prove the following *total* Hoare triple for any $\mathit{ns} \in \mathit{List}\,\mathbb{Z}$:

$$\big[\mathsf{olist}_\ell\ \mathit{ns}\big]\ \mathsf{iterincr}(\ell)\ \big[\lambda\_.\ \mathsf{olist}_\ell\,(\mathsf{map}\ (\lambda n.\, n+1)\ \mathit{ns})\big]$$

The proof goes simply by induction on the length of *ns*.

Concurrent Separation Logic Separation logic works well for *concurrent* programs, just like for sequential programs (O'Hearn, 2004; Brookes and O'Hearn, 2016). Separation logic for concurrent programs is sometimes called *concurrent separation logic.*

For example, if the language has the primitive $\mathsf{fork}\,\{\,e\,\}$ for forking a thread that executes the expression e, we can add the following Hoare triple rules:

$$\frac{\big\{P\big\}\, e\, \big\{\lambda\_.\, \top\big\}}{\big\{P\big\}\ \mathsf{fork}\,\{\,e\,\}\ \big\{\lambda\_.\, \top\big\}}\ \textsc{phoare-fork} \qquad \frac{\big[P\big]\, e\, \big[\lambda\_.\, \top\big]}{\big[P\big]\ \mathsf{fork}\,\{\,e\,\}\ \big[\lambda\_.\, \top\big]}\ \textsc{thoare-fork}$$

Here we write $\top \in iProp$ for the pure proposition $\lambda\_.\,\top$. Combining these rules with the frame rules PHOARE-FRAME, THOARE-FRAME, we can derive the following:

$$\frac{\{P\}\,e\,\{\lambda\_.\,\top\}}{\{P * Q\}\ \mathsf{fork}\,\{\,e\,\}\ \{\lambda\_.\,\top * Q\}} \qquad \frac{[P]\,e\,[\lambda\_.\,\top]}{[P * Q]\ \mathsf{fork}\,\{\,e\,\}\ [\lambda\_.\,\top * Q]} \tag{1.11}$$

At the high level, this reasoning is sound thanks to the *separating conjunction* $*$ clearly separating the parts P, Q of mutable state to be owned by the two computational agents, the child thread (owning P) and the parent thread (owning Q). The semantics for concurrent separation logic is somewhat trickier than the sequential setting, but it has long been known (Brookes, 2004).

Shared Immutable State by Fractional Ownership Separation logic works perfectly when the parts of mutable state owned by different computational agents are clearly *disjoint* with respect to the separating conjunction $*$. But naive separation logic suffers in the situation where multiple computational agents *share* some parts of mutable state.

For *shared immutable state*, or state *not mutated while shared*, the situation is easy. A standard approach to such immutable sharing is to use *fractional ownerhip* (Boyland, 2003; Bornat et al., 2005), a 'fictional' form of ownership extended with *fractional* quantities for splitting.

For heap memory, we can use the *fractional* points-to token $\ell \overset{q}{\mapsto} v$, parameterized with a *fraction* $q \in \mathbb{Q}_{>0}$.[8] The original points-to token $\ell \mapsto v$ is defined as $\ell \overset{1}{\mapsto} v$, using 1 for the *full* fraction. The fractional points-to token can be *split* and *merged* according to fractions:

$$\ell \overset{q+r}{\mapsto} v \;=\; \ell \overset{q}{\mapsto} v * \ell \overset{r}{\mapsto} v \qquad \mapsto\text{-FRACT}$$

Also, the fraction of the points-to token cannot be more than 1:

$$\frac{q > 1}{\ell \overset{q}{\mapsto} v \;=\; \bot} \qquad \mapsto\text{-OVER1}$$

Two fractional points-to tokens to one location are ensured to observe the exact same value of the memory cell:

$$\ell \overset{q}{\mapsto} v * \ell \overset{r}{\mapsto} v' \;\models\; v = v' * \ell \overset{q+r}{\mapsto} v \qquad \mapsto\text{-AGREE}$$

We can read from the memory with a *fractional* points-to token $\ell \overset{q}{\mapsto} v$, using the following rules that extend PHOARE-LOAD and THOARE-LOAD:

$$\{\,\ell \overset{q}{\mapsto} v\,\}\ !\ell\ \{\,\lambda v'.\, v' = v * \ell \overset{q}{\mapsto} v\,\} \qquad \text{PHOARE-LOAD}^*$$

$$[\,\ell \overset{q}{\mapsto} v\,]\ !\ell\ [\,\lambda v'.\, v' = v * \ell \overset{q}{\mapsto} v\,] \qquad \text{THOARE-LOAD}^*$$

On the other hand, we cannot write a new value to the memory just by updating a fractional points-to token $\ell \overset{q}{\mapsto} v$ of $q < 1$, because there can exist another points-to token $\ell \overset{r}{\mapsto} v$, which is invalidated by that memory write.

1.3 Propositional Sharing

Sharing is caring

A saying in English

[8] We write $\mathbb{Q}_{>0}$ for the set of positive rational numbers.

Challenge: Shared Mutable State The real challenge lies in *shared mutable state*, i.e., state that can be *mutated while shared*. Programmers commonly use mechanisms for shared mutable state.

For a simple example, functional languages such as MyCaml and Standard ML support a *shared mutable reference* `ref T` for a copyable reference to a mutable memory cell that stores an object of type `T`.[9] For a more advanced example, Rust supports a *shared* reference to a *mutex-guarded object* `&Mutex<T>`, which can be safely copied and shared across threads. Any thread can *acquire* the lock of the mutex via the reference to temporarily get the *ownership* of the content object of type `T`, being able to mutate the object freely.

How can we reason about such shared mutable state?

At a high level, mutations of shared mutable state across multiple computational agents generally follow some *contract* (or *protocol*) for any kind of correctness.

For example, the shared mutable reference type `ref T` expresses the contract that always an object typed `T` is stored in the memory cell that the reference points to. The contract for a shared reference to a mutex-guarded object in Rust `&Mutex<T>` is trickier: the guarded object `T` can be taken out only after the lock is acquired, and any object of type `T` should be stored back when the lock is released.[10] The contracts in the examples above are pretty rich and even parameterized over the *type* `T`.

Propositional Sharing In order to express and reason about such rich contracts, modern separation logics like Iris (Jung et al., 2015, 2018b) have introduced advanced mechanisms for *propositional sharing*, i.e., sharing of mutable state among multiple computational agents based on contracts *described by separation logic propositions*.[11]

Separation logics that support propositional sharing have achieved a great success in various challenging verification projects (Jung et al., 2018a; Timany et al., 2018; Hinrichsen et al., 2020; Dang et al., 2020; Gregersen et al., 2021; Hinrichsen et al., 2022; Matsushita et al., 2022; Jacobs et al., 2022; Timany et al., 2023). Notable among them is *RustBelt* (Jung et al., 2018a). It established a semantic foundation for Rust's ownership type system and formally proved the memory and thread safety of well-typed Rust programs for a realistic subset of Rust.

1.3.1 Shared Invariants

A central mechanism for propositional sharing is the *(shared) invariant* $\boxed{P} \in \mathit{iProp}$ (Hobor et al., 2008; Buisse et al., 2011; Svendsen et al., 2013; Svendsen and Birkedal, 2014), whose modern usage was established by the *Iris separation logic framework* (Jung et al., 2015, 2016; Krebbers et al., 2017a; Jung et al., 2018b).[12] [13] It is an assertion that

[9] To be precise, ML-family languages like MyCaml and Standard ML use (rather exotic) postfix syntax `T ref` for type constructors. But we use the prefix syntax `ref T` here for accessibility and consistency.

[10] To be precise, Rust's shared reference type `&U` is parameterized `&'a U` with the *lifetime* `'a`, the time period that the reference is valid. The type `&'a Mutex<T>` also asserts the contract that it acquires the lock only during the lifetime `'a`.

[11] The term 'propositional sharing' was coined by us to emphasize the fact that we are interested in *sharing* characterized by *separation logic propositions*. This concept is related to the terms *higher-order ghost state* and *second-order ghost state* introduced by Jung et al. (2016).

[12] Actually, the invariant connective $\boxed{P}^{\mathcal{N}}$ has the namespace $\mathcal{N} \in \mathit{Namespace}$ parameter, but we omit it for now.

[13] This mechanism is sometimes referred to as the *'impredicative invariant'*. However, what the word 'impredicative' means here is unclear. Impredicativity usually refers to self-referential definition. But Jung et al. (2018b, § 2.2) say as follows: "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." Using Nola, one can construct an invariant connective $\mathtt{inv}^{\mathcal{N}}\, P \in \mathit{nProp}$

a separation logic proposition $P \in iProp$ is maintained as an *'invariant'*, or that the situation described by P is always kept during the computation.

Proof Rules Notably, the invariant $\boxed{P}$ is *duplicable* with respect to the separating conjunction $*$:

$$\boxed{P} \;=\; \boxed{P} * \boxed{P} \qquad \textsc{iinv-dup}$$

In general, a separation logic proposition P is not duplicable ($P \neq P * P$) when it asserts *exclusive* ownership. This is the case especially when P directly contains a *points-to token* $\ell \mapsto v$, e.g., $P = (\ell \mapsto \text{true}) \vee (\ell \mapsto \text{false})$. On the other hand, the invariant $\boxed{P}$ is duplicable (iinv-dup) and thus *can be shared* among multiple computational agents, especially among multiple *threads* (recall (1.11) in § 1.2.3).

To use invariants, Iris introduces the *fancy update* modality $\Rrightarrow$, a *monadic* modality that represents a 'logical step' updating the inner state for invariants.[14] The fancy update $\Rrightarrow$ satisfies the following rules:

$$\frac{P \vDash Q}{\Rrightarrow P \vDash \Rrightarrow Q}\ \Rrightarrow\textsc{-mono} \qquad P \vDash \Rrightarrow P \quad \Rrightarrow\textsc{-intro}$$

$$\Rrightarrow\Rrightarrow P = \Rrightarrow P \quad \Rrightarrow\textsc{-idemp} \qquad (\Rrightarrow P) * Q \vDash \Rrightarrow(P * Q) \quad \Rrightarrow\textsc{-frame}$$

Also, Hoare triples $\{P\}\, e\, \{\Psi\}$, $[P]\, e\, [\Psi]$ are designed so that they absorb the fancy update $\Rrightarrow$, meaning that the 'logical step' of the fancy update $\Rrightarrow$ can be performed before and after any program execution step:

$$\frac{\{P\}\, e\, \{\Psi\}}{\{\Rrightarrow P\}\, e\, \{\Psi\}}\ \Rrightarrow\textsc{-phoare} \qquad \frac{[P]\, e\, [\Psi]}{[\Rrightarrow P]\, e\, [\Psi]}\ \Rrightarrow\textsc{-thoare}$$

$$\frac{\{P\}\, e\, \{\lambda v.\, \Rrightarrow \Psi\, v\}}{\{P\}\, e\, \{\Psi\}}\ \textsc{phoare-}\Rrightarrow \qquad \frac{[P]\, e\, [\lambda v.\, \Rrightarrow \Psi\, v]}{[P]\, e\, [\Psi]}\ \textsc{thoare-}\Rrightarrow$$

An invariant assertion $\boxed{P} \in iProp$ can be created by storing the separation logic proposition $P \in iProp$ in the fancy update $\Rrightarrow$:

$$P \vDash \Rrightarrow \boxed{P} \qquad \textsc{iinv-alloc}'$$

Roughly speaking, the fancy update manages an 'imaginary' memory of invariants that stores a proposition $P \in iProp$ for each invariant (see (1.18) etc. in § 1.4). The rule iinv-alloc′ 'allocates' a new memory cell that stores P to get the invariant $\boxed{P}$. Using this rule, one can allocate an invariant in Hoare triples as follows, for example:

$$\frac{\{\boxed{P} * Q\}\, e\, \{\Psi\}}{\{P * Q\}\, e\, \{\Psi\}} \qquad \frac{[\boxed{P} * Q]\, e\, [\Psi]}{[P * Q]\, e\, [\Psi]}$$

Iris has the following rules for accessing the shared content P of an invariant $\boxed{P}$ in Hoare triples:[15]

$$\frac{\{(\triangleright P) * Q\}\, e\, \{\lambda v.\, (\triangleright P) * \Psi\, v\}}{\{\boxed{P} * Q\}\, e\, \{\Psi\}}\ \textsc{phoare-iinv}$$

that can be *freely nested* and thus seems 'impredicative' in this sense. But we do not think such invariants are impredicative, because Nola's domain equations (see (1.20), § 1.5) are not self-referential. For clarity, we avoid using the term 'impredicative invariant' in this dissertation.

[14] For simplicity, we omit here the *mask* parameters $\mathcal{E}, \mathcal{E}'$ of the fancy update ${}_{\mathcal{E}}\Rrightarrow_{\mathcal{E}'}$, which serve to prohibit *reentrancy* to invariants in combination with the *namespace* $\mathcal{N}$ parameter of the invariant $\boxed{P}^{\mathcal{N}}$. See § 3.2.2 for the details.

[15] We should actually take care of the *mask* parameter $\mathcal{E}$ of Hoare triples, but we omit it here for simplicity. Also for simplicity, we omit an important side condition that the expression e should be *atomic*. See § 3.1 for the details.

$$\frac{\big[(\triangleright P) * Q\big]\; e \;\big[\lambda v.\ (\triangleright P) * \Psi\, v\big]}{\big[\boxed{P} * Q\big]\; e\; \big[\Psi\big]}\quad \textsc{thoare-iinv}$$

The rules roughly say that the content P of the invariant $\boxed{P}$ can be accessed in the precondition as long as P is restored in the postcondition. The content P is weakened by the *later modality* $\triangleright P$, an ill-behaved modality, and we discuss the problems about it later in § 1.4. But for now the reader can just ignore it. Iris has the following more basic rule for accessing the content of an invariant in terms of the fancy update $\Rrightarrow$:

$$\frac{(\triangleright P) * Q \;\vDash \Rrightarrow \big((\triangleright P) * R\big)}{\boxed{P} * Q \;\vDash \Rrightarrow R}\quad \textsc{iinv-acc}$$

Examples Using the invariant, one can express, for example, a *shared mutable reference* storing a boolean at the location $\ell \in Loc$ by

$$\ell : \texttt{ref bool} \quad \triangleq \quad \boxed{(\ell \mapsto \mathsf{true}) \lor (\ell \mapsto \mathsf{false})},$$

meaning that it is always the case that "ℓ stores true or false". Although this invariant can be *shared* (IINV-DUP), it also allows *writing* any boolean value $b \in \mathbb{B}$ to the location ℓ, because the following can be derived from PHOARE-IINV and PHOARE-STORE:[16] [17]

$$\Big\{ \boxed{(\ell \mapsto \mathsf{true}) \lor (\ell \mapsto \mathsf{false})} \Big\}\ \ell \leftarrow b\ \Big\{ \lambda\_.\ \top \Big\}.$$

Also, the invariant ensures that we can always *read* a boolean value from the location ℓ, because the following can be derived from PHOARE-IINV and PHOARE-LOAD:

$$\Big\{ \boxed{(\ell \mapsto \mathsf{true}) \lor (\ell \mapsto \mathsf{false})} \Big\}\ !\ell\ \Big\{ \lambda v.\ v = \mathsf{true} \lor v = \mathsf{false} \Big\}.$$

A reference to a boolean reference can be expressed in a similar way:

$$\ell : \texttt{ref (ref bool)} \quad \triangleq \quad \boxed{\exists \ell'.\ (\ell \mapsto \ell') * \boxed{(\ell' \mapsto \mathsf{true}) \lor (\ell' \mapsto \mathsf{false})}} \tag{1.12}$$

The key to expressing such *nested reference types* is the ability to *nest* the invariant connective $\boxed{-}$. We can freely *copy* the inner reference ℓ' from this nested reference:

$$\Big\{ \big(\ell : \texttt{ref (ref bool)}\big) \Big\}\ !\ell\ \Big\{ \lambda v.\ \exists \ell' \text{ s.t. } v = \ell'.\ \triangleright \boxed{(\ell' \mapsto \mathsf{true}) \lor (\ell' \mapsto \mathsf{false})} \Big\} \tag{1.13}$$

Also, we can freely *mutate* the inner reference of this nested reference to any location ℓ'' satisfying the invariant $\ell'' : \texttt{ref bool}$:

$$\Big\{ \big(\ell : \texttt{ref (ref bool)}\big) * \boxed{(\ell'' \mapsto \mathsf{true}) \lor (\ell'' \mapsto \mathsf{false})} \Big\}\ \ell \leftarrow \ell''\ \Big\{ \lambda\_.\ \top \Big\} \tag{1.14}$$

Note that the operations like (1.13) and (1.14) are *not* supported by the following variant $\ell : \texttt{ref (ownref bool)}$ of (1.12), which removes the invariant connective for the inner reference and thus does not nest invariants:

$$\ell : \texttt{ref (ownref bool)} \quad \triangleq \quad \boxed{\exists \ell'.\ (\ell \mapsto \ell') * \big((\ell' \mapsto \mathsf{true}) \lor (\ell' \mapsto \mathsf{false})\big)}$$

[16] We assume a simple memory model where the load and store operations are atomic.

[17] We can ignore the later modality $\triangleright$ put on the content $(\ell \mapsto \mathsf{true}) \lor (\ell \mapsto \mathsf{false})$, because the content is classified as a *timeless* proposition (explained later in § 2.1.2). Simple propositions such as the points-to token $\ell \mapsto v$ and the disjunction of timeless propositions are timeless.

For a more interesting example, we model a shared reference to a mutex-guarded object &Mutex<T> in Rust.[18] We can model the type refmutex T for such a reference as follows:

$$\ell : \texttt{refmutex}\ T \quad \triangleq \quad \boxed{\big(\ell \mapsto \mathsf{false} \ast T(\ell+1)\big) \lor \ell \mapsto \mathsf{true}} \tag{1.15}$$

Here, we model the type T as the separation logic predicate $T\colon \mathit{Loc} \to \mathit{iProp}$. The body of the invariant is the disjunction of two cases, *unlocked* or *locked*. When the mutex is unlocked, the tag at ℓ is set to false and the object T is stored at the next location $\ell + 1$. When the mutex is locked, the tag is set to true. A thread can try to acquire the lock with *compare-and-swap* cas, and when it succeeds, the ownership of the object T is transferred to the thread:

$$\big\{\, \ell : \texttt{refmutex}\ T \,\big\}\ \mathsf{cas}(\ell, \mathsf{false}, \mathsf{true})\ \big\{\, \lambda v.\ v = \mathsf{false} \lor \big(v = \mathsf{true} \ast \triangleright T(\ell+1)\big) \,\big\}.$$

Here, the *compare-and-swap* $\mathsf{cas}(\ell, v, w)$ is an atomic primitive operation that checks if the current value stored at the location ℓ is v, updates the value at ℓ to w only if the check succeeds, and returns whether the check has succeeded and the update has been performed. This operation satisfies the following Hoare triple rules:

$$\big\{\, \ell \mapsto v \,\big\}\ \mathsf{cas}(\ell, v, w)\ \big\{\, \lambda u.\ u = \mathsf{true} \ast \ell \mapsto w \,\big\} \quad \text{PHOARE-CAS-TRUE}$$

$$\frac{v' \neq v}{\big\{\, \ell \mapsto v' \,\big\}\ \mathsf{cas}(\ell, v, w)\ \big\{\, \lambda u.\ u = \mathsf{false} \ast \ell \mapsto v \,\big\}} \quad \text{PHOARE-CAS-FALSE}$$

The thread can release the lock by storing back the ownership of the object T and setting the tag to false:

$$\big\{\, \ell : \texttt{refmutex}\ T \ast \triangleright T(\ell+1) \,\big\}\ \ell \leftarrow \mathsf{false}\ \big\{\, \lambda\_.\, \top \,\big\}.$$

Rust-Style Borrows Another key mechanism for propositional sharing is *Rust-style borrows*.

The *Rust* programming language employs a powerful *ownership type system*, as introduced in § 1.2.2. A key feature of Rust is the *borrowing* machinery. In Rust, the ownership of an object of the type T can be temporarily *borrowed* to create a *mutable reference* of the type &'a **mut** T, which can freely update the borrowed object during the time period of the *lifetime* 'a. Various flexible operations are supported for borrowing, including *subdivision* and *reborrowing*. Rust's borrowing machinery can be understood as a form of *sharing* between the borrowers and the lender, under a *contract* described by *ownership types* T.

RustBelt's *lifetime logic* (Jung et al., 2018a, § 5) semantically modeled Rust-style borrows in the *Iris separation logic* as an advanced form of *propositional sharing*, which can be roughly seen as an advanced version of Iris's invariant mechanism. In particular, it introduces an Iris proposition called *full borrow* $\&^{\alpha}_{\mathsf{full}} P \in \mathit{iProp}$, which is roughly an advanced version of Iris's invariant $\boxed{P}$ that can model Rust's mutable reference type &'a **mut** T. Using the lifetime logic, *RustBelt* (Jung et al., 2018a) established a semantic foundation for Rust's ownership type system, formally verifying the memory and thread safety of well-typed Rust programs for a realistic subset of Rust. RustBelt's lifetime logic has also been used by other research work (Dang et al., 2020; Yanovski et al., 2021; Matsushita et al., 2022). Still, the lifetime logic also suffers from the *later modality* $\triangleright$ in accessing the borrowed content, just like Iris's invariant mechanism.

We explain Rust's borrows and RustBelt's lifetime logic in more detail later in § 6.1.

[18] For simplicity, we think just of the case where the lifetime 'a of the reference &'a Mutex<T> is *static* 'static, i.e., lasts forever in the program execution. By introducing *Rust-style borrows*, we can model a general situation where the lifetime may not be static. See also § 6.3.3.

1.4 Problem: Later Modality and Step-Indexing

Necessity is the mother of invention

A saying in English

Later Modality Existing approaches to propositional sharing presented in the previous section § 1.3 suffered from the *later modality* $\triangleright$. In these approaches, access to contents under propositional sharing is allowed only under this modality $\triangleright$ (recall PHOARE-IINV, THOARE-IINV and IINV-ACC in § 1.3.1).

First, the later modality $\triangleright$ slightly *weakens* a proposition:

$$P \models \triangleright P \qquad \triangleright P \not\models P.$$

Also, the modality is *ill-behaved*, in that it is not idempotent and does not commute with the fancy update modality $\Rrightarrow$:

$$\triangleright\triangleright P \neq \triangleright P \qquad \triangleright \Rrightarrow P \not\models \Rrightarrow \triangleright P \qquad \Rrightarrow \triangleright P \not\models \triangleright \Rrightarrow P.$$

In particular, connectives for propositional sharing such as the invariant $\boxed{P}$ lose the power to access the shared content when put under the later modality $\triangleright$.[19] One comes across such connectives under the later modality when these connectives are *nested*, which is essential to express especially *nested reference types*, as we saw in the example of `ref (ref bool)` (1.12) (§ 1.3.1).

This calls for the ability to *strip off the later modality* $\triangleright$ in program verification.

Paradox of a naive later-free invariant One might expect the following naive rule, a variant of THOARE-IINV without the later modality to hold instead:

$$\frac{\left[P * Q\right] e \left[\lambda v.\ P * \Psi v\right]}{\left[\boxed{P} * Q\right] e \left[\Psi\right]} \ \text{THOARE-IINV-NAIVE ?}$$

However, this naive rule—where the shared content P can be an *arbitrary* separation logic proposition—causes a *paradox*, wrongly proving a non-terminating program to terminate.

As a background, it is well known that higher-order references can cause non-termination, by the following folklore construction known as *Landin's knot* (written in OCaml):

```
let l : ref (unit -> unit) = ref (fun _ -> ()) in
l := (fun _ -> !l ());   !l ()
```

Code 1.4: Landin's knot

We create a *shared mutable reference* to a *closure* `l : ref (unit -> unit)`, with a benign initial value `fun _ -> ()`. Then we update the reference's content into a closure that calls the closure stored in the reference `l` itself. Finally, we call the closure stored in the reference `l`, which causes an infinite loop.

Now let us consider the following expression landin corresponding to Code 1.4 expressed in the target language of our program logic:

$$\mathsf{landin} \quad \triangleq \quad \mathsf{let}\ \ell := \mathsf{ref}\left(\mathsf{fun}()\{\}\right)\ \mathsf{in}\quad \ell \leftarrow \mathsf{fun}()\{!\ell()\};\quad !\ell() \tag{1.16}$$

[19] The reason for this is described later in § 3.1.3.

The naive rule THOARE-IINV-NAIVE allows us to wrongly prove that landin terminates, i.e., prove the following:

$$\left[\top\right]\ \mathsf{landin}\ \left[\top\right] \tag{1.17}$$

First, when the reference ℓ is initialized with the trivial function $\mathsf{fun}\,()\,\{\,\}$, we can create the following invariant J:

$$J \ \triangleq\ \boxed{\exists f.\ \ell \mapsto f \ *\ \left[\top\right] f()\left[\top\right]}.$$

We can (wrongly) verify the safety $\left[J\right] \ell \leftarrow g \left[\top\right]$ of storing the new function $g \triangleq \mathsf{fun}\,()\,\{\,!\ell\,()\,\}$ by applying the naive rule THOARE-IINV-NAIVE. Here, the premise of the rule is satisfied by proving $\left[J\right] g() \left[\top\right]$, by applying THOARE-IINV-NAIVE to (wrongly) prove $\left[J\right] !\ell\,() \left[\top\right]$.[20] Finally, the last function call $!\ell\,()$ has been proved to terminate, which is a contradiction.

The rule with the later modality $\triangleright$ avoids this paradox, because the later modality $\triangleright$ weakens the total Hoare triple $\left[\top\right] f() \left[\top\right]$ inside the invariant J, which correctly stops the assertion $\left[J\right] !\ell\,() \left[\top\right]$ from being proved.

This paradox for the total Hoare triple is natural but seems new in the literature. There is also a related known paradox found by Krebbers et al. (2017a, § 5), which is about the fancy update $\Rrightarrow$. We discuss and analyze these paradoxes later in § 3.4.

Step-Indexing A standard remedy to this is to employ *step-indexing* (Nakano, 2000; Appel and McAllester, 2001; Birkedal et al., 2012) in program logic, or let each *execution step* of the program strip off one later modality.[21] Step-indexing admits the following Hoare triple rule for stripping one later modality $\triangleright$ off of the precondition for each (pure) reduction step $e \hookrightarrow e'$:

$$\frac{e \hookrightarrow e' \qquad \{P\}\, e'\, \{\Psi\}}{\{\triangleright P\}\, e\, \{\Psi\}}\ \textsc{step-phoare}$$

In terms of the weakest precondition, we have the following rule:

$$\frac{e \hookrightarrow e' \qquad P \vDash \mathsf{pwp}\ e'\,\{\Psi\}}{\triangleright P \vDash \mathsf{pwp}\ e\,\{\Psi\}}\ \textsc{step-pwp}$$

Such elimination of the later modality is admissible because, roughly speaking, the weakest precondition is defined like the following (for simplicity, we ignore the state mutation and the stuckness check altogether):

$$\mathsf{pwp}\ e\,\{\Psi\} \quad \text{is roughly} \quad \big(\exists v \text{ s.t. } e = v.\ \Psi\, v\big) \ \vee\ \big(\forall e' \hookleftarrow e.\ \triangleright \mathsf{pwp}\ e'\,\{\Psi\}\big)$$

Each recursive occurrence of pwp after a reduction $e' \hookleftarrow e$ is guarded by the later modality $\triangleright$, making the rules like STEP-PHOARE and STEP-PWP admissible.

[20] Technically, this reasoning should work because the invariant $\boxed{P}$ is persistent and the Hoare triple is defined via the persistence modality $\Box$ and the weakest precondition: $\left[P\right] e \left[\Psi\right] \triangleq \Box(P \to \mathsf{twp}\ e\left[\Psi\right])$. In particular, we can get $\left[\top\right] g() \left[\top\right]$ from the persistent premises $\left[J\right] g() \left[\top\right]$ and J.

[21] We use the term *indexing* for the act of stratifying the semantics of the (base) *logic* by indices (typically natural numbers), whose increase is expressed by the later modality $\triangleright$. On the other hand, we use the term *step-indexing* for a design of *program semantics* such that execution steps are associated with the indices of the base logic. Some people (including the Iris community) use the term step-indexing for both, but we distinguish the two for clarity.

Step-Indexing Blocks Liveness Verification However, there are fundamental difficulties in using step-indexed program logic for verifying *liveness properties*, including termination and total correctness (§ 1.1.2). This problem is well-known. For example, Gäher et al. (2022, § 1.1) say as follows (citations reindexed):

> However, Iris's use of *step-indexing* (Appel and McAllester, 2001) means that Iris-based approaches like ReLoC (Frumin et al., 2018) do not support reasoning about liveness properties such as termination preservation.

Gäher et al. (2022) built Simuliris, a separation logic framework built on Iris. Simuliris provides a relational program logic that targets fair-termination-sensitive contextual refinement, which is a *liveness property*. Because step-indexing blocks reasoning about liveness properties as the quote says, Simuliris is *not step-indexed*. In turn, Simuliris *gives up* supporting the later-requiring mechanisms for propositional sharing, such as Iris's invariants.

The difficulties with liveness properties can be observed via the following *Löb induction* rule that holds for the later modality:

$$\frac{\triangleright P \vDash P}{\top \vDash P}\ \textsc{löb}$$

It says that, when P can be proved assuming $\triangleright P$, P holds without any assumption. Semantically, this rule can be proved by induction over the internal indices for the later modality $\triangleright$.

At the high level, the rule admits *coinductive* reasoning under the weakening by the later modality $\triangleright$. It thus enables coinductive reasoning about loops and recursions in programs for program logic with the ability to eliminate the later modality. But this power is a double-edged sword. It generally makes the program logic ensure only *safety* properties but not *liveness* properties.

We can actually prove a *paradox* for total program logic with the ability to eliminate the later modality $\triangleright$. First, assume toward contradiction that the total weakest precondition $\mathsf{twp}\ e\ [\Psi]$ admits the following later-eliminating rule like STEP-PWP:

$$\frac{e \hookrightarrow e' \qquad P \vDash \mathsf{twp}\ e'\ [\Psi]}{\triangleright P \vDash \mathsf{twp}\ e\ [\Psi]}\ \textsc{step-twp}?$$

Also, suppose a loop expression loop satisfying $\mathsf{loop} \hookrightarrow \mathsf{loop}$. Then combining STEP-TWP with Löb induction LÖB, we can prove the following:

$$\cfrac{\cfrac{\mathsf{loop} \hookrightarrow \mathsf{loop} \qquad \mathsf{twp}\ \mathsf{loop}\ [\lambda\_.\,\bot] \vDash \mathsf{twp}\ \mathsf{loop}\ [\lambda\_.\,\bot]}{\triangleright\, \mathsf{twp}\ \mathsf{loop}\ [\lambda\_.\,\bot] \vDash \mathsf{twp}\ \mathsf{loop}\ [\lambda\_.\,\bot]}\ \textsc{step-twp}}{\top \vDash \mathsf{twp}\ \mathsf{loop}\ [\lambda\_.\,\bot]}\ \textsc{löb}$$

This means that the execution of loop terminates with the postcondition $\bot$, which is clearly a contradiction. Indeed, the total weakest precondition $\mathsf{twp}\ e\ [\Psi]$ in Iris is defined in a *non-step-indexed* way with the *least fixed point* (like (1.6)).

Problems in Safety Verification The later modality $\triangleright$ can be problematic even in *safety* verification.

For example, in order to traverse a nested data structure modeled with k-fold nesting of later-requiring propositional sharing, one should eliminate k laters $\triangleright^k$, or even worse, k laters interleaved with the fancy update $(\triangleright \Rrightarrow)^k$. It is often the case that we want to perform such a traversal in a *logical* step, without performing any *physical* step of the program execution. Rules like STEP-PHOARE, stripping one later for one program step, are not enough for this purpose.

Various workarounds have been proposed to tackle such problems with the later modality (see § 9.3 for details), but they are often costly and ad hoc. We would be better off if we did not need the later modality $\triangleright$ for propositional sharing.

Later Modality from Indexed Semantics Existing approaches suffered from the *later modality* $\triangleright$ for the invariant mechanism because they used *indexed semantics* of separation logic to model invariants. We explain the reason for this.

Naively, to manage invariants $\boxed{P}$, we introduce an imaginary 'memory' of invariants *InvMem* that maps each 'invariant name' $\iota \in \mathit{InvName}$ to a separation logic proposition $P \in \mathit{iProp}$, extending the traditional model of separation logic propositions that just considers heap memory (1.7) (§ 1.2.3):

$$\mathit{iProp} \ \triangleq_? \ \mathit{Heap} \times \mathit{InvMem} \to \mathit{Prop} \qquad \mathit{InvMem} \ \triangleq_? \ \mathit{InvName} \xrightarrow{\text{fin}} \mathit{iProp} \tag{1.18}$$

Unfortunately, this definition is *invalid*. The domain equations of (1.18) are *not solvable*, because they have *circular dependencies* between the separation logic propositions *iProp* and the invariant memory *InvMem*.

Existing approaches like Iris solve this by *indexing* the semantics over some well-ordered set $\mathbb{I}$, which is typically the set of natural numbers $0, 1, 2, \ldots \in \mathbb{N}$. The domain equations for *iProp* and *InvMem* are modified as follows:

$$\mathit{iProp} \ \triangleq \ \mathit{Heap} \times \mathit{InvMem} \xrightarrow{\text{ne}} \widetilde{\mathit{Prop}} \qquad \mathit{InvMem} \ \triangleq \ \mathit{InvName} \xrightarrow{\text{fin}} \blacktriangleright \mathit{iProp} \tag{1.19}$$

An *indexed* proposition $\tilde{\phi} \in \widetilde{\mathit{Prop}} \triangleq \mathbb{I} \xrightarrow{\text{anti}} \mathit{Prop}$ is a predicate over indices $\mathbb{I}$ that is antitone, i.e., becomes more refined as the index increases.[22] The indexed propositions $\widetilde{\mathit{Prop}}$ support the *later modality*

$$\triangleright \tilde{\phi} \ \triangleq \ \lambda i.\ \forall j < i.\ \tilde{\phi}\ j,$$

which intuitively means that $\tilde{\phi}$ will hold in the 'next' index. The sets like *InvName* and *iProp* are *indexed*. An *indexed set* A is a set equipped with an *indexed equality* ($\tilde{=}$): $A \times A \to \widetilde{\mathit{Prop}}$, whose limit $\forall i.\ (a \mathrel{\tilde{=}} a')\ i$ agrees with the genuine equality $a = a'$.[23] The *later* constructor $\blacktriangleright$ on an indexed set weakens the indexed equality $\tilde{=}$ by the later modality $\triangleright$. Functions between indexed sets are required to be *non-expansive* (denoted as $\xrightarrow{\text{ne}}$), i.e., respect the indexed equality $\tilde{=}$.

The 'indexed' domain equations of (1.19) are *solvable*, thanks to the guard of the later constructor $\blacktriangleright$ (America and Rutten, 1989; Birkedal et al., 2010). At the same time, this later constructor $\blacktriangleright$ is the exact source of the *later modality* $\triangleright$ the invariant mechanism suffers from in access rules like IINV-ACC.

1.5 Our Solution: Nola

The day providential to itself. The hour. There is no later.

Cormac McCarthy, *The Road*

This dissertation proposes a novel general framework, *Nola*,[24] providing separation logic with advanced mechanisms for *propositional sharing* such as invariants and borrows without requiring the *later modality* $\triangleright$ in access.

[22] In this dissertation, we use tilde to mark indexed things.

[23] The indexed set is also known as the *ordered family of equivalence relations (OFE)*.

[24] The name of our framework, Nola, has two origins. One is short for 'No later', since the framework

Our Key Idea Recall the observation about the *semantics* of separation logic presented just above. The later modality in existing approaches comes from the *circular domain equations* (1.18) between the separation logic propositions *iProp* and the resources for propositional sharing like *InvMem*.

Our key idea is that we do not need to directly use *iProp* in the resources like *InvMem*. Instead, we use a *syntactic data type nProp* of propositions for propositional sharing and separately build the *semantic interpretation* $[\![\ ]\!] \colon \mathit{nProp} \to \mathit{iProp}$, interpreting each syntactic proposition $P \in \mathit{nProp}$ into a semantic proposition $[\![P]\!] \in \mathit{iProp}$. For example, we can build the model of separation logic with invariants as follows, without indexing the semantics:

$$\begin{gathered} \mathit{InvMem} \triangleq \mathit{InvName} \xrightarrow{\text{fin}} \mathit{nProp} \\ \mathit{iProp} \triangleq \mathit{Heap} \times \mathit{InvMem} \to \mathit{Prop} \qquad [\![\ ]\!] \colon \mathit{nProp} \to \mathit{iProp} \end{gathered} \tag{1.20}$$

This semantics provides a *new* logic connective for an invariant $\mathsf{inv}\, P \in \mathit{iProp}$ that takes *syntactic data* for a proposition $P \in \mathit{nProp}$ instead of a *semantic* proposition $P \in \mathit{iProp}$. The proof rules for using the invariant depend on the interpretation $[\![\ ]\!]$ of the syntactic propositions *nProp*. In particular, we have developed later-free versions of *shared invariants* by Iris (Jung et al., 2015) and *Rust-style borrows* by RustBelt's *lifetime logic* (Jung et al., 2018a; Jung, 2020).

For example, Nola provides the following *later-free* proof rules for accessing an invariant, improving on the original rules THOARE-IINV and IINV-ACC:

$$\frac{\big[[\![P]\!] * Q\big]\; e \;\big[\lambda v.\, [\![P]\!] * \Psi\, v\big]'}{\big[\mathsf{inv}\, P * Q\big]\; e \;\big[\Psi\big]'}\ \textsc{thoare-inv}$$

$$\frac{[\![P]\!] * Q \vDash \Rrightarrow' \big([\![P]\!] * R\big)}{\mathsf{inv}\, P * Q \vDash \Rrightarrow' R}\ \textsc{inv-acc}$$

We get the *semantic interpretation* $[\![P]\!] \in \mathit{iProp}$ of the syntactic proposition $P \in \mathit{nProp}$ of the invariant, instead of $\triangleright P$ in the original rules THOARE-IINV and INV-ACC. Here, we use a new total Hoare triple $\big[P\big]\; e \;\big[\Psi\big]'$ and a new fancy update $\Rrightarrow'$ for Nola's invariants.

Power of Parameterization Our framework supports a *generalized* setting, in that the mechanisms for propositional sharing are *parameterized* over the choice of the syntactic data type for propositions *nProp* and its interpretation $[\![\ ]\!] \colon \mathit{nProp} \to \mathit{iProp}$. This parameterization has the following advantages.

From a practical viewpoint, the parameterization brings customizability and extensibility. The propositions one wants to store in mechanisms for propositional sharing, such as invariants and borrows, depend on the goal of verification. Thanks to the parameterization, one can freely instantiate the parameters *nProp* and $[\![\ ]\!]$ according to one's verification purpose. When one wants to support a new class of propositions to be stored, one just needs to extend *nProp* and $[\![\ ]\!]$ with that class.

From a theoretical viewpoint, the parameterization clarifies which logical connectives can be safely stored in the mechanisms for propositional sharing and which logical connectives are problematic. In particular, we show that arbitrarily *nesting* of the invariant connective admits later-free proof rules, which is a new achievement (see § 9.1

makes the later modality unnecessary. The other is 'NOLA', a nickname for New Orleans, Louisiana. This city was the site of ACM POPL 2020, the last top PL conference held before the COVID pandemic. I attended the conference during my first year as a master's student and had a great time interacting with many fascinating people, having delicious seafood, listening to a street jazz band, and so on. I named the framework after the city's nickname partly out of nostalgia for the days there.

for comparison with existing work). On the other hand, we observe that (unguarded) *impredicative quantifiers* cannot be directly stored in the mechanisms for propositional sharing. We analyze the source of the paradox of Landin's knot (1.17) (§ 1.4) and the known paradox by Krebbers et al. (2017a, § 5) (explained in § 3.4.1) as a *Hoare triple* $[P]\ e\ [\Psi]$ or *fancy update* $\Rrightarrow$ stored in an invariant, which internally uses an impredicative quantifier (see § 3.4.2 for details).

Semantic Alteration The logic connectives for propositional sharing like $\mathsf{inv}\, P$ are naively based on the *syntactic agreement* of propositions *nProp*. But we often want to *alter* the content propositions P of these connectives in a *semantic* way. A traditional approach tackled this by constructing a predicate for alteration *syntactically*, considering all the wanted proof rules *upfront*. But that is not quite acceptable. Remarkably, we have found a novel, general construction of such a predicate one can obtain *for free* once one builds *semantics*. The key idea is to *parameterize* the semantics of propositions $[\![\ ]\!] \colon \mathit{nProp} \to \mathit{iProp}$ with an undecided predicate δ to be used for alteration.

Origin of Nola's Core Idea The core idea of Nola, using *syntax* of separation logic propositions for propositional sharing, was born through the author's experience in the prior work, RustHorn (Matsushita, 2019; Matsushita et al., 2020, 2021) and RustHornBelt (Matsushita, 2021; Matsushita et al., 2022).

RustHorn proposed a new verification method for Rust programs, with a novel idea of using prophecies to model Rust's mutable borrows, and formally proved the correctness of the method for a small core of Rust (see § 7.1.1 for details). The proof naturally took advantage of the *syntactic* nature of Rust's type system and did not require the later modality at all.

Later, RustHornBelt extended RustBelt (Jung et al., 2018a) to get a modular, extensible proof of RustHorn's method for a large subset of Rust, including APIs such as `Vec` and `Mutex`, by *semantically* modeling Rust's ownership types and RustHorn-style functional specifications in the separation logic Iris (see § 7.1.2 for details). However, RustHornBelt's proof suffered from the *later modality* for invariants and borrows, which posed fundamental difficulties in supporting liveness properties.

The author was unsatisfied with the later modality in RustHornBelt's proof, which was not present in RustHorn's proof. As a substitute for RustHornBelt's approach, the author once considered introducing a highly general ownership type system that supports low-level verification like separation logic and building a syntactic soundness proof of the type system, which would naturally avoid the later modality. However, the author felt that a fully syntactic approach would not be scalable. Later on, after consideration in this direction, the author came up with the core idea of Nola, i.e., to use *syntax* for the separation logic propositions for propositional sharing and still do low-level verification in *semantic* separation logic.

1.5.1 Our Contributions

Our contributions can be summarized as follows.

- We propose a novel, general framework, *Nola*, that provides advanced mechanisms for propositional sharing with *later-free* proof rules in separation logic. Its key idea is to *isolate* the *syntactic* data type *nProp* for the separation logic propositions to be stored from its *semantic* interpretation $[\![\ ]\!] \colon \mathit{nProp} \to \mathit{iProp}$. Our proof rules for propositional sharing mechanisms are generalized over the choice of *nProp* and $[\![\ ]\!]$.

For our framework Nola, we have developed later-free versions of propositional sharing mechanisms for *shared invariants* by Iris (Jung et al., 2015, 2018b) (§ 3.2) and *Rust-style borrows* by RustBelt's lifetime logic (Jung et al., 2018a) (Chapter 6).

- We have discovered a novel, general technique for *semantic alteration* of the content propositions of the logic connectives for propositional sharing (Chapter 4).

- We have demonstrated the power of our Nola framework with non-trivial examples. We verified the strong normalization under a type system that supports higher-order reference types (Chapter 5). Also, we constructed a new general mechanism called *prophetic borrows* that refines RustHornBelt (Matsushita et al., 2022)'s approach (Chapter 7).

- We have found a *new paradox of later-free invariants*, which significantly simplifies the existing one by Krebbers et al. (2017a, § 5), showing that a later-free invariant leads to a contradiction even in the absence of nested invariants or impredicative quantifiers (§ 3.4.1).

  We also analyze that the real source of the paradox is the fancy update modality directly stored in the invariant, which is naturally avoided in this framework, and arbitrarily nesting of invariants is not problematic at all.

- We have fully mechanized our framework on top of the Iris separation logic framework in the Coq Proof Assistant (Coq Team, 2023), making it easy to combine it with existing Iris developments. We have also mechanized the case studies discussed in this dissertation.

1.5.2 Future Applications

Before diving into the technical details, we mention some possible future applications of our work, Nola.

- By insights from our work, the invariant and borrow can possibly be safely integrated into first-order[25] separation-logic-based automated verification platforms such as VeriFast (Jacobs et al., 2011) and Viper (Müller et al., 2016), being free from the later modality and naturally supporting liveness verification.

- RustHornBelt (Matsushita et al., 2022), a semantic foundation for prophetic functional Rust verifiers such as RustHorn (Matsushita et al., 2020) and Creusot (Denis et al., 2022), can possibly be rebuilt on Nola to eliminate the need for the later modality and support liveness verification.

- Simuliris (Gäher et al., 2022) can possibly be used to verify optimizations under the guarantee of ownership types modeled with Nola's invariant and borrow, especially unifying the approaches of Stacked Borrows (Jung et al., 2020a), a memory model designed for Rust's borrows, and RustBelt (Jung et al., 2018a), a semantic foundation for Rust's ownership types.

1.5.3 Dissertation Organization

The rest of this dissertation is organized as follows.

- Chapter 2 presents the technical preliminaries on the Iris separation logic framework, introducing Iris's core features and resource customization.

[25] By 'first-order', we roughly mean that Hoare triples etc. are not supported as first-class formulas in the logic. See § 3.4.2 for limitations in supporting higher-order features.

- Chapter 3 presents an overview of the design principle and usage of our framework, Nola, focusing on the later-free *shared invariant* mechanism. We present the interface of our invariant mechanism and show how to use them through a verification example of iterative mutation of a shared mutable singly linked list. We also present our new paradox of later-free invariants and discuss the general expressivity of the framework and how our framework naturally avoids paradoxes.

- Chapter 4 presents a novel, general technique for *semantic alteration* of the content propositions of the logical connectives for propositional sharing.

- Chapter 5 presents a case study of our framework, verifying *strong normalization* (i.e., termination) of functional programs under a stratified type system that supports higher-order references.

- Chapter 6 presents the later-free Rust-style *borrow* mechanism of our framework. The mechanism allows non-step-indexed separation logic to support the features of RustBelt's lifetime logic (Jung et al., 2018a), which can model and reason about Rust-style borrows in a general, semantic way.

- Chapter 7 presents the later-free *prophetic borrow* mechanism of our framework, built on our borrow mechanism. Our prophetic borrow mechanism abstracts and refines RustHornBelt (Matsushita et al., 2022)'s reasoning approach to functional verification about Rust-style borrows with RustHorn-style *prophecies*.

- Chapter 8 reports on our mechanization of the Nola framework in the Coq Proof Assistant.

- Chapter 9 discusses existing work related to this dissertation.

- Chapter 10 concludes the dissertation.

Chapter 2

Technical Preliminaries on Iris

下學而上達
Study below to reach up

Confucius, the *Analects*

This chapter presents the technical preliminaries on the Iris separation logic framework. Readers can skip it when first reading this dissertation and return to relevant parts to understand the details. Section 2.1 presents Iris's core features and Section 2.2 presents resource customization. For a better understanding of Iris, we also recommend referring to a journal paper (Jung et al., 2018b), a lecture note (Birkedal and Bizjak, 2023), a high-level introduction (Jung, 2020, Part I), the user reference (Iris Team, 2023a), and the Coq development (Iris Team, 2023b).

2.1 Iris's Core Features

Now we present Iris's core features. The set of Iris propositions, i.e., propositions in the Iris separation logic, is *iProp*. We write P, Q, R for metavariables of Iris propositions ranging over *iProp*.

2.1.1 Basic Features

Entailment The *entailment* $P \vDash Q$ on Iris propositions $P, Q \in iProp$ forms a partial order:

$$P \vDash P \quad \vDash\text{-REFL} \qquad \frac{P \vDash Q \quad Q \vDash R}{P \vDash R} \quad \vDash\text{-TRANS} \qquad \frac{P \vDash Q \quad Q \vDash P}{P = Q} \quad \vDash\text{-ANTISYM}$$

Also, we write $\vDash P$ for the entailment $\top \vDash P$ from the truth $\top \in iProp$ (introduced just later) to an Iris proposition $P \in iProp$, which is equivalent to $\top = P$.

Intuitionistic Connectives Iris propositions have all the connectives for intuitionistic predicate logic: the *truth* $\top$, *falsefood* $\bot$, *conjunction* $P \land Q$, *disjunction* $P \lor Q$, *implication* $P \to Q$, *universal quantification* $\forall a \in A.\, P_a$, and *existential quantification* $\exists a \in A.\, P_a$. The domain A of (universal/existential) quantification can be a set of *any universe*, including the set of Iris propositions *iProp*, meaning that quantification on *iProp* is *impredicative*. Iris satisfies all proof rules of intuitionistic predicate logic.[1] In particular, the following rules hold:

$$P \vDash \top \quad \top\text{-INTRO} \qquad \bot \vDash P \quad \bot\text{-ELIM}$$

[1] Technically, this is because these logical connectives are modeled based on *Kripke semantics* of intuitionistic predicate logic.

$$\frac{R \vDash P \quad R \vDash Q}{R \vDash P \land Q}\ \textsc{$\land$-intro} \qquad P \land Q \vDash P \ \textsc{$\land$-elim}_\textsc{l} \qquad P \land Q \vDash Q \ \textsc{$\land$-elim}_\textsc{r}$$

$$P \vDash P \lor Q \ \textsc{$\lor$-intro}_\textsc{l} \qquad Q \vDash P \lor Q \ \textsc{$\lor$-intro}_\textsc{r} \qquad \frac{P \vDash R \quad Q \vDash R}{P \lor Q \vDash R}\ \textsc{$\lor$-elim}$$

$$\frac{R \land P \vDash Q}{R \vDash P \to Q}\ \textsc{$\to$-intro} \qquad (P \to Q) \land P \vDash Q \ \textsc{$\to$-elim}$$

$$\frac{\forall a \in A.\ \big(Q \vDash P_a\big)}{Q \vDash \forall a \in A.\, P_a}\ \textsc{$\forall$-intro} \qquad \frac{a_0 \in A}{(\forall a \in A.\, P_a) \vDash P_{a_0}}\ \textsc{$\forall$-elim}$$

$$\frac{a_0 \in A}{P_{a_0} \vDash \exists a \in A.\, P_a}\ \textsc{$\exists$-intro} \qquad \frac{\forall a \in A.\ \big(P_a \vDash Q\big)}{(\exists a \in A.\, P_a) \vDash Q}\ \textsc{$\exists$-elim}$$

Pure-Proposition Embedding Any *pure* proposition $\phi \in \mathit{Prop}$ can be *embedded* into an Iris proposition $\ulcorner\phi\urcorner \in \mathit{iProp}$. Intuitionistic connectives commute with pure-proposition embedding:

$$\ulcorner\top\urcorner = \top \ \textsc{pure-$\top$} \qquad \ulcorner\bot\urcorner = \bot \ \textsc{pure-$\bot$}$$

$$\ulcorner\phi \land \psi\urcorner = \ulcorner\phi\urcorner \land \ulcorner\psi\urcorner \ \textsc{pure-$\land$} \qquad \ulcorner\phi \lor \psi\urcorner = \ulcorner\phi\urcorner \lor \ulcorner\psi\urcorner \ \textsc{pure-$\lor$}$$

$$\ulcorner\phi \to \psi\urcorner = \ulcorner\phi\urcorner \to \ulcorner\psi\urcorner \ \textsc{pure-$\to$}$$

$$\ulcorner\forall a \in A.\, \phi_a\urcorner = \forall a \in A.\, \ulcorner\phi_a\urcorner \ \textsc{pure-$\forall$} \qquad \ulcorner\exists a \in A.\, \phi_a\urcorner = \exists a \in A.\, \ulcorner\phi_a\urcorner \ \textsc{pure-$\exists$}$$

We usually omit $\ulcorner - \urcorner$ and just write ϕ for $\ulcorner\phi\urcorner$ in Iris propositions.

Separating Conjunction The key logical connective in Iris is the *separating conjunction* $P * Q$. It asserts the ownership and knowledge of two *disjoint* fragments of mutable state respectively owned by P and Q. The logical connective is *monotone*, *commutative*, *associative*, and *unital* by the truth $\top$:

$$\frac{P \vDash P' \quad Q \vDash Q'}{P * Q \vDash P' * Q'}\ \textsc{$*$-mono}$$

$$P * Q = Q * P \ \textsc{$*$-comm} \qquad (P * Q) * R = P * (Q * R) \ \textsc{$*$-assoc}$$

$$P * \top = P \ \textsc{$*$-unit}$$

The Iris separation logic is *affine* in the sense that it satisfies the following rule for eliminating the separating conjunction, or technically *left weakening* (by $*$-unit):

$$P * Q \vDash P \ \textsc{$*$-elim}$$

The affineness means that any fragment of ownership can be *leaked* at any time. Iris adopts this design especially because Iris's invariant mechanism makes it impossible to ensure the absence of ownership leaks (Jung, 2020, § 6.2). We should substantially modify the logic and the design of the invariant mechanism to ensure the absence of ownership leaks (Bizjak et al., 2019).

Also note that the separating conjunction can be weakened into the usual conjunction (by $*$-elim):

$$P * Q \vDash P \land Q \ \textsc{$*$-$\land$}$$

Separating Implication The *separating implication* (a.k.a. *magic wand*) $P \mathrel{-\!\!*} Q$ is to the separating conjunction $*$ what the usual implication $\to$ is to the usual conjunction

$\land$. More formally, the separating implication $-\!\!*$ is the *right adjoint* of the separating conjunction $*$, satisfying the following rules analogous to $\to$-INTRO and $\to$-ELIM:

$$\frac{R * P \vDash Q}{R \vDash P -\!\!* Q}\ -\!\!*\text{-INTRO} \qquad (P -\!\!* Q) * P \vDash Q \quad -\!\!*\text{-ELIM}$$

In other words, the separating implication $P -\!\!* Q$ is the *weakest* of the propositions R satisfying the property $R * P \vDash Q$, i.e., what can turn into Q when separately conjoined with P. Roughly speaking, $P -\!\!* Q$ is something like "Q minus P".

The separating implication $-\!\!*$ is weaker than the usual implication $\to$ by $*$-$\land$:

$$P \to Q \vDash P -\!\!* Q \quad \to\text{-}-\!\!*$$

The existence of the separating implication makes the separating conjunction preserve the existential quantification, which is an instance of the 'left adjoints preserve colimits' rule well-known in category theory:

$$P * \exists a \in A.\, Q_a \;=\; \exists a \in A.\ P * Q_a \quad *\text{-}\exists$$

Persistence Modality Iris has the *persistence modality* $\square P$, which roughly means that the assertion P holds without any *exclusive* ownership. The key rule is the following, strengthening the usual conjunction $\land$ into the separating conjunction $*$ when one conjunct is under the persistence modality $\square$:

$$\square P \land Q \;=\; \square P * Q \quad \square\text{-}\land\text{-}*$$

This roughly means that the ownership owned by a proposition under the persistence modality $\square P$ is *disjoint* from any kind of ownership. Also, the persistence modality forms a *comonad*, satisfying the following rules:

$$\frac{P \vDash Q}{\square P \vDash \square Q}\ \square\text{-MONO} \qquad \square P \vDash P \quad \square\text{-ELIM} \qquad \square P \;=\; \square\square P \quad \square\text{-IDEMP}$$

The persistence modality commutes with the conjunction $\land$, disjunction $\lor$, universal quantification $\forall$, existential quantification $\exists$, pure-proposition embedding $\ulcorner - \urcorner$, and separating conjunction $*$:

$$\square P \land \square Q \;=\; \square(P \land Q) \quad \square\text{-}\land \qquad \square(P \lor Q) \;=\; \square P \lor \square Q \quad \square\text{-}\lor$$

$$\forall a \in A.\ \square P_a \;=\; \square(\forall a \in A.\, P_a) \quad \square\text{-}\forall \qquad \square(\exists a \in A.\, P_a) \;=\; \exists a \in A.\ \square P_a \quad \square\text{-}\exists$$

$$\square \ulcorner \phi \urcorner \;=\; \ulcorner \phi \urcorner \quad \square\text{-PURE} \qquad \square P * \square Q \;=\; \square(P * Q) \quad \square\text{-}*$$

Also, under the persistence modality $\square$, the usual conjunction $\land$ coincide with the separating conjunction $*$ and the separating implication $-\!\!*$ coincide with the usual implication $\to$:

$$\square(P \land Q) \;=\; \square(P * Q) \quad \text{IN-}\square\text{-}\land\text{-}* \qquad \square(P -\!\!* Q) \;=\; \square(P \to Q) \quad \text{IN-}\square\text{-}-\!\!*\text{-}\to$$

Parsistent Propositions We say a proposition P is *persitent* if it is equal to $\square P$, i.e., if we can introduce the persistence modality $\square$ on P:

$$P \text{ is persistent} \quad \triangleq \quad P \;=\; \square P$$

It roughly means that the assertion P holds without any exclusive ownership.

The usual conjunction $\land$ coincides with the separating conjunction $*$ when one conjunct is persistent by $\Box$-$\land$-$*$:

$$\frac{P \text{ is persistent}}{P \land Q \;=\; P * Q} \quad \textsc{persist-}\land\textsc{-}*$$

There are several consequences of $\textsc{persist-}\land\textsc{-}*$. We can *duplicate* a persistent proposition P into $P * P$. Entailment can *retain* the premise when the conclusion is persistent. The separating implication $\mathrel{-\!\!*}$ coincides with the usual implication $\to$ when the input is persistent.

$$\frac{P \text{ is persistent}}{P \;\vDash\; P * P} \quad \textsc{persist-dup} \qquad\qquad \frac{P \vDash Q \quad Q \text{ is persistent}}{P \;\vDash\; P * Q} \quad \textsc{persist-retain}$$

$$\frac{P \text{ is persistent}}{P \mathrel{-\!\!*} Q \;=\; P \to Q} \quad \textsc{persist-}\mathrel{-\!\!*}\textsc{-}\to$$

Also, we can introduce the persistence modality $\Box$ on an entailment whose premise is persistent, by $\Box$-MONO:

$$\frac{P \vDash Q \quad P \text{ is persistent}}{P \;\vDash\; \Box Q} \quad \textsc{persist-}\Box\textsc{-intro}$$

The following rules for finding persistent propositions can be derived:

$$\Box P \text{ is persistent} \quad \Box\textsc{-persist} \qquad\qquad \ulcorner \phi \urcorner \text{ is persistent} \quad \textsc{pure-persist}$$

$$\frac{P \text{ and } Q \text{ are persistent}}{P \land Q \text{ is persistent}} \quad \land\textsc{-persist} \qquad\qquad \frac{P \text{ and } Q \text{ are persistent}}{P \lor Q \text{ is persistent}} \quad \lor\textsc{-persist}$$

$$\frac{\forall a \in A.\ \big(P_a \text{ is persistent}\big)}{(\forall a \in A.\, P_a) \text{ is persistent}} \quad \forall\textsc{-persist} \qquad\qquad \frac{\forall a \in A.\ \big(P_a \text{ is persistent}\big)}{(\exists a \in A.\, P_a) \text{ is persistent}} \quad \exists\textsc{-persist}$$

$$\frac{P \text{ and } Q \text{ are persistent}}{P * Q \text{ is persistent}} \quad *\textsc{-persist}$$

Basic Update Iris has the *basic update modality* $\dot{\Rrightarrow} P$, which roughly means that the assertion P holds after internal update of the *resources*. The modality forms a *monad*:

$$\frac{P \vDash Q}{\dot{\Rrightarrow} P \;\vDash\; \dot{\Rrightarrow} Q} \quad \dot{\Rrightarrow}\textsc{-mono} \qquad P \;\vDash\; \dot{\Rrightarrow} P \quad \dot{\Rrightarrow}\textsc{-intro} \qquad \dot{\Rrightarrow}\dot{\Rrightarrow} P \;=\; \dot{\Rrightarrow} P \quad \dot{\Rrightarrow}\textsc{-idemp}$$

Also, the monad $\dot{\Rrightarrow}$ is *strong* with respect to the *separating conjunction* $*$:

$$(\dot{\Rrightarrow} P) \;*\; Q \;\vDash\; \dot{\Rrightarrow}(P * Q) \quad \dot{\Rrightarrow}\textsc{-frame}$$

This can be understood as a primitive variant of the *frame rule* for the Hoare triple (PHOARE-FRAME in § 1.2.3).

Also, the basic update modality on a pure proposition can be stripped off:

$$\dot{\Rrightarrow} \ulcorner \phi \urcorner \;=\; \ulcorner \phi \urcorner \quad \dot{\Rrightarrow}\textsc{-pure}$$

This property is not satisfied by the *fancy update* $\Rrightarrow$ introduced later in § 3.1.

As explained later in § 2.2.1, the basic update works nicely with the *resource ownership* proposition $\boxed{o}^{\gamma}$.

2.1.2 Around the Later Modality

Later Modality The *later modality* $\triangleright P$ is an ill-behaved modality and our dissertation is mainly about how to eliminate the need for it, as explained in § 1.4. The later modality $\triangleright P$ originates from *indexing* of the semantics of the logic. Roughly speaking, $\triangleright P$ means that the assertion P holds in the *next* index. The later modality is monotone and weakens a proposition:

$$\frac{P \vDash Q}{\triangleright P \vDash \triangleright Q}\ \triangleright\text{-MONO} \qquad P \vDash \triangleright P\ \ \triangleright\text{-INTRO}$$

However, it is *not idempotent*:

$$\triangleright\triangleright P \neq \triangleright P$$

Iris also has the following *Löb induction* rule, which generalizes LÖB introduced in § 1.4:

$$(\triangleright P) \to P \;=\; P \quad \text{LÖB-GEN}$$

The later modality commutes with the truth, conjunction, disjunction, universal quantification, *inhabited* existential quantification, separating conjunction, and persistence modality:

$$\triangleright\top = \top \quad \triangleright\text{-}\top$$

$$(\triangleright P) \land (\triangleright Q) = \triangleright(P \land Q) \quad \triangleright\text{-}\land \qquad \triangleright(P \lor Q) = (\triangleright P) \lor (\triangleright Q) \quad \triangleright\text{-}\lor$$

$$\forall a \in A.\ \triangleright P_a = \triangleright(\forall a \in A.\ P_a) \quad \triangleright\text{-}\forall \qquad \frac{A \neq \varnothing}{\triangleright(\exists a \in A.\ P_a) = \exists a \in A.\ \triangleright P_a}\ \triangleright\text{-}\exists$$

$$\triangleright(P * Q) = \triangleright P * \triangleright Q \quad \triangleright\text{-}* \qquad \triangleright\Box P = \Box\triangleright P \quad \triangleright\text{-}\Box$$

However, it does *not* commute with the *basic update* modality $\dot{\Rrightarrow}$:

$$\triangleright\dot{\Rrightarrow} P \nvDash \dot{\Rrightarrow}\triangleright P \qquad \dot{\Rrightarrow}\triangleright P \nvDash \triangleright\dot{\Rrightarrow} P$$

Except-0 Modality The *except-0 modality* $\diamond P$ is defined as follows:

$$\diamond P \;\triangleq\; P \lor \triangleright\bot$$

Roughly speaking, it means that P holds for all indices except for the index 0, i.e., the initial index.

The except-0 modality forms a monad:

$$\frac{P \vDash Q}{\diamond P \vDash \diamond Q}\ \diamond\text{-MONO} \qquad P \vDash \diamond P\ \ \diamond\text{-INTRO} \qquad \diamond\diamond P = \diamond P\ \ \diamond\text{-IDEMP}$$

The except-0 modality is stronger than the later modality. Also, the except-0 modality is absorbed by the later modality.

$$\diamond P \vDash \triangleright P \quad \diamond\text{-}\triangleright \qquad \diamond\triangleright P = \triangleright P \quad \diamond\text{-}\triangleright\text{-ABSORB}$$

Timeless Propositions We say a proposition P is *timeless* if $\triangleright P$ is equal to $\diamond P$:

$$P \text{ is timeless} \;\triangleq\; \triangleright P = \diamond P$$

It means in the indexed semantics that P is constant over the indices.

The following rules for finding timeless propositions can be derived:

$$\ulcorner\phi\urcorner \text{ is timeless} \quad \text{PURE-TIMELESS}$$

$$\frac{P \text{ and } Q \text{ are timeless}}{P \wedge Q \text{ is timeless}}\ \wedge\text{-TIMELESS} \qquad \frac{P \text{ and } Q \text{ are timeless}}{P \vee Q \text{ is timeless}}\ \vee\text{-TIMELESS}$$

$$\frac{Q \text{ is timeless}}{P \to Q \text{ is timeless}}\ \to\text{-TIMELESS}$$

$$\frac{\forall a \in A.\ \big(P_a \text{ is timeless}\big)}{(\forall a \in A.\, P_a) \text{ is timeless}}\ \forall\text{-TIMELESS} \qquad \frac{\forall a \in A.\ \big(P_a \text{ is timeless}\big)}{(\exists a \in A.\, P_a) \text{ is timeless}}\ \exists\text{-TIMELESS}$$

$$\frac{P \text{ and } Q \text{ are timeless}}{P * Q \text{ is timeless}}\ *\text{-TIMELESS} \qquad \frac{Q \text{ is timeless}}{P \mathrel{-\!\!*} Q \text{ is timeless}}\ \mathrel{-\!\!*}\text{-TIMELESS}$$

$$\frac{P \text{ is timeless}}{\square P \text{ is timeless}}\ \square\text{-TIMELESS}$$

Unfortunately, the *basic update* $\dot{\Rrightarrow} P$ is *not* timeless even if P is timeless.

Guarded Recursion A mapping $f : A \longrightarrow B$ between indexed sets A, B is said to be *contractive* if the indexed equality of inputs weakened by the *later modality* $\triangleright(a \mathrel{\tilde{=}} a')$ ensures the indexed equality of outputs $f\ a \mathrel{\tilde{=}} f\ a'$. The *later modality* $\triangleright : iProp \longrightarrow iProp$ is a typical example of a contractive mapping. Contractiveness is significant because it guarantees the existence of a unique fixed point.

Theorem 2.1 (Banach's Fixed Point Theorem)**.** *Any* contractive *mapping* $f : A \longrightarrow A$ *has a* unique *fixed point* $\operatorname{rec} f \in A$, *satisfying* $\operatorname{rec} f \ = \ f\,(\operatorname{rec} f)$.

Recursion whose self-reference is *contractive* is called *guarded recursion*. For a small but tricky example, the following definition of the proposition $\mathsf{lliar} \in iProp$ is *well-defined* (despite the similarity to the liar's paradox), because it is guarded recursion thanks to the guard by the later modality $\triangleright$:

$$\mathsf{lliar} \quad \triangleq \quad (\triangleright \mathsf{lliar}) \to \bot$$

Actually the proposition lliar thus defined is equal to the falsehood $\bot$, because $(\triangleright \bot) \to \bot$ is equal to $\bot$ by Löb induction LÖB-GEN.

2.2 Resources

Separation logic can reason about *ownership*. In a traditional setting, separation logic is just about ownership of *heap memory cells*, and the separation logic proposition *iProp* is modeled as a predicate $Heap \longrightarrow Prop$ over a heaplet $Heap \triangleq Loc \overset{\text{fin}}{\rightharpoonup} Val$ (1.7). But in modern settings, we want to reason about ownership of *various kinds of resources*, not just heap memory cells.

In Iris, resources are mathematically modeled as an intricate *indexed* algebra, nicknamed the *camera*.[2] The carrier $\llcorner\tilde{\mathcal{A}}\lrcorner$ of a camera $\tilde{\mathcal{A}}$ is an *indexed* set (a.k.a. ordered family of equivalences). Because a camera is rather hard to understand due to indexing, we instead present here a *non-indexed*, *discrete* version of the camera, called the *resource algebra*, abbreviated as *RA*. Any resource algebra $\mathcal{A}$ can be seen as a camera. All the resources for propositional sharing provided by our Nola framework are modeled as a resource algebra.

We first present a general topic about the resource algebra and the resource ownership connective $\boxed{o}^{\gamma}_{\mathcal{A}} \in iProp$ (§ 2.2.1) and then present various constructions of resource algebras (§ 2.2.2). We also discuss the example of the heap resource algebra HEAP for reasoning about heap memory (§ 2.2.3).

2 The name camera used to be a nickname of CMRA, an acronym for "complete metric resource algebra". At that time, Iris's camera required the completeness of the carrier. However, later versions of Iris dropped the completeness requirement from its camera and still kept the nickname camera for continuity.

2.2.1 Resource Algebra and Resource Ownership

Resource Algebra A *resource algebra (RA)* $\mathcal{A}$ consists of the *carrier set* $\llcorner\mathcal{A}\lrcorner$, the *product* operation $\cdot_{\mathcal{A}} \colon \llcorner\mathcal{A}\lrcorner \times \llcorner\mathcal{A}\lrcorner \to \llcorner\mathcal{A}\lrcorner$, the *core* operation $|-|_{\mathcal{A}} \colon \llcorner\mathcal{A}\lrcorner \rightharpoonup \llcorner\mathcal{A}\lrcorner$, the *validity* predicate $\checkmark_{\mathcal{A}} \colon \llcorner\mathcal{A}\lrcorner \to \mathit{Prop}$, satisfying the following rules (we omit the subscript $\mathcal{A}$ and use the metavariable o for an element of $\llcorner\mathcal{A}\lrcorner$):

$$o \cdot o' \;=\; o' \cdot o \quad \cdot\text{-COMM} \qquad (o \cdot o') \cdot o'' \;=\; o \cdot (o' \cdot o'') \quad \cdot\text{-ASSOC}$$

$$\frac{|o| \text{ is defined}}{|o| \cdot o = o}\ \text{CORE-ABSORB} \qquad \frac{|o| \text{ is defined}}{||o|| = |o|}\ \text{CORE-IDEMP}$$

$$\frac{|o| \text{ is defined}}{\exists o''.\ |o \cdot o'| = |o| \cdot o''}\ \text{CORE-}\cdot \qquad \frac{\checkmark(o \cdot o')}{\checkmark\, o}\ \checkmark\text{-}\cdot$$

The *carrier set* (a.k.a. *underlying set*) $\llcorner\mathcal{A}\lrcorner$ is the set of *resource* representations considered by the resource algebra $\mathcal{A}$.

The *product* $o \cdot o'$ is a binary operation over resources $o, o' \in \llcorner\mathcal{A}\lrcorner$, which corresponds to the *separating conjunction* $P * Q$ (see OWN-$\cdot$-$*$ shown later). The product is required to be commutative ($\cdot$-COMM) and associative ($\cdot$-ASSOC), to make the separating conjunction $*$ commutative ($*$-COMM) and associative ($*$-ASSOC). Note that the product of a resource algebra is *total*, unlike the disjoint union of heaplets $H_1 + H_2$ we considered in the semantics of (1.8) (§ 1.2.3). This is because a resource algebra separately has the *validity* predicate $\checkmark$.

The *core* $|o|$ is a *partial* unary operation over resources $o \in \llcorner\mathcal{A}\lrcorner$, which corresponds to the *persistence modality* $\square P$ (see OWN-CORE-PERSIST shown later). The core $|o|$ of a resource o is absorbed by o (CORE-ABSORB). The core operation is idempotent (CORE-IDEMP). The core $|o \cdot o'|$ of a product $o \cdot o'$ includes the core $|o|$ of an operand o of the product (CORE-$\cdot$).

The *validity* predicate $\checkmark o$ is a unary predicate over resources $o \in \llcorner\mathcal{A}\lrcorner$. A resource owned by a separation logic proposition $P \in \mathit{iProp}$ is always valid (see OWN-$\checkmark$ later). If a product $o \cdot o'$ is valid, then an operand o of it is also valid ($\checkmark$-$\cdot$).

We can derive the *inclusion* relation $o \lesssim_{\mathcal{A}} o'$ on resources $o, o' \in \llcorner\mathcal{A}\lrcorner$, meaning that o is a part of o':[3]

$$o \lesssim o' \quad \triangleq \quad \exists o^?_+.\ o' = o \cdot o^?_+$$

Here, $o^?_+$ is either in $\llcorner\mathcal{A}\lrcorner$ or undefined. The inclusion relation forms a pre-order, i.e., is reflexive and transitive.

A *unital resource algebra* is a resource algebra $\mathcal{A}$ with the *unit* resource $\varepsilon_{\mathcal{A}} \in \llcorner\mathcal{A}\lrcorner$, satisfying the following rules:

$$o \cdot \varepsilon = o \quad \cdot\text{-}\varepsilon \qquad |\varepsilon| = \varepsilon \quad \text{CORE-}\varepsilon \qquad \checkmark\,\varepsilon \quad \checkmark\text{-}\varepsilon$$

The unit ε is the unit element of the product ($\cdot$-ε). The core of the unit is the unit itself (CORE-ε). The unit is valid ($\checkmark$-ε). A resource algebra is not unital by default because we want to support the notion of *exclusive* resources, introduced later.

Resource Update The *resource update* relation $o \rightsquigarrow^{\in}_{\mathcal{A}} O'$ between a resource $o \in \llcorner\mathcal{A}\lrcorner$ and a set of resources $O' \subseteq \llcorner\mathcal{A}\lrcorner$ is defined as follows:

$$o \rightsquigarrow^{\in} O' \quad \triangleq \quad \forall o^?_f \text{ s.t. } \checkmark(o \cdot o^?_f).\ \exists o' \in O'.\ \checkmark(o' \cdot o^?_f)$$

Intuitively, it means that when we have a resource o as a part of the 'global' resource $o \cdot^? o^?_f$, we can update that part into a resource o' in the set O', without violating the

[3] The product $o_1 \cdot o^?_2$ is defined as $o_1 \cdot o^?_2$ if $o^?_2$ is defined and as o_1 if $o^?_2$ is not.

validity invariant $\checkmark$ of the global resource. Here, $o_f^?$ represents the *frame*, either in $\llcorner\mathcal{A}\lrcorner$ or undefined.

As a special case, we write $o \rightsquigarrow_{\mathcal{A}} o'$ for $o \rightsquigarrow_{\mathcal{A}} \{o'\}$.

The resource update corresponds to the *basic update* modality $\dot{\Rrightarrow} P$ (see OWN-$\rightsquigarrow^{\in}$ shown later).

We say a resource $o \in \mathcal{A}$ is *exclusive* if the following holds:

$$o \text{ is exclusive} \quad \triangleq \quad \forall o' \in \llcorner\mathcal{A}\lrcorner.\ \neg \checkmark(o \cdot o')$$

Intuitively, it means that having the resource o allows *no frame*. For an exclusive resource o, we can update it into any valid resource o':

$$\frac{o \text{ is exclusive} \qquad \checkmark o'}{o \rightsquigarrow o'} \ \textsc{exclus-}\rightsquigarrow$$

Component Cameras The separation logic proposition *iProp* in Iris is *parameterized* over the *family of component cameras* $(\tilde{\mathcal{A}}_i)_{i \in I}$ for some index set I.[4]

Technically, the *global camera* $\tilde{\mathcal{G}}$ for the Iris proposition *iProp* is defined as follows given the list of component cameras:

$$\tilde{\mathcal{G}} \ \triangleq\ \textstyle\prod_{i \in I} \left(\mathit{GhostName} \overset{\text{fin}}{\rightharpoonup} \tilde{\mathcal{A}}_i \right).$$

It is the product of the finite map camera $\mathit{GhostName} \overset{\text{fin}}{\rightharpoonup} \tilde{\mathcal{A}}_i$ to each component camera $\tilde{\mathcal{A}}_i$. The product and finite map cameras are defined in a way analogous to the product and finite map RAs, which are explained later in § 2.2.2. The set *GhostName* is countably infinite and its elements $\gamma \in \mathit{GhostName}$ are called a *ghost name*.[5]

The Iris proposition *iProp* is semantically modeled as, roughly speaking, a monotone predicate over the valid elements of the global camera $\tilde{\mathcal{G}}$:

$$\mathit{iProp} \quad \text{is roughly} \quad \left\{ o \in \tilde{\mathcal{G}} \;\middle|\; \tilde{\checkmark}\, o \right\} \xrightarrow{\text{mono}} \widetilde{\mathit{Prop}}.$$

The precise definition is a bit more subtle because a camera's validity predicate $\tilde{\checkmark}$ is indexed. The monotonicity of *iProp* means that, for any $P \in \mathit{iProp}$, resource inclusion $o \lesssim o'$ implies the entailment $P\,o \vDash P\,o'$ on indexed propositions $\widetilde{\mathit{Prop}}$; this monotonicity makes the Iris separation logic *affine* ($*$-ELIM, § 2.1.1).

Importantly, the component cameras $\tilde{\mathcal{A}}_i$ can depend on the Iris propositions *iProp* under the guard of the *later* constructor $\blacktriangleright$ (like (1.19) in § 1.4). This is essential to Iris's existing mechanisms for propositional sharing. For example, Iris's existing camera for the invariant iInv depends on $\blacktriangleright$ *iProp*, as we see in § 3.1.3.

Resource Ownership Iris has the *resource ownership* connective $\boxed{o}^{\gamma}_{\tilde{\mathcal{A}}} \in \mathit{iProp}$ for owning a resource $o \in \tilde{\mathcal{A}}$ of a component camera $\tilde{\mathcal{A}}$ at a ghost name $\gamma \in \mathit{GhostName}$.

Here we think of the case when the component camera $\tilde{\mathcal{A}}$ is a resource algebra $\mathcal{A}$. Iris has the following rules for the resource ownership connective:

$$\boxed{o \cdot o'}^{\gamma} = \boxed{o}^{\gamma} * \boxed{o'}^{\gamma} \ \ \textsc{own-}\cdot\textsc{-}* \qquad \frac{|o| = o}{\boxed{o}^{\gamma} \text{ is persistent}} \ \textsc{own-core-persist}$$

$$\boxed{o}^{\gamma} \vDash \checkmark o \ \ \textsc{own-}\checkmark \qquad \frac{\checkmark o}{\vDash \dot{\Rrightarrow} \exists \gamma.\ \boxed{o}^{\gamma}} \ \textsc{own-alloc}$$

[4] In the Coq mechanization of Iris, the index set I is restricted to a finite set.

[5] The word 'ghost' comes from the term 'ghost state'. The term 'ghost state' originally means a program state that does not appear in the actual execution but is used for verification. By extension, the Iris community uses the term 'ghost state' for any resources of separation logic.

$$\frac{o \rightsquigarrow^{\in} O'}{\boxed{o}^{\gamma} \vDash \dot{\Rrightarrow} \exists o' \in O'.\ \boxed{o'}^{\gamma}}\ \textsc{own-}\rightsquigarrow^{\in} \qquad \frac{o \rightsquigarrow o'}{\boxed{o}^{\gamma} \vDash \dot{\Rrightarrow} \boxed{o'}^{\gamma}}\ \textsc{own-}\rightsquigarrow$$

For resource ownership, the product $\cdot$ corresponds to the separating conjunction $*$ (OWN-$\cdot$-$*$). Ownership of a resource that is a core is persistent (OWN-CORE-PERSIST). Ownership ensures validity (OWN-$\checkmark$). We can allocate a valid resource into a fresh ghost name γ (OWN-ALLOC). We can update the resource of the ownership connective with respect to the resource update relation (OWN-$\rightsquigarrow^{\in}$, OWN-$\rightsquigarrow$). Also, when the resource algebra $\mathcal{A}$ is unital, the ownership of the unit ε can be obtained for free:[6]

$$\vDash \dot{\Rrightarrow} \boxed{\varepsilon}^{\gamma} \quad \textsc{own-}\varepsilon$$

Also, the resource ownership is *timeless* for the resource algebra $\mathcal{A}$:

$$\frac{\mathcal{A} \text{ is a (discrete) resource algebra}}{\boxed{o}^{\gamma}_{\mathcal{A}} \text{ is timeless}}\ \textsc{own-ra-timeless}$$

This does not hold for the resource ownership $\boxed{o}^{\gamma}_{\tilde{\mathcal{A}}}$ of a non-discrete camera $\tilde{\mathcal{A}}$.

2.2.2 Various Constructions of Resource Algebras

Now we present various constructions of resource algebras.

Exclusive RA The *exclusive RA* Ex A over a set A is defined as follows:

$$\llcorner \mathrm{Ex}\, A \lrcorner \ni o ::= \mathrm{ex}\, a \ \ (a \in A) \mid \lightning \qquad o \cdot o' \triangleq \lightning$$
$$|o| \text{ is undefined} \qquad \checkmark o \triangleq o \neq \lightning$$

The *exclusive token* ex a exclusively owns an element $a \in A$. We also have the *invalidity* $\lightning$, returned by any product. The exclusive resource ex a is *exclusive*. So by EXCLUS-$\rightsquigarrow$, we can freely update the content of the exclusive resource: ex $a \rightsquigarrow$ ex a'.

Agreement RA The *agreement RA* AG A over a set A is defined as follows:[7]

$$\llcorner \mathrm{Ag}\, A \lrcorner \ni o ::= \mathrm{ag}\, a \ \ (a \in A) \mid \lightning$$
$$\mathrm{ag}\, a \cdot \mathrm{ag}\, a' \triangleq \begin{cases} \mathrm{ag}\, a & a = a' \\ \lightning & \text{otherwise} \end{cases} \qquad o \cdot \lightning \triangleq \lightning \cdot o \triangleq \lightning$$
$$|o| = o \qquad \checkmark o \triangleq o \neq \lightning$$

We have the *agreement* witness ag a, which is idempotent. We also have the invalidity $\lightning$, returned by the product of inconsistent witnesses. Agreement witnesses ag a and ag a' can validly coexist only if $a = a'$ holds.

Unit RA The *unit RA* UNIT is an defined as follows:

$$\llcorner \mathrm{Unit} \lrcorner \triangleq \mathit{1} \qquad () \cdot () \triangleq () \qquad |()| \triangleq () \qquad \checkmark() \triangleq \top$$

It is unital with the unit $\varepsilon_{\mathrm{Unit}} \triangleq ()$.

6 One may well expect a stronger rule $\vDash \boxed{\varepsilon}^{\gamma}$. The basic update $\dot{\Rrightarrow}$ is required here due to Iris's technical problems in avoiding axioms.
7 The actual agreement RA in Iris is unital, but we omit the unit for simplicity.

Fraction RA The *fraction RA* Frac is defined as follows:

$$\llcorner \textsc{Frac} \lrcorner \triangleq \mathbb{Q}_{>0} \qquad q \cdot r \triangleq q + r \qquad |q| \text{ is undefined} \qquad \checkmark q \triangleq q \le 1$$

The full fraction 1 is exclusive.

Discardable Fraction RA The *discardable fraction RA* Dfrac is roughly the fraction RA Frac with the power to discard a fraction. Formally, it is defined as follows:

$$\llcorner \textsc{Dfrac} \lrcorner \ni o ::= q \ (q \in \mathbb{Q}_{>0}) \mid \star \mid q + \star \qquad q \cdot r \triangleq q + r$$

$$\star \cdot \star \triangleq \star \qquad q \cdot \star \triangleq \star \cdot q \triangleq (q + \star) \cdot \star \triangleq \star \cdot (q + \star) \triangleq q + \star$$

$$q \cdot (r + \star) \triangleq (q + \star) \cdot r \triangleq (q + \star) \cdot (r + \star) \triangleq (q + r) + \star$$

$$|q| \text{ is undefined} \qquad |\star| \triangleq |q + \star| \triangleq \star$$

$$\checkmark q \triangleq q \le 1 \qquad \checkmark \star \triangleq \top \qquad \checkmark (q + \star) \triangleq q < 1$$

The *discard witness* $\star$ persistently observes a discard of a fraction. The resource $q + \star$ is a combination of a fraction q and a discard witness $\star$. The existence of the discard witness $\star$ ensures that the global fraction is less than 1 (by $\checkmark(q+\star) \triangleq q < 1$). The full fraction 1 is exclusive, just like in Frac.

The discardable fraction RA satisfies the following resource update rules:

$$q \rightsquigarrow \star \quad \textsc{dfrac-discard} \qquad \star \rightsquigarrow^{\in} \{q \mid q \in \mathbb{Q}_{>0}\} \quad \textsc{dfrac-restore}$$

We can always discard a fraction q to get a discard witness $\star$ (dfrac-discard). Conversely, we can restore a fraction q out of a discard witness $\star$ (dfrac-restore).

Powerset RA The *powerset RA* Pow A over a set A is defined as follows:

$$\llcorner \mathrm{Pow}\, A \lrcorner \ni S_* ::= S \ (S \in \mathrm{Pow}\, A) \mid ↯$$

$$S \cdot S' \triangleq \begin{cases} S \cup S' & S \cap S' = \varnothing \\ ↯ & \text{otherwise} \end{cases} \qquad S_* \cdot ↯ \triangleq ↯ \cdot S_* \triangleq ↯$$

$$|S_*| \triangleq \varnothing \qquad \checkmark S_* \triangleq S_* \neq ↯$$

This RA is unital with the unit $\varepsilon \triangleq \varnothing$. This RA equips the subsets of A with the *disjoint union*, using the invalidity $↯$ for the union of overlapping sets.

Similarly, we define the *finite powerset RA* FinPow A over a set A by setting

$$\llcorner \mathrm{FinPow}\, A \lrcorner \ni S_* ::= S \ (S \in \mathrm{Pow}_{\mathrm{fin}}\, A) \mid ↯,$$

restricting S to *finite* subsets of A, and setting other operations just like in Pow A.

Sum RA The *sum RA* $\mathcal{A} +_{↯} \mathcal{B}$ of RAs $\mathcal{A}, \mathcal{B}$ is defined as follows:[8]

$$\llcorner \mathcal{A} +_{↯} \mathcal{B} \lrcorner \ni o_* ::= \mathsf{inl}\, o_1 \ (o_1 \in \llcorner \mathcal{A} \lrcorner) \mid \mathsf{inr}\, o_2 \ (o_2 \in \llcorner \mathcal{B} \lrcorner) \mid ↯$$

$$\mathsf{inl}\, o_1 \cdot \mathsf{inl}\, o_1' \triangleq \mathsf{inl}\,(o_1 \cdot_{\mathcal{A}} o_1') \qquad \mathsf{inr}\, o_2 \cdot \mathsf{inr}\, o_2' \triangleq \mathsf{inr}\,(o_2 \cdot_{\mathcal{B}} o_2')$$

$$\mathsf{inl}\, o_1 \cdot \mathsf{inr}\, o_2 \triangleq \mathsf{inr}\, o_2 \cdot \mathsf{inl}\, o_1 \triangleq ↯ \cdot o_* \triangleq o_* \cdot ↯ \triangleq ↯$$

$$|\mathsf{inl}\, o_1| \triangleq \mathsf{inl}^?\, |o_1|_{\mathcal{A}} \qquad |\mathsf{inr}\, o_2| \triangleq \mathsf{inr}^?\, |o_2|_{\mathcal{B}} \qquad |↯| \triangleq ↯$$

$$\checkmark \mathsf{inl}\, o_1 \triangleq \checkmark_{\mathcal{A}}\, o_1 \qquad \checkmark \mathsf{inr}\, o_2 \triangleq \checkmark_{\mathcal{B}}\, o_2 \qquad \checkmark ↯ \triangleq \bot$$

In this RA, we have a resource $\mathsf{inl}\, o_1$ from the RA $\mathcal{A}$ and a resource $\mathsf{inr}\, o_2$ from the RA $\mathcal{B}$. Combining resources from different RAs results in the *invalidity* $↯$. The injection $\mathsf{inl}\, o / \mathsf{inr}\, o$ is exclusive if o is exclusive.

[8] The partial injection $\mathsf{inl}^?\, o^? / \mathsf{inr}^?\, o^?$ is defined only if $o^?$ is defined.

Product RA The *product RA* $\mathcal{A} \times \mathcal{B}$ of RAs $\mathcal{A}, \mathcal{B}$ is defined component-wise as follows:[9]

$$\llcorner \mathcal{A} \times \mathcal{B} \lrcorner \triangleq \llcorner \mathcal{A} \lrcorner \times \llcorner \mathcal{B} \lrcorner \qquad (o_1, o_2) \cdot (o_1', o_2') \triangleq (o_1 \cdot_{\mathcal{A}} o_1',\ o_2 \cdot_{\mathcal{B}} o_2')$$
$$\big|(o_1, o_2)\big| \triangleq \big(|o_1|_{\mathcal{A}},\ |o_2|_{\mathcal{B}}\big)^? \qquad \checkmark(o_1, o_2) \triangleq \checkmark_{\mathcal{A}}\, o_1 \wedge \checkmark_{\mathcal{B}}\, o_2$$

When both $\mathcal{A}$ and $\mathcal{B}$ are unital, the product $\mathcal{A} \times \mathcal{B}$ is also unital with the unit $(\varepsilon_{\mathcal{A}}, \varepsilon_{\mathcal{B}})$.

A resource update over the product RA can be built up component-wise:

$$\frac{o_1 \rightsquigarrow^{\in} O_1' \quad o_2 \rightsquigarrow^{\in} O_2'}{(o_1, o_2) \rightsquigarrow^{\in} O_1' \times O_2'} \;\; \times\text{-}\rightsquigarrow^{\in}$$

Finite Map RA The *finite map RA* $I \xrightarrow{\text{fin}} \mathcal{A}$ of an index set I that is infinite and an RA $\mathcal{A}$ is defined point-wise as follows:[10]

$$\llcorner I \xrightarrow{\text{fin}} \mathcal{A} \lrcorner \triangleq I \xrightarrow{\text{fin}} \llcorner \mathcal{A} \lrcorner \qquad (f \cdot g)\, i \triangleq f\, i \cdot^? g\, i$$
$$|f|\, i \triangleq |f\, i| \qquad \checkmark f \triangleq \forall i \in \operatorname{dom} f.\ \checkmark(f\, i)$$

The finite map RA is unital, with the empty finite map $\varnothing$ being the unit $\varepsilon_{I \xrightarrow{\text{fin}} \mathcal{A}} \triangleq \varnothing$.

The finite map RA satisfies the following resource update rules:

$$\frac{\checkmark_{\mathcal{A}}\, o}{\varnothing \rightsquigarrow^{\in} \{i := o \mid i \in I\}} \;\; \xrightarrow{\text{fin}}\text{-ALLOC}$$

$$\frac{\forall i \in \operatorname{dom} f.\ f\, i \rightsquigarrow^{\in} O_i'}{f \rightsquigarrow^{\in} \big\{ g \;\big|\; \operatorname{dom} g = \operatorname{dom} f \ \wedge\ \forall i \in \operatorname{dom} f.\ g\, i \in O_i' \big\}} \;\; \xrightarrow{\text{fin}}\text{-}\rightsquigarrow^{\in}$$

We write $[i := x]$ for a singleton finite map that only maps i to x. We can allocate a singleton map $[i := o]$ of a valid resource o for some index $i \in I$, thanks to the infiniteness of I and the domain finiteness of the global resource ($\xrightarrow{\text{fin}}$-ALLOC). Also, we can update a finite map point-wise ($\xrightarrow{\text{fin}}\text{-}\rightsquigarrow^{\in}$).

Option RA The *option RA* OPTION $\mathcal{A}$ over a resource algebra $\mathcal{A}$ is defined as follows:

$$\llcorner \text{OPTION}\ \mathcal{A} \lrcorner \triangleq \mathit{Option}\, \llcorner \mathcal{A} \lrcorner$$
$$\mathsf{some}\, o \cdot \mathsf{some}\, o' \triangleq \mathsf{some}(o \cdot_{\mathcal{A}} o') \qquad o_* \cdot \mathsf{none} \triangleq \mathsf{none} \cdot o_* \triangleq o_*$$
$$|\mathsf{some}\, o| \triangleq \begin{cases} \mathsf{some}\, |o|_{\mathcal{A}} & |o|_{\mathcal{A}} \text{ is defined} \\ \mathsf{none} & \text{otherwise} \end{cases} \qquad |\mathsf{none}| \triangleq \mathsf{none}$$
$$\checkmark(\mathsf{some}\, o) \triangleq \checkmark_{\mathcal{A}}\, o \qquad \checkmark\, \mathsf{none} \triangleq \top$$

It is also unital by the unit $\varepsilon \triangleq \mathsf{none}$. The option RA OPTION $\mathcal{A}$ can be understood as an extension of $\mathcal{A}$ with the unit none.

[9] The partial pair $(o_1^?, o_2^?)^?$ is defined only if both $o_1^?$ and $o_2^?$ are defined.

[10] The partial product $o_1^? \cdot^? o_2^?$ is defined as $o_1^? \cdot o_2^?$ if both $o_1^?$ and $o_2^?$ are defined, as $o_1^?$ if $o_1^?$ is defined and $o_2^?$ is not, as $o_2^?$ if $o_2^?$ is defined and $o_1^?$ is not, and is undefined otherwise. The core $|o^?|$ is defined only if $o^?$ and $|o^?|$ are defined.

Authoritative RA The *authoritative RA* $\textsc{Auth}\,\mathcal{A}$ over a unital resource algebra $\mathcal{A}$ is defined as follows:

$$\llcorner\textsc{Auth}\,\mathcal{A}\lrcorner \;\triangleq\; \llcorner\textsc{Auth}'\,\mathcal{A}\lrcorner \qquad \textsc{Auth}'\,\mathcal{A} \;\triangleq\; \textsc{Option}(\textsc{Ex}\,\llcorner\mathcal{A}\lrcorner) \times \mathcal{A}$$

$$o_* \cdot o'_* \;\triangleq\; o_* \cdot_{\textsc{Auth}'\,\mathcal{A}} o'_* \qquad |o_*| \;\triangleq\; |o_*|_{\textsc{Auth}'\,\mathcal{A}} \qquad \checkmark(\mathsf{none}, o_f) \;\triangleq\; \checkmark_{\mathcal{A}}\, o_f$$

$$\checkmark(\mathsf{some}\, o_+, o_f) \;\triangleq\; \exists o_a \text{ s.t. } o_+ = \mathsf{ex}\, o_a.\;\; \checkmark_{\mathcal{A}}\, o_a \;\wedge\; o_f \lesssim_{\mathcal{A}} o_a$$

It is also unital by the unit $\varepsilon_{\textsc{Auth}'\,\mathcal{A}}$. In the authoritative RA, we have the *authoritative* token $\bullet\, o \;\triangleq\; (\mathsf{some}(\mathsf{ex}\, o), \varepsilon)$ and the *fragment* token $\circ\, o \;\triangleq\; (\mathsf{none}, o)$, satisfying the following properties:

$$\circ\, o \cdot \circ\, o' \;=\; \circ(o \cdot o') \qquad \circ\, \varepsilon \;=\; \varepsilon \qquad |\circ\, o| \;=\; \circ|o|$$

$$\neg \checkmark(\bullet\, o \cdot \bullet\, o') \qquad \checkmark(\circ\, o) \;\triangleq\; \checkmark_{\mathcal{A}}\, o$$

$$\checkmark(\bullet\, o_a \cdot \circ\, o_f) \;=\; \checkmark_{\mathcal{A}}\, o_a \;\wedge\; o_f \lesssim_{\mathcal{A}} o_a \qquad \checkmark\text{-}\bullet\text{-}\circ$$

When we have both an authoritative token $\bullet\, o_a$ and a fragment token $\circ\, o_f$, the fragment o_f is ensured to be a part of the resource o_a owned by the authoritative token ($\checkmark$-$\bullet$-$\circ$).

2.2.3 Example: Heap Resource Algebra

The *heap resource algebra* $\textsc{Heap}$ for reasoning about heap memory can be defined as follows:

$$\textsc{Heap} \;\triangleq\; \textsc{Auth}\left(\mathit{Loc} \overset{\text{fin}}{\rightharpoonup} \textsc{Frac} \times \textsc{Ag}\,\mathit{Val}\right)$$

It is an *authoritative* RA over the *finite map* RA $\overset{\text{fin}}{\rightharpoonup}$ from locations Loc to values under *fractional ownership* $\textsc{Frac} \times \textsc{Ag}\,\mathit{Val}$.

To use the heap RA, we add it to the component cameras of Iris, which enables using the resource ownership $\boxed{o}^{\gamma}_{\textsc{Heap}}$ over the heap RA $\textsc{Heap}$ (§ 2.2.1).

The fractional *points-to token* $\ell \overset{q}{\mapsto} v$ is modeled as follows:

$$\ell \overset{q}{\mapsto} v \;\triangleq\; \boxed{\circ\,[\ell := (q, \mathsf{ag}\, v)]}^{\gamma_{\textsc{Heap}}}_{\textsc{Heap}}$$

It owns a *fragment* resource $\circ$ of a singleton map from $\ell \in \mathit{Loc}$ to $(q, \mathsf{ag}\, v)$, observing the value $v \in \mathit{Val}$ with fractional ownership $q \in \mathbb{Q}_{>0}$. Here, we assume that a ghost name $\gamma_{\textsc{Heap}} \in \mathit{GhostName}$ has been globally taken (by the rule HEAP-INIT introduced later). We just write $\ell \mapsto v$ for the full points-to token $\ell \overset{1}{\mapsto} v$.

The *exclusive heap token* $\mathsf{heap}\, H$ for a heap $H \in \mathit{Heap} \triangleq \mathit{Loc} \overset{\text{fin}}{\rightharpoonup} \mathit{Val}$ is modeled as follows:[11]

$$\mathsf{heap}\, H \;\triangleq\; \boxed{\bullet\left(\mathsf{map}\,\left(\lambda v.\,(1, \mathsf{ag}\, v)\right)\, H\right)}^{\gamma_{\textsc{Heap}}}_{\textsc{Heap}}$$

It owns an *authoritative* resource $\bullet$ of a finite map corresponding to the heap H that has a full-ownership resource $(1, \mathsf{ag}\,(H\,\ell))$ observing the value $H\,\ell$ over the locations $\ell \in \operatorname{dom} H$.

We take the global ghost name $\gamma_{\textsc{Heap}}$ by the following rule:

$$\vDash \dot{\Rrightarrow} \left(\exists\, \gamma_{\textsc{Heap}} \in \mathit{GhostName}.\;\; \mathsf{heap}\, \varnothing\right) \qquad \text{HEAP-INIT}$$

After a basic update $\dot{\Rrightarrow}$, we get $\gamma_{\textsc{Heap}} \in \mathit{GhostName}$ with the exclusive heap token $\mathsf{heap}\, \varnothing$ initialized with the empty heap state $\varnothing$. We can prove this using OWN-ALLOC.

[11] The map function $\mathsf{map}\, f\, as$ for a function $f\colon X \to A$ and a finite map $as\colon X \overset{\text{fin}}{\rightharpoonup} B$ returns the finite map $bs\colon X \overset{\text{fin}}{\rightharpoonup} B$ such that $\operatorname{dom} bs = \operatorname{dom} as$ and $bs\, x = f\,(as\, x)$ for all $x \in \operatorname{dom} as$.

We can prove the following rules for memory allocation, read and write:

$$\mathsf{heap}\, H \;\vDash \dot{\Rrightarrow} \big(\exists\, \ell \in \mathit{Loc}.\;\; \mathsf{heap}\; H\{\ell := v\} \,*\, \ell \mapsto v\,\big) \qquad \textsc{heap-alloc}$$

$$\mathsf{heap}\, H \,*\, \ell \mapsto v \;\vDash\; H\,\ell = v \qquad \textsc{heap-read}$$

$$\mathsf{heap}\, H \,*\, \ell \mapsto v \;\vDash \dot{\Rrightarrow} \big(\,\mathsf{heap}\; H\{\ell := w\} \,*\, \ell \mapsto w\,\big) \qquad \textsc{heap-write}$$

Here, we write $H\{\ell := v\}$ for H with the value at the location ℓ updated to v. These rules hold thanks to own-$\rightsquigarrow^{\in}$ and the relationship between the authoritative resource $\bullet$ and the fragment resource $\circ$. We can use these rules to model Hoare triple rules for memory allocation, read and write (thoare-alloc, thoare-load and thoare-store in § 3.3.2).

We can also prove the following rules on the fractional points-to token, similar to those presented in § 1.2.3:

$$\ell \overset{q+r}{\mapsto} v \;=\; \ell \overset{q}{\mapsto} v \,*\, \ell \overset{r}{\mapsto} v \qquad \mapsto\text{-FRACT}$$

$$\frac{q > 1}{\ell \overset{q}{\mapsto} v \;=\; \bot} \;\; \mapsto\text{-OVER1} \qquad\qquad \ell \overset{q}{\mapsto} v \,*\, \ell \overset{r}{\mapsto} v' \;\vDash\; v = v' \qquad \mapsto\text{-AGREE}$$

The rule $\mapsto$-fract can be proved using own-$\cdot$-$*$. The rules $\mapsto$-over1 and $\mapsto$-agree can be proved using own-$\checkmark$.

Chapter 3

Overview of Our Framework

Seeing is believing

A saying in English

This chapter presents an overview of the design principle and usage of our framework, Nola, focusing on the later-free *shared invariant* mechanism.

This chapter is organized as follows. Section 3.1 reviews the invariants of Iris. Section 3.2 presents Nola's later-free invariant mechanism. Section 3.3 shows how to use them through a verification example of iterative mutation of a shared mutable singly linked list. Section 3.4 presents our new paradox of later-free invariants and discusses how our framework naturally avoids paradoxes and the general expressivity of our framework.

3.1 Preliminaries on Iris's Invariants

Subsection 3.1.1 explains the *fancy update* ${}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'} P$, a core modality for using Iris's invariants, and Iris's *Hoare triples*, which are modeled using the fancy update. Subsection 3.1.2 explains Iris's invariant mechanism. Subsection 3.1.3 presents the model for Iris's invariant mechanism.

3.1.1 Fancy Update and Hoare Triples

Fancy Update A core logical connective for using Iris's invariants is the *fancy update* modality ${}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'}$.[1] Roughly speaking, the fancy update is the *basic update* modality $\dot{\Rrightarrow}$ (§ 2.1.1) enriched with the internal 'memory' for invariants. The fancy update is parameterized by the *masks* $\mathcal{E}, \mathcal{E}' \subseteq \mathit{InvName}$ (omitted in § 1.3.1), representing the set of available 'invariant names' before and after the update, respectively. We introduce the shorthand ${\Rrightarrow}_{\mathcal{E}} \triangleq {}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}}$ for the case the mask is unchanged. We usually use this mask-unchanging fancy update ${\Rrightarrow}_{\mathcal{E}}$ of the more general mask-changing one ${}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'}$.

The fancy update forms an (indexed) *monad*:

$$\frac{P \vDash Q}{{}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'} P \vDash {}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'} Q}\ {\Rrightarrow}\text{-MONO} \qquad P \vDash {\Rrightarrow}_{\mathcal{E}} P \quad {\Rrightarrow}\text{-INTRO}$$

$${}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'}\, {}_{\mathcal{E}'}{\Rrightarrow}_{\mathcal{E}''} P \vDash {}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}''} P \quad {\Rrightarrow}\text{-TRANS}$$

Also, the monad is strong with respect to the separating conjunction $*$, i.e., satisfies the following *frame* rule:

$$\big({}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'} P\big) * Q \vDash {}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'} (P * Q) \quad {\Rrightarrow}\text{-FRAME}$$

[1] We do not use the standard notation where masks are written as superscripts for a mask-changing fancy update, because we want to write the custom *world satisfaction* W as the superscript in the *extended fancy update* ${}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'}$, which the Nola framework introduces (§ 3.2.1).

The fancy update also satisfies the frame rule on masks:

$$ {}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'}\, P \;\vDash\; {}_{\mathcal{E}+\mathcal{E}''}{\Rrightarrow}_{\mathcal{E}'+\mathcal{E}''}\, P \qquad \Rrightarrow\text{-}\textsc{mask-frame} $$

From this rule, we can derive the following monotonicity on the mask:

$$ \frac{\mathcal{E} \subseteq \mathcal{E}'}{{\Rrightarrow}_{\mathcal{E}}\, P \;\vDash\; {\Rrightarrow}_{\mathcal{E}'}\, P} \quad \Rrightarrow\text{-}\textsc{mask-}\subseteq $$

The basic update $\dot{\Rrightarrow}$ can turn into the fancy update ${\Rrightarrow}_{\mathcal{E}}$:

$$ \dot{\Rrightarrow}\, P \;\vDash\; {\Rrightarrow}_{\mathcal{E}}\, P \qquad \dot{\Rrightarrow}\text{-}{\Rrightarrow} $$

Also, the fancy update absorbs the except-0 modality:

$$ \diamond\, {}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'}\, P \;=\; {}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'}\, P \quad \diamond\text{-}{\Rrightarrow} \qquad\qquad {}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'}\, \diamond P \;=\; {}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'}\, P \quad {\Rrightarrow}\text{-}\diamond $$

Unfortunately, the fancy update over a pure predicate ${\Rrightarrow}_{\mathcal{E}} \ulcorner\phi\urcorner$ is not equal to $\ulcorner\phi\urcorner$, unlike the basic update $\dot{\Rrightarrow} \ulcorner\phi\urcorner$ ($\dot{\Rrightarrow}$-pure). Still, the fancy update satisfies the following weaker rule:

$$ Q * \left(Q \mathrel{-\!\!*} {\Rrightarrow}_{\mathcal{E}} \ulcorner\phi\urcorner\right) \;\vDash\; {\Rrightarrow}_{\mathcal{E}} \left(\ulcorner\phi\urcorner * Q\right) \qquad \Rrightarrow\text{-}\textsc{pure-keep} $$

Usually, by applying Q to $Q \mathrel{-\!\!*} {\Rrightarrow}_{\mathcal{E}}\, P$, we get only P after the fancy update ${\Rrightarrow}_{\mathcal{E}}$, but this rule $\Rrightarrow$-pure-keep lets us keep Q as well when P is a pure proposition $\ulcorner\phi\urcorner$.

Hoare Triples Iris's Hoare triples $\{P\}\, e\, \{\Psi\}_{\mathcal{E}}$, $[P]\, e\, [\Psi]_{\mathcal{E}}$ are parameterized with the *mask* $\mathcal{E}$, because they are modeled with the *fancy update* ${\Rrightarrow}_{\mathcal{E}}$ (see (3.5) and (3.6) in § 3.1.3 for more details).

Thanks to that, the Hoare triples absorb the fancy update:[2]

$$ \frac{\{P\}\, e\, \{\Psi\}_{\mathcal{E}}}{\{{\Rrightarrow}_{\mathcal{E}} P\}\, e\, \{\Psi\}_{\mathcal{E}}} \quad \Rrightarrow\text{-}\textsc{phoare} \qquad\qquad \frac{[P]\, e\, [\Psi]_{\mathcal{E}}}{[{\Rrightarrow}_{\mathcal{E}} P]\, e\, [\Psi]_{\mathcal{E}}} \quad \Rrightarrow\text{-}\textsc{thoare} $$

$$ \frac{\{P\}\, e\, \{\lambda v.\ {\Rrightarrow}_{\mathcal{E}} \Psi v\}_{\mathcal{E}}}{\{P\}\, e\, \{\Psi\}_{\mathcal{E}}} \quad \textsc{phoare-}{\Rrightarrow} \qquad\qquad \frac{[P]\, e\, [\lambda v.\ {\Rrightarrow}_{\mathcal{E}} \Psi v]_{\mathcal{E}}}{[P]\, e\, [\Psi]_{\mathcal{E}}} \quad \textsc{thoare-}{\Rrightarrow} $$

This intuitively means that the fancy update can be performed before and after any computation step.

Also, the Hoare triples are monotone over the mask:

$$ \frac{\{P\}\, e\, \{\Psi\}_{\mathcal{E}} \quad \mathcal{E} \subseteq \mathcal{E}'}{\{P\}\, e\, \{\Psi\}_{\mathcal{E}'}} \quad \textsc{phoare-mask-}\subseteq \qquad\qquad \frac{[P]\, e\, [\Psi]_{\mathcal{E}} \quad \mathcal{E} \subseteq \mathcal{E}'}{[P]\, e\, [\Psi]_{\mathcal{E}'}} \quad \textsc{thoare-mask-}\subseteq $$

Iris's Hoare triples also satisfy the usual rules discussed in § 1.2.3. For example, they satisfy the following *frame* rules:

$$ \frac{\{P\}\, e\, \{\Psi\}_{\mathcal{E}}}{\{P * R\}\, e\, \{\lambda v.\ \Psi v * R\}_{\mathcal{E}}} \quad \textsc{phoare-frame} $$

$$ \frac{[P]\, e\, [\Psi]_{\mathcal{E}}}{[P * R]\, e\, [\lambda v.\ \Psi v * R]_{\mathcal{E}}} \quad \textsc{thoare-frame} $$

[2] The Hoare triples are actually persistent Iris propositions in *iProp* instead of pure propositions in *Prop*. So rules with Hoare triple hypotheses are interpreted as an entailment $\vDash$ between Iris propositions. For example, the rule $\Rrightarrow$-phoare is interpreted as $\{P\}\, e\, \{\Psi\}_{\mathcal{E}} \vDash \{{\Rrightarrow}_{\mathcal{E}} P\}\, e\, \{\Psi\}_{\mathcal{E}}$.

Also, the total Hoare triple entails the partial one:

$$\frac{\left[P\right] e \left[\Psi\right]_{\mathcal{E}}}{\left\{P\right\} e \left\{\Psi\right\}_{\mathcal{E}}} \quad \text{THOARE-PHOARE}$$

This language has the following rules for memory allocation, read and write:[3]

$$\left[\top\right] \ \ \mathsf{ref}\, v \ \left[\lambda w.\ \exists \ell \in \mathit{Loc} \text{ s.t. } w = \ell.\ \ell \mapsto v\right]_{\varnothing} \quad \text{THOARE-ALLOC}$$

$$\left[\ell \overset{q}{\mapsto} v\right] \ !\ell \ \left[\lambda w.\ w = v \ast \ell \overset{q}{\mapsto} v\right]_{\varnothing} \quad \text{THOARE-LOAD}$$

$$\left[\ell \mapsto v\right] \ \ell \leftarrow w \ \left[\lambda \_.\ \ell \mapsto w\right]_{\varnothing} \quad \text{THOARE-STORE}$$

3.1.2 Iris's Invariants

Iris's *invariant* connective $\boxed{P}^{\mathcal{N}} \in \mathit{iProp}$ *persistently* asserts that the situation described by the separation logic proposition $P \in \mathit{iProp}$ always holds, as explained in § 1.3.1.

Importantly, the proposition P stored into Iris's invariant $\boxed{P}^{\mathcal{N}}$ can be an *arbitrary* Iris proposition, which can contain the invariant connective itself $\boxed{-}^{\mathcal{N}}$, enabling *nested invariants.*

The *namespace* $\mathcal{N} \in \mathit{Namespace}$ of the invariant connective $\boxed{P}^{\mathcal{N}}$ (which we omitted in § 1.3.1) represents a set of possible invariant names $\iota \in \mathit{InvName}$ given to the invariant.[4] The namespace is used for prohibiting *reentrancy* to invariants, as we will see later in the invariant access rules (PHOARE-IINV etc.).

Proof Rules Notably, the invariant connective is *persistent*:

$$\boxed{P}^{\mathcal{N}} \text{ is persistent} \quad \text{IINV-PERSIST}$$

In particular, the invariant connective is *duplicable*, since persistence entails duplicability (PERSIST-DUP, § 2.1.1):

$$\boxed{P}^{\mathcal{N}} \vDash \boxed{P}^{\mathcal{N}} \ast \boxed{P}^{\mathcal{N}} \quad \text{IINV-DUP}$$

An invariant $\boxed{P}^{\mathcal{N}}$ can be allocated by storing the shared content $\triangleright P$:

$$\triangleright P \vDash \mathrel{\mid\!\!\Rrightarrow}_{\varnothing} \boxed{P}^{\mathcal{N}} \quad \text{IINV-ALLOC}$$

Here, note the use of the *fancy update* $\mathrel{\mid\!\!\Rrightarrow}_{\varnothing}$.

Iris has the following rules for *accessing* the shared content P of an invariant $\boxed{P}^{\mathcal{N}}$ in the Hoare triples:

$$\frac{\left\{(\triangleright P) \ast Q\right\} e \left\{\lambda v.\ (\triangleright P) \ast \Psi v\right\}_{\mathcal{E}} \quad e \text{ is atomic}}{\left\{\boxed{P}^{\mathcal{N}} \ast Q\right\} e \left\{\Psi\right\}_{\mathcal{N}+\mathcal{E}}} \quad \text{PHOARE-IINV}$$

$$\frac{\left[(\triangleright P) \ast Q\right] e \left[\lambda v.\ (\triangleright P) \ast \Psi v\right]_{\mathcal{E}} \quad e \text{ is atomic}}{\left[\boxed{P}^{\mathcal{N}} \ast Q\right] e \left[\Psi\right]_{\mathcal{N}+\mathcal{E}}} \quad \text{THOARE-IINV}$$

[3] This dissertation uses Iris's *HeapLang*, a simple imperative language with heap memory, for the target low-level language. To be precise, it uses a slight variant of HeapLang with a primitive ndnat that takes a non-deterministic natural number.

[4] To be precise, a namespace $\mathcal{N} \in \mathit{Namespace} \subseteq \mathrm{Pow}\, \mathit{InvName}$ is a principal filter $\uparrow\iota \subseteq \mathit{InvName}$ generated from some invariant name $\iota \in \mathit{InvName}$, where $\mathit{InvName}$ is partially ordered by some prefix relation. This ensures especially that the set $\mathcal{N}$ has an *infinite* number of elements.

The rules roughly say that the shared content P of the invariant $\boxed{P}^{\mathcal{N}}$ can be accessed in the precondition as long as P is restored in the postcondition. The content P is weakened by the *later modality* $\triangleright P$, which our Nola framework eliminates as explained in § 3.2. We have the side condition that the expression e is *atomic*, i.e., takes only one computational step, for soundness in the presence of multiple threads accessing the invariant. To prohibit *reentrancy* to invariants, the Hoare triple is parameterized with the mask $\mathcal{E}$, the set of 'available' *invariant names* $\iota \in \mathit{InvName}$. Every time we access an invariant $\boxed{P}^{\mathcal{N}}$, its namespace $\mathcal{N}$ is consumed from the mask.[5]

We also have the following more basic rule for accessing the content of an invariant in terms of the *fancy update* $\Rrightarrow$:

$$\frac{(\triangleright P) * Q \vDash \Rrightarrow_{\mathcal{E}} \big((\triangleright P) * R\big)}{\boxed{P}^{\mathcal{N}} * Q \vDash \Rrightarrow_{\mathcal{N}+\mathcal{E}} R}\ \textsc{iinv-acc}$$

More generally, we have the following rule for accessing the invariant content in terms of the *mask-changing* fancy update:

$$\boxed{P}^{\mathcal{N}} \vDash {}_{\mathcal{N}}\!\Rrightarrow_{\varnothing} \big((\triangleright P) * \big((\triangleright P) \mathrel{-\!\!*} {}_{\varnothing}\!\Rrightarrow_{\mathcal{N}} \top\big)\big) \quad \textsc{iinv-acc-ch}$$

By consuming the mask $\mathcal{N}$, we can get the shared content $\triangleright P$ and the separating implication $(\triangleright P) \mathrel{-\!\!*} {}_{\varnothing}\!\Rrightarrow_{\mathcal{N}} \top$. The separating implication lets us retrieve the mask $\mathcal{N}$ by storing the content $\triangleright P$ back. From the rule IINV-ACC-CH, we can derive the access rules for the Hoare triples PHOARE-IINV, THOARE-IINV, because the Hoare triples are defined using the fancy update modality $\Rrightarrow_{\mathcal{E}}$.

Unfortunately, Iris's invariant connective $\boxed{P}^{\mathcal{N}}$ is *not timeless* (§ 2.1.2), meaning that a later $\triangleright$ put on the connective cannot be freely stripped off. More precisely, an invariant under the later modality $\triangleright \boxed{P}^{\mathcal{N}}$ does not have the power to access the shared content P. This is problematic for handling *nested invariants* since the access to Iris's invariants (IINV-ACC etc.) is weakened by the later modality $\triangleright$.

Adequacy of Hoare Triples We here mention the adequacy theorems for Iris's Hoare triples $\{P\}\, e\, \{\Psi\}_{\mathcal{E}}$ and $[P]\, e\, [\Psi]_{\mathcal{E}}$. Recall that the Hoare triples are modeled using the *fancy update* $\Rrightarrow$, which is designed to support Iris's invariants (see also (3.5) and (3.6) in § 3.1.3).

Iris has the following adequacy theorem for the partial Hoare triple:

Theorem 3.1 (Adequacy of the Partial Hoare Triple). *If*

$$\forall \gamma_{\mathrm{HEAP}}, \gamma_{\mathrm{INV}}.\ \vDash \{\top\}\, e\, \{\phi\}_{\mathit{InvName}}$$

holds for a pure postcondition $\phi\colon \mathit{Val} \to \mathit{Prop}$,

- *any execution of the program* e never gets stuck, *and*
- *whenever the program* e *terminates with a value* $v \in \mathit{Val}$, *the* postcondition $\phi\, v \in \mathit{Prop}$ *holds.*

The global mask $\mathit{InvName}$ is chosen for simplicity (recall PHOARE-MASK-⊆). The premise is universally quantified over the *ghost names* $\gamma_{\mathrm{HEAP}}, \gamma_{\mathrm{INV}} \in \mathit{GhostName}$ (§ 2.2.1) for the heap and invariant mechanisms. Recall that the ghost name γ_{HEAP} for heap is taken when the exclusive heap token heap H is allocated (see HEAP-INIT, § 2.2.3). The ghost name γ_{INV} is similarly taken by Iris's invariant mechanism.

Iris has the following termination adequacy theorem for the total Hoare triple:

[5] We write $A + B$ for the *disjoint union*, i.e., the union $A \cup B$ defined only if the sets are disjoint $A \cap B = \varnothing$.

Theorem 3.2 (Termination Adequacy of the Total Hoare Triple). *If*

$$\forall \gamma_{\textsc{Heap}}, \gamma_{\textsc{Inv}}.\ \vDash \left[\top\right] e \left[\lambda\_.\,\top\right]_{\mathit{InvName}}$$

holds, then the program e always terminates.

Since the total Hoare triple entails the partial one (THOARE-PHOARE), we can combine Theorem 3.1 and Theorem 3.2 to derive the following adequacy theorem for the total Hoare triple.

Corollary 3.3 (Adequacy of the Total Hoare Triple). *If*

$$\forall \gamma_{\textsc{Heap}}, \gamma_{\textsc{Inv}}.\ \vDash \left[\top\right] e \left[\phi\right]_{\mathit{InvName}}$$

holds for a pure postcondition $\phi\colon \mathit{Val} \to \mathit{Prop}$,

- *any execution of the program e* never gets stuck, *and*
- *the program e* always terminates *with a value $v \in \mathit{Val}$ that satisfies the* postcondition *$\phi\, v \in \mathit{Prop}$.*

The adequacy theorem for the partial Hoare triple Theorem 3.1 comes from the following adequacy theorem for the fancy update $\Rrightarrow$ and later $\triangleright$ modalities Theorem 3.4.

Theorem 3.4 (Adequacy of the Fancy Update and Later Modalities). *If*

$$\forall \gamma_{\textsc{Inv}}.\ \vDash (\Rrightarrow_{\mathit{InvName}} \triangleright)^n \ulcorner \phi \urcorner$$

holds for a pure proposition $\phi \in \mathit{Prop}$ and a natural number $n \in \mathbb{N}$, then ϕ holds.

Because the partial Hoare triple targets only a *safety* property, it can instantiate the natural number n of Theorem 3.4 using the length of finite execution traces to be considered.

On the other hand, the adequacy theorem for the total Hoare triple Corollary 3.3 comes from the rule $\Rrightarrow$-PURE-KEEP for the fancy update $\Rrightarrow$ and pure propositions.

3.1.3 Model

We present the model for Iris's invariant mechanism.

Iris's Invariant Camera To use Iris's invariants, we add a *camera* IINV to the family of component cameras for the Iris proposition *iProp* (§ 2.2.1). The camera is defined as follows:

$$\textsc{iInv} \quad \triangleq \quad \textsc{Auth}\left(\mathit{InvName} \overset{\text{fin}}{\rightharpoonup} \textsc{Ag}\,(\blacktriangleright \mathit{iProp})\right) \tag{3.1}$$

It is an *authoritative* camera AUTH over the finite map camera $\overset{\text{fin}}{\rightharpoonup}$ from invariant names *InvName* to the *agreement* camera AG over later-guarded propositions $\blacktriangleright$ *iProp*.

Importantly, the camera IINV depends on the Iris proposition *iProp*, while *iProp* itself depends on the component cameras. This circular dependency is valid thanks to the guard of the *later* constructor $\blacktriangleright$, coming from Iris's indexed semantics. But this later constructor $\blacktriangleright$ is the exact source of the later modality $\triangleright$ we suffer from (§ 1.4).

Note that the camera IINV corresponds to the 'invariant memory' *InvMem* of the rough model (1.19) we showed.

Iris's Invariant Connective The invariant connective $\boxed{P}^{\mathcal{N}}$ is modeled as follows:[6]

$$\boxed{P}^{\mathcal{N}} \quad\triangleq\quad \exists\iota\in\mathcal{N}.\ \boxed{\circ\,[\iota := \mathrm{ag}(\mathsf{next}\,P)]}^{\gamma_{\mathrm{iInv}}}_{\mathrm{iInv}} \tag{3.2}$$

It uses the *ghost name* $\gamma_{\mathrm{iInv}} \in \mathit{GhostName}$ for the invariant. Such a ghost name is freshly taken when a resource is allocated (recall OWN-ALLOC in § 2.2.1). The data constructor next corresponds to the *later constructor* $\blacktriangleright$ used in iInv. Due to this next, the *agreement* obtained by this token is weakened by the *later modality* $\triangleright$.

Iris's World Satisfaction For reasoning about *masks* $\mathcal{E} \subseteq \mathit{InvName}$, Iris adds to the component cameras the resource algebras Dis, En for the *disabled* and *enabled* invariant names, defined as follows:

$$\mathrm{Dis} \ \triangleq\ \mathrm{FinPow}\ \mathit{InvName} \qquad \mathrm{En} \ \triangleq\ \mathrm{Pow}\ \mathit{InvName}$$

Now the *world satisfaction* Wiinv for Iris's invariants is defined as follows:

$$\mathsf{Wiinv} \quad\triangleq\quad \exists I.\ \boxed{\bullet\,\mathrm{ag}\,(\mathsf{next}\,I)}^{\gamma_{\mathrm{iInv}}}_{\mathrm{iInv}} \ast \mathop{\Huge\ast}_{\iota\in\mathrm{dom}\,I} \Big(\big((\triangleright I\,\iota) \ast \boxed{\{\iota\}}^{\gamma_{\mathrm{Dis}}}_{\mathrm{Dis}}\big) \vee \boxed{\{\iota\}}^{\gamma_{\mathrm{En}}}_{\mathrm{En}}\Big) \tag{3.3}$$

Here, $I\colon \mathit{InvName} \xrightarrow{\text{fin}} \mathit{iProp}$ manages the global information about the currently allocated invariants, associating an invariant name $\iota \in \mathrm{dom}\,I \subseteq \mathit{InvName}$ with an Iris proposition $P \in \mathit{iProp}$. We write $\mathrm{ag}\,(\mathsf{next}\,I)$ for the finite map map $\big(\lambda P.\,\mathrm{ag}\,(\mathsf{next}\,P)\big)\ I$. The definition uses the ghost names $\gamma_{\mathrm{Dis}}, \gamma_{\mathrm{En}}$ for the disabled and enabled invariant names.

For each invariant ι, the world satisfaction Wiinv stores either the proposition for the closed or open state. When the invariant is closed, the world satisfaction stores the shared content $I\,\iota$ under the *later modality* $\triangleright$ and the exclusive witness $\boxed{\{\iota\}}^{\gamma_{\mathrm{Dis}}}_{\mathrm{Dis}}$. When the invariant is open, the world satisfaction just stores the exclusive witness $\boxed{\{\iota\}}^{\gamma_{\mathrm{En}}}_{\mathrm{En}}$, which is taken from the *mask* $\mathcal{E}$ of the fancy update (see (3.4)).

Fancy Update Now the *fancy update* modality ${}_{\mathcal{E}}{\mathrel{|\!\!\Rrightarrow}}_{\mathcal{E}'} P$ is modeled as follows, using the world satisfaction Wiinv:

$${}_{\mathcal{E}}{\mathrel{|\!\!\Rrightarrow}}_{\mathcal{E}'} P \quad\triangleq\quad \mathsf{Wiinv} \ast \boxed{\mathcal{E}}^{\gamma_{\mathrm{En}}}_{\mathrm{En}} \mathrel{-\!\!\ast} \dot{\mathrel{|\!\!\Rrightarrow}} \diamond \big(\mathsf{Wiinv} \ast \boxed{\mathcal{E}'}^{\gamma_{\mathrm{En}}}_{\mathrm{En}} \ast P\big) \tag{3.4}$$

This model (3.4) can be understood by analogy with the *state monad* in functional programming, where the world satisfaction Wiinv is the 'global mutable state' managed by the fancy update. The except-0 modality $\diamond$ used in the definition is just for supporting the rules $\mathrel{|\!\!\Rrightarrow}$-$\diamond$ and $\diamond$-$\mathrel{|\!\!\Rrightarrow}$.

Why the Later Modality Is Needed The later modality $\triangleright$ is needed on the shared content $I\,\iota$ in the world satisfaction Wiinv (3.3), by which Iris's invariants suffer from later-requiring proof rules like IINV-ACC.

The reason for the later modality $\triangleright$ in Wiinv can be explained as follows. The resource $[\iota := \mathrm{ag}\,(\mathsf{next}\,P)]$ in the invariant $\boxed{P}^{\mathcal{N}}$ (3.2) observes the agreement between P and $I\,\iota$ of the world satisfaction Wiinv. But the agreement is weakened by the later modality as $\triangleright(P \stackrel{\sim}{=} I\,\iota)$, due to the next constructor (or $\blacktriangleright$ in iInv). This later-weakened equality entails $(\triangleright P) \stackrel{\sim}{=} \triangleright(I\,\iota)$ but not $P \stackrel{\sim}{=} I\,\iota$.

[6] To be precise, the latest versions of Iris give to the invariant $\boxed{P}^{\mathcal{N}}$ a more advanced model, namely the "accessor" that describes the behavior of invariants using mask-changing fancy updates. As shown later in Chapter 4, we can also support this kind of model in the Nola style.

Remark 3.5 (Why the Invariant Loses the Power under a Later). The invariant assertion $\boxed{P}^{\mathcal{N}}$ under the later modality, $\triangleright \boxed{P}^{\mathcal{N}}$, does *not* have the power to access the shared content P, as mentioned in § 1.4.

First, the invariant assertion $\boxed{P}^{\mathcal{N}}$ is *not timeless* because the camera iInv for Iris's invariant mechanism is not discrete (see own-ra-timeless, § 2.2.1).

Moreover, the later modality $\triangleright$ on the invariant assertion *weakens the agreement* between P and $I\,\iota$ observed by the resource $[\iota := \text{ag}\,(\text{next}\,P)]$ in the invariant assertion $\boxed{P}^{\mathcal{N}}$ (3.2) into $\triangleright\triangleright(P \stackrel{\sim}{=} I\,\iota)$, instead of $\triangleright(P \stackrel{\sim}{=} I\,\iota)$. While this doubly later-weakened equality $\triangleright\triangleright(P \stackrel{\sim}{=} I\,\iota)$ may still allow taking out the content (getting $\triangleright\triangleright P$ instead of $\triangleright P$), it does not have the power to *restore* the content $\triangleright I\,\iota$ for the world satisfaction Wiinv.

Model of the Hoare triples For a better understanding, we also present the semantics of the Hoare triples.

The partial Hoare triple $\{P\}\,e\,\{\Psi\}_{\mathcal{E}} \in \mathit{iProp}$ is defined as the persistent implication $\Box(P \twoheadrightarrow * \text{pwp}\;e\,\{\Psi\}_{\mathcal{E}})$ from the precondition P to the partial weakest precondition $\text{pwp}\;e\,\{\Psi\}_{\mathcal{E}} \in \mathit{iProp}$. The partial weakest precondition is roughly defined as follows, omitting the stuckness check and concurrency altogether:

$$\begin{aligned}\text{pwp}\;e\,\{\Psi\}_{\mathcal{E}} \quad &\triangleq \quad \text{roughly} \quad \big(\exists v \in \mathit{Val}\ \text{s.t.}\ e = v.\ \Rrightarrow_{\mathcal{E}} \Psi\,v\big) \ \lor \\ &\Big(\forall H.\ \ \text{heap}\,H \twoheadrightarrow\!*\ {}_{\mathcal{E}}\!\Rrightarrow_{\varnothing} \forall (e', H') \hookleftarrow (e, H).\ \triangleright\, {}_{\varnothing}\!\Rrightarrow_{\mathcal{E}} \big(\text{heap}\,H' * \text{pwp}\;e\,\{\Psi\}_{\mathcal{E}}\big)\Big)\end{aligned} \tag{3.5}$$

Here $\text{heap}\,H \in \mathit{iProp}$ exclusively asserts that the global heap memory state is $H \in \mathit{Heap}$ (§ 2.2.3). Notably, the fancy update $\Rrightarrow$ is used in this definition. The frame rule phoare-frame comes from the frame rule for the fancy update $\Rrightarrow$-frame.

Note that this definition (3.5) is *step-indexed* in that the *later modality* $\triangleright$ *guards* the self-reference to pwp after one step of reduction. Thanks to the guard by the later modality $\triangleright$, which is *contractive*, this is a *guarded recursion* and the solution to the equation (3.5) *uniquely exists* (recall Theorem 2.1, § 2.1.2). This can roughly be understood as a *coinductive* definition, like (1.10) in § 1.2.3.

Similarly, the *total* Hoare triple $[P]\,e\,[\Psi]_{\mathcal{E}}$ is defined as $\Box(P \twoheadrightarrow * \text{twp}\;e\,[\Psi]_{\mathcal{E}})$, where the *total* weakest precondition $\text{pwp}\;e\,\{\Psi\}_{\mathcal{E}}$ is roughly defined as follows:

$$\begin{aligned}\text{twp}\;e\,[\Psi]_{\mathcal{E}} \quad &\triangleq_{\mu} \quad \text{roughly} \quad \big(\exists v \in \mathit{Val}\ \text{s.t.}\ e = v.\ \Rrightarrow_{\mathcal{E}} \Psi\,v\big) \ \lor \\ &\Big(\forall H.\ \ \text{heap}\,H \twoheadrightarrow\!*\ {}_{\mathcal{E}}\!\Rrightarrow_{\varnothing} \forall (e', H') \hookleftarrow (e, H).\ {}_{\varnothing}\!\Rrightarrow_{\mathcal{E}} \big(\text{heap}\,H' * \text{twp}\;e\,[\Psi]_{\mathcal{E}}\big)\Big)\end{aligned} \tag{3.6}$$

Unlike the partial weakest precondition (3.5), the total weakest precondition is defined by the *least fixed point* and thus is *not* step-indexed, having no later modality $\triangleright$ inside.

3.2 Nola's Later-Free Invariants

Now we introduce Nola's later-free invariant mechanism.

Key Idea First we present our key idea, which we roughly explained in § 1.5.

The source of the later modality $\triangleright$ for Iris's invariant mechanism was that the camera iInv for the invariant depends on the Iris proposition *iProp* (3.1) (§ 3.1.3), while *iProp* itself depends on the component cameras including iInv (§ 2.2.1), as explained by a rough model (1.19) in § 1.4.

Nola eliminates this circular dependency by replacing iInv with a new *resource algebra* $\text{Inv}_{\mathit{nProp}}$, which depends only on a *syntactic* data type of propositions *nProp* to be shared in the invariant machinery. The resource algebra gives rise to a new *invariant token* $\mathsf{inv}^{\mathcal{N}}\, P$, which takes a syntactic proposition $P \in nProp$ instead of a semantic one $P \in iProp$. Separately we supply the *semantic interpretation* $[\![\ ]\!]\colon nProp \to iProp$ of syntactic propositions $P \in nProp$ into semantic Iris propositions $[\![P]\!] \in iProp$. Nola's invariant mechanism works with an *extended fancy update* ${}_{\mathcal{E}}{\Rrightarrow}^{\mathsf{Winv}\,[\![\ ]\!]}_{\mathcal{E}'}$ enriched with a new *world satisfaction* $\mathsf{Winv}\,[\![\ ]\!]$, which is parameterized over the semantic interpretation $[\![\ ]\!]$ of *nProp*. In this way, we can provide *later-free proof rules*.

Roughly speaking, the invariant mechanism of Nola is parameterized with:

- *nProp*, the syntactic data type of propositions to be shared by the invariant machinery; and
- $[\![\ ]\!]\colon nProp \to iProp$, the semantic interpretation of *nProp*.

But it is vital to understand more precise *dependencies* between the notions, which we discuss later.

Subsection 3.2.1 presents the *extended fancy update* ${}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'}$ and the *extended Hoare triples* $\{P\}\, e\, \{\Psi\}^{W}_{\mathcal{E}}$, $[P]\, e\, [\Psi]^{W}_{\mathcal{E}}$ with a *custom world satisfaction* W. Subsection 3.2.2 presents Nola's invariant mechanism. Subsection 3.2.3 presents the model for Nola's invariant mechanism.

3.2.1 Extended Fancy Update and Hoare Triples

For Nola's later-free mechanisms for propositional sharing, we newly introduce the *extended fancy update modality* ${}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'} P$ and the *extened Hoare triples* $\{P\}\, e\, \{\Psi\}^{W}_{\mathcal{E}}$, $[P]\, e\, [\Psi]^{W}_{\mathcal{E}}$ with a *custom world satisfaction* $W \in iProp$.

Extended Fancy Update We introduce the *extended fancy update modality* ${}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'} P$ with a *custom world satisfaction* $W \in iProp$, defined as follows:

$$ {}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'} P \quad \triangleq \quad W \mathrel{-\!\!*} {}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'} (W * P) $$

This can be understood by analogy with the *state monad*, where W is the global mutable state. Recall that the original fancy update ${}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'} P$ is modeled with the world satisfaction Wiinv (3.4) (§ 3.1.3). The extended fancy update ${}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'}$ adds W to that world satisfaction. This new modality satisfies the proof rules like the original fancy update (§ 3.1), such as the following:

$$ \frac{P \vDash Q}{{}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'} P \vDash {}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'} Q}\ {\Rrightarrow}\textsc{w-mono} \qquad P \vDash {\Rrightarrow}^{W}_{\mathcal{E}} P \quad {\Rrightarrow}\textsc{w-intro} $$

$$ {}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'}\, {}_{\mathcal{E}'}{\Rrightarrow}^{W}_{\mathcal{E}''} P \vDash {}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}''} P \quad {\Rrightarrow}\textsc{w-trans} $$

$$ \left({}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'} P\right) * Q \vDash {}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'} (P * Q) \quad {\Rrightarrow}\textsc{w-frame} $$

$$ {}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'} P \vDash {}_{\mathcal{E}+\mathcal{E}''}{\Rrightarrow}^{W}_{\mathcal{E}'+\mathcal{E}''} P \quad {\Rrightarrow}\textsc{w-mask-frame} $$

$$ \diamond\, {}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'} P = {}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'} P \quad \diamond\text{-}{\Rrightarrow}\textsc{w} \qquad {}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'} \diamond P = {}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'} P \quad {\Rrightarrow}\textsc{w-}\diamond $$

We also have the following rule for expanding the world satisfaction:

$$ \frac{W' = W * W_{+}}{{}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'} P \vDash {}_{\mathcal{E}}{\Rrightarrow}^{W'}_{\mathcal{E}'} P}\ {\Rrightarrow}\textsc{w-expand} $$

The premise $W' \vDash W * (W \mathrel{-\!\!*} W')$ says that W is a part of the world satisfaction W'. Note that the new modality with $W = \top$ agrees with the original fancy update:

$$ {}_{\mathcal{E}}{\Rrightarrow}^{\top}_{\mathcal{E}'} P = {}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}'} P \quad {\Rrightarrow}\textsc{w-}\top $$

Extended Hoare Triples We also introduce *extended Hoare triples* $\{P\}\, e\, \{\Psi\}^{W}_{\mathcal{E}}$, $[P]\, e\, [\Psi]^{W}_{\mathcal{E}}$ with a *custom world satisfaction* $W \in \mathit{iProp}$. They use the extended fancy update $\mathrel{|\!\!\Rrightarrow}^{W}$ instead of $\mathrel{|\!\!\Rrightarrow}$ in their definition (recall (3.5) and (3.6)).

The extended Hoare triples absorb the extended fancy update $\mathrel{|\!\!\Rrightarrow}^{W}$ like $\mathrel{|\!\!\Rrightarrow}$-PHOARE etc.:

$$\frac{\{P\}\, e\, \{\Psi\}^{W}_{\mathcal{E}}}{\{\mathrel{|\!\!\Rrightarrow}^{W}_{\mathcal{E}} P\}\, e\, \{\Psi\}^{W}_{\mathcal{E}}}\ \textsc{$\mathrel{|\!\!\Rrightarrow}$W-HoareW} \qquad \frac{[P]\, e\, [\Psi]^{W}_{\mathcal{E}}}{[\mathrel{|\!\!\Rrightarrow}^{W}_{\mathcal{E}} P]\, e\, [\Psi]^{W}_{\mathcal{E}}}\ \textsc{$\mathrel{|\!\!\Rrightarrow}$W-THoareW}$$

$$\frac{\{P\}\, e\, \{\lambda v.\ \mathrel{|\!\!\Rrightarrow}^{W}_{\mathcal{E}} \Psi\, v\}^{W}_{\mathcal{E}}}{\{P\}\, e\, \{\Psi\}^{W}_{\mathcal{E}}}\ \textsc{HoareW-$\mathrel{|\!\!\Rrightarrow}$W} \qquad \frac{[P]\, e\, [\lambda v.\ \mathrel{|\!\!\Rrightarrow}^{W}_{\mathcal{E}} \Psi\, v]^{W}_{\mathcal{E}}}{[P]\, e\, [\Psi]^{W}_{\mathcal{E}}}\ \textsc{THoareW-$\mathrel{|\!\!\Rrightarrow}$W}$$

We also have the following rules for expanding the world satisfaction of Hoare triples, like $\mathrel{|\!\!\Rrightarrow}$W-EXPAND:

$$\frac{W' = W * W_{+} \qquad \{P\}\, e\, \{\Psi\}^{W}_{\mathcal{E}}}{\{P\}\, e\, \{\Psi\}^{W'}_{\mathcal{E}}}\ \textsc{HoareW-Expand}$$

$$\frac{W' = W * W_{+} \qquad [P]\, e\, [\Psi]^{W}_{\mathcal{E}}}{[P]\, e\, [\Psi]^{W'}_{\mathcal{E}}}\ \textsc{THoareW-Expand}$$

Note that the extended Hoare triples with $W = \top$ agree with the original Hoare triples:

$$\{P\}\, e\, \{\Psi\}^{\top}_{\mathcal{E}} = \{P\}\, e\, \{\Psi\}_{\mathcal{E}} \quad \textsc{HoareW-$\top$}$$

$$[P]\, e\, [\Psi]^{\top}_{\mathcal{E}} = [P]\, e\, [\Psi]_{\mathcal{E}} \quad \textsc{THoareW-$\top$}$$

Adequacy of Extended Hoare Triples The extended Hoare triples $\{P\}\, e\, \{\Psi\}^{W}_{\mathcal{E}}$, $[P]\, e\, [\Psi]^{W}_{\mathcal{E}}$ with a custom world satisfaction W satisfy the following adequacy theorems, which extend Theorem 3.1 and Theorem 3.2 in § 3.1.

Theorem 3.6 (Adequacy of the New Partial Hoare Triple). *If*

$$\forall \gamma_{\text{Heap}}, \gamma_{\text{Inv}}.\ \ \models \mathrel{|\!\!\Rrightarrow}_{\mathit{InvName}} \left(\exists W.\ W * \{\top\}\, e\, \{\phi\}^{W}_{\mathit{InvName}}\right)$$

holds for a pure postcondition ϕ: $\mathit{Val} \to \mathit{Prop}$,

- *any execution of the program e* never gets stuck*, and*
- *whenever the program e terminates with a value $v \in \mathit{Val}$, the* postcondition *$\phi\, v \in \mathit{Prop}$ holds.*

Theorem 3.7 (Termination Adequacy of the New Total Hoare Triple). *If*

$$\forall \gamma_{\text{Heap}}, \gamma_{\text{Inv}}.\ \ \models \mathrel{|\!\!\Rrightarrow}_{\mathit{InvName}} \left(\exists W.\ W * [\top]\, e\, [\lambda\_.\, \top]^{W}_{\mathit{InvName}}\right)$$

holds, then the program e always terminates.

The premise is existentially quantified over the *custom world satisfaction* W and requires to allocate W outside the target Hoare triple, or before the program e is executed.

3.2.2 Nola's Later-Free Invariants

Now we are ready to present Nola's invariant mechanism.

Proof Rules To begin with, Nola introduces an *invariant token* $\mathsf{inv}^{\mathcal{N}}\, P \in \mathit{iProp}$ for a syntactic proposition $P \in \mathit{nProp}$. Importantly, the invariant token $\mathsf{inv}^{\mathcal{N}}\, P$ is *persistent*:

$$\mathsf{inv}^{\mathcal{N}}\, P \;\text{ is persistent} \qquad \textsc{inv-persist}$$

Nola's invariant mechanism introduces a *world satisfaction* $\mathsf{Winv}\,[\![\,]\!] \in \mathit{iProp}$, which depends on the semantic interpretation $[\![\,]\!] \colon \mathit{nProp} \to \mathit{iProp}$ of syntactic propositions $P \in \mathit{nProp}$. Nola's invariants are used with the extended fancy update $\mathrel{\dot{\Rrightarrow}}^{\mathsf{Winv}\,[\![\,]\!]}$ with this world satisfaction $\mathsf{Winv}\,[\![\,]\!]$.

We can allocate an invariant $\mathsf{inv}^{\mathcal{N}}\, P$ by storing the *interpretation* $[\![P]\!]$ of the syntactic proposition P:

$$[\![P]\!] \;\vDash \Rrightarrow^{\mathsf{Winv}\,[\![\,]\!]}_{\varnothing}\; \mathsf{inv}^{\mathcal{N}}\, P \qquad \textsc{inv-alloc}$$

This is very similar to the rule for Iris's invariants IINV-ALLOC, but uses the extended fancy update $\Rrightarrow^{\mathsf{Winv}\,[\![\,]\!]}$ instead of $\Rrightarrow$. Also, we store the *interpretation* $[\![P]\!]$ of the syntactic proposition P. As a stronger version of INV-ALLOC, we also have the following rule:

$$\mathsf{inv}^{\mathcal{N}}\, P \mathrel{-\!\!*} [\![P]\!] \;\vDash \Rrightarrow^{\mathsf{Winv}\,[\![\,]\!]}_{\varnothing}\; \mathsf{inv}^{\mathcal{N}}\, P \qquad \textsc{inv-alloc-rec}$$

This means that we can assume the invariant assertion $\mathsf{inv}^{\mathcal{N}}\, P$ before we store $[\![P]\!]$ for the invariant.

We can get access to the shared content $[\![P]\!]$ of an invariant $\mathsf{inv}^{\mathcal{N}}\, P$ with respect to the extended fancy update $\Rrightarrow^{\mathsf{Winv}\,[\![\,]\!]}$ as follows:

$$\frac{[\![P]\!] * Q \;\vDash \Rrightarrow^{\mathsf{Winv}\,[\![\,]\!]}_{\mathcal{E}}\; \big([\![P]\!] * R\big)}{\mathsf{inv}^{\mathcal{N}}\, P * Q \;\vDash \Rrightarrow^{\mathsf{Winv}\,[\![\,]\!]}_{\mathcal{N}+\mathcal{E}}\; R} \qquad \textsc{inv-acc}$$

Notably, we have *no later modality* $\triangleright$ here. Note that $P \in \mathit{nProp}$ is a syntactic proposition while $Q, R \in \mathit{iProp}$ are semantic Iris propositions. More generally, we have the following access rule for a mask-changing fancy update, just like IINV-ACC-CH:

$$\mathsf{inv}^{\mathcal{N}}\, P \;\vDash {}_{\mathcal{N}}\!\Rrightarrow^{\mathsf{Winv}\,[\![\,]\!]}_{\varnothing} \Big([\![P]\!] * \big([\![P]\!] \mathrel{-\!\!*} {}_{\varnothing}\!\Rrightarrow^{\mathsf{Winv}\,[\![\,]\!]}_{\mathcal{N}}\, \top\big)\Big) \qquad \textsc{inv-acc-ch}$$

The extended Hoare triples with Nola's world satisfaction $W = \mathsf{Winv}\,[\![\,]\!]$ satisfy the following later-free access rules on Nola's invariants, which can be derived from the access rule over the fancy update INV-ACC-CH:

$$\frac{\big\{[\![P]\!] * Q\big\}\, e\, \big\{\lambda v.\; [\![P]\!] * \Psi\, v\big\}^{\mathsf{Winv}\,[\![\,]\!]}_{\mathcal{E}} \qquad e \text{ is atomic}}{\big\{\mathsf{inv}^{\mathcal{N}}\, P * Q\big\}\, e\, \big\{\Psi\big\}^{\mathsf{Winv}\,[\![\,]\!]}_{\mathcal{N}+\mathcal{E}}} \qquad \textsc{phoare-inv}$$

$$\frac{\big[[\![P]\!] * Q\big]\, e\, \big[\lambda v.\; [\![P]\!] * \Psi\, v\big]^{\mathsf{Winv}\,[\![\,]\!]}_{\mathcal{E}} \qquad e \text{ is atomic}}{\big[\mathsf{inv}^{\mathcal{N}}\, P * Q\big]\, e\, \big[\Psi\big]^{\mathsf{Winv}\,[\![\,]\!]}_{\mathcal{N}+\mathcal{E}}} \qquad \textsc{thoare-inv}$$

We also have the following rule for initializing the invariant machinery and acquiring the world satisfaction $\mathsf{Winv}\,[\![\,]\!]$:

$$\vDash \dot{\Rrightarrow}\, \big(\exists \gamma_{\mathrm{INV}}.\; \forall [\![\,]\!].\; \mathsf{Winv}\,[\![\,]\!]\big) \qquad \textsc{winv-alloc}$$

It takes a fresh ghost name $\gamma_{\mathrm{INV}} \in \mathit{GhostName}$ for Nola's invariant mechanism. The invariant token $\mathsf{inv}^{\mathcal{N}}\, P$ and the world satisfaction $\mathsf{Winv}\,[\![\,]\!]$ actually implicitly depend on this ghost name γ_{INV}. The key is the universal quantification over the interpretation $[\![\,]\!]$ here. Thanks to this, we can perform WINV-ALLOC to get γ_{INV}, construct the interpretation $\llbracket\ \rrbracket$ depending on it, and finally get the world satisfaction by instantiating $\forall\llbracket\ \rrbracket.\ \mathsf{Winv}\,\llbracket\ \rrbracket$ with that constructed $\llbracket\ \rrbracket$. This creation of the world satisfaction WINV-ALLOC is essential to satisfy the premise of the *adequacy theorems* Theorem 3.6 and Theorem 3.7 for the extended Hoare triples with a custom world satisfaction.

Also, the new invariant token $\mathsf{inv}^{\mathcal{N}}\,P$ is *timeless*:

$$\mathsf{inv}^{\mathcal{N}}\,P \text{ is timeless} \quad \textsc{inv-timeless}$$

Still, we usually do not care about this fact, because the accesses to Nola's invariants (INV-ACC etc.) and other mechanisms for propositional sharing are no longer weakened by the later modality $\triangleright$.

Dependencies Notably, the resource algebra for Nola's invariants $\textsc{Inv}_{nProp}$ and the invariant token $\mathsf{inv}^{\mathcal{N}}\,P$ depend only on the syntactic data type *nProp*, *not* on the semantic interpretation $\llbracket\ \rrbracket$. What depends on the semantic interpretation $\llbracket\ \rrbracket$ is basically limited to the *extended fancy update* ${}_{\mathcal{E}}{\Rrightarrow}^{\mathsf{Winv}\,\llbracket\ \rrbracket}_{\mathcal{E}'}$, or its *world satisfaction* $\mathsf{Winv}\,\llbracket\ \rrbracket$.

The dependencies between the components from the Nola framework and its user can be summarized as follows:

$$\begin{array}{ccccccc}
\boxed{\mathsf{User}} & & \boxed{\mathsf{Library}} & & \boxed{\mathsf{User}} & & \boxed{\mathsf{Library}} \\
 & & \textsc{Inv}\ nProp & & & & \mathsf{Winv}\,\llbracket\ \rrbracket \\
nProp & \longrightarrow & iProp & \longrightarrow & \llbracket\ \rrbracket & \longrightarrow & \\
 & & \mathsf{inv} & & & & \text{proof rules}
\end{array}$$

More precisely, the interactions between the user and the Nola framework can be described as follows:

1. The *user* constructs the *syntactic data type nProp* for propositions to be shared by the invariant mechanism.
2. The *Nola framework* provides the *resource algebra* $\textsc{Inv}_{nProp}$ for the invariant mechanism.
3. The *user* adds $\textsc{Inv}_{nProp}$ to the component cameras of the Iris proposition *iProp* and takes the ghost name $\gamma_{\textsc{Inv}}$ for the invariant mechanism.
4. The *Nola framework* provides the *invariant token* $\mathsf{inv}^{\mathcal{N}}\,P \in iProp$ that stores a syntactic proposition $P \in nProp$.
5. The *user* defines the *semantic interpretation* $\llbracket\ \rrbracket \colon nProp \to iProp$. Importantly, this can depend on the invariant token $\mathsf{inv}^{\mathcal{N}}$.
6. The *Nola framework* provides the *world satisfaction* $\mathsf{Winv}\,\llbracket\ \rrbracket \in iProp$ for the invariant mechanism. The *Nola framework* also provides proof rules for the invariant $\mathsf{inv}^{\mathcal{N}}$ that involve the extended fancy update ${\Rrightarrow}^{\mathsf{Winv}\,\llbracket\ \rrbracket}$ of the world satisfaction $\mathsf{Winv}\,\llbracket\ \rrbracket$.

Remarkably, we get the *invariant token* $\mathsf{inv}^{\mathcal{N}}\,P$ before we construct the interpretation $\llbracket\ \rrbracket$. The token $\mathsf{inv}^{\mathcal{N}}\,P$ only knows the *syntactic* data P and not its interpretation. The semantic interpretation $\llbracket\ \rrbracket$ is considered only by the world satisfaction $\mathsf{Winv}\,\llbracket\ \rrbracket$ of the fancy update ${\Rrightarrow}^{\mathsf{Winv}\,\llbracket\ \rrbracket}$. This is the key to supporting *nested invariants*, as we see later in § 3.3.3.

Note that the data type *nProp* should be constructed independently of *iProp*. This is because *iProp* depends on the component cameras, which include the resource algebra $\textsc{Inv}_{nProp}$ that depends on *nProp*. In particular, we cannot set $nProp \triangleq_{?} iProp$ and $\llbracket P \rrbracket \triangleq_{?} P$. Indeed, that leads to a contradiction by the known paradox of later-free invariants (§ 3.4.1).

3.2.3 Model

We present the model for Nola's invariant mechanism. The model is fairly close to Iris's invariant mechanism.

Invariant Resource Algebra To begin with, the resource algebra $\textsc{Inv}_{nProp}$ for Nola's invariants is defined as follows:

$$\textsc{Inv}_{nProp} \quad \triangleq \quad \textsc{Auth}\left(\mathit{GhostName} \overset{\text{fin}}{\rightharpoonup} \textsc{Ag}\ \mathit{nProp}\right)$$

This is very similar to the camera $\textsc{iInv}$ for Iris's invariant (3.1) (§ 3.1.3). The only difference is that the target of the agreement $\textsc{Ag}$ is syntactic propositions *nProp*. Because it does not depend on *iProp*, we do not need the later constructor ▶, which is the key to supporting later-free proof rules.

Nola's Invariant Connective The invariant token inv for *nProp* is modeled as follows:

$$\mathsf{inv}^{\mathcal{N}}\,P \quad \triangleq \quad \exists \iota \in \mathcal{N}.\ \boxed{\circ\,[\iota := \mathsf{ag}\,P]}^{\gamma_{\textsc{Inv}}}_{\textsc{Inv}_{nProp}}$$

We use the ghost name $\gamma_{\textsc{Inv}}$ for Nola's invariant mechanism, taken by the rule WINV-ALLOC. This is analogous to Iris's invariant $\boxed{P}^{\mathcal{N}}$ (3.2) (§ 3.1.3).

Nola's World Satisfaction The world satisfaction $\mathsf{Winv}\,[\![\ ]\!]$ for Nola's invariant is defined as follows:

$$\mathsf{Winv}\,[\![\ ]\!] \quad \triangleq \quad \exists I.\ \boxed{\bullet\,\mathsf{ag}\,I}^{\gamma_{\textsc{Inv}}}_{\textsc{Inv}_{nProp}} \ * \ \mathop{\circledast}_{\iota \in \operatorname{dom} I} \left(\left([\![I\,\iota]\!] * \boxed{\{\iota\}}^{\gamma_{\textsc{Dis}}}_{\textsc{Dis}}\right) \vee \boxed{\{\iota\}}^{\gamma_{\textsc{En}}}_{\textsc{En}}\right) \tag{3.7}$$

This is analogous to Iris's world satisfaction Wiinv (3.3) (§ 3.1.3). Nola's world satisfaction is existentially quantified over $I : \mathit{InvName} \overset{\text{fin}}{\rightharpoonup} \mathit{nProp}$, a finite map to *nProp* instead of *iProp*. We write $\mathsf{ag}\,I$ for the finite map $\lambda\iota \in \operatorname{dom} I.\ \mathsf{ag}\,(I\,\iota)$. Notably, Nola's world satisfaction $\mathsf{Winv}\,[\![\ ]\!]$ stores the *semantic interpretation* $[\![I\,\iota]\!]$ obtained by the parameter $[\![\ ]\!]$ for each closed invariant, instead of $\triangleright I\,\iota$ in Wiinv. Nola's world satisfaction $\mathsf{Winv}\,[\![\ ]\!]$ does not need the later modality $\triangleright$, because the agreement observed by Nola's invariant token $\mathsf{inv}^{\mathcal{N}}$ is not weakened by the later modality $\triangleright$. As a result, we can provide later-free proof rules like INV-ACC.

Given the model above, the soundness of the proof rules is straightforward.

Theorem 3.8. *The proof rules for Nola's invariants (e.g.,* INV-ACC *and* WINV-ALLOC*) are sound for every nProp and* $[\![\ ]\!]$.

Proof. Straightforward. The proof goes like Iris's later-requiring invariant, except that we are free from the constructor next. To prove the rule WINV-ALLOC, we initialize the map I inside the world satisfaction $\mathsf{Winv}\,[\![\ ]\!]$ to the empty map $\varnothing$, which makes the interpretation $[\![\ ]\!]$ is irrelevant. □

3.3 Example: Linked List Mutation

To see how to use Nola's invariant mechanism, let us consider a simple example.

3.3.1 Verification Target

We verify *termination* of iterative *mutation* of a singly linked list, for an *unbounded number* of elements, among *unboundedly many threads.*

Basic Iterative Mutation For iterative mutation, we first consider the following recursive function $\mathsf{iter}_{f,c}(\ell)$:

$$\mathsf{fun}\ \ \mathsf{iter}_{f,c}(\ell)\ \big\{\ \mathsf{if}\ \ !c \neq 0\ \ \mathsf{then}\ \big(f(\ell);\ c \leftarrow !c - 1;\ \mathsf{iter}_{f,c}(!(\ell+1))\big)\ \big\} \tag{3.8}$$

The iteration function $\mathsf{iter}_{f,c}(\ell)$ takes a pointer ℓ to the singly linked list to be mutated, which changes in the course of recursion. The function is also parameterized by a mutation function f and a pointer to a counter c. Then it checks whether the counter value $!c$ stored at c is non-zero. Only in the case the value is zero does the function terminate immediately. The function mutates the head element at ℓ by calling the mutation function f. Then it decrements the counter value stored at c by one. Finally, it recursively calls the function $\mathsf{iter}_{f,c}$ on the tail list at $!(\ell+1)$.

In summary, the iteration function $\mathsf{iter}_{f,c}(\ell)$ calls the function f over the first $!c$ elements of the singly linked list. We want to verify that the iterative mutation $\mathsf{iter}_{f,c}(\ell)$ always terminates, assuming that the mutation by f always terminates.

Here, the function f should not perform invalid memory access for the termination of $\mathsf{iter}_{f,c}(\ell)$. In particular, if f can bump up the counter value at c, then the function $\mathsf{iter}_{f,c}(\ell)$ can possibly never terminate. Such constraints can be easily expressed in separation logic.

With Non-Determinism We also want to perform the iterative mutation for an *unbounded number* of elements. We consider the following function $\mathsf{iter}^{\mathsf{nd}}_{f}(\ell)$:

$$\mathsf{fun}\ \ \mathsf{iter}^{\mathsf{nd}}_{f}(\ell)\ \big\{\ \mathsf{let}\ \ c := \mathsf{ref}\,\mathsf{ndnat}\ \ \mathsf{in}\ \ \mathsf{iter}_{f,c}(\ell)\ \big\} \tag{3.9}$$

The function first takes a *non-deterministic* natural number ndnat and allocates a fresh memory cell c initialized with that number. Then it calls the iterative mutation function $\mathsf{iter}_{f,c}(\ell)$ using c as the counter. Notably, the function $\mathsf{iter}^{\mathsf{nd}}_{f}(\ell)$ can perform mutation by f *unboundedly many times*, because ndnat can return any *unboundedly large* natural number. Still, we want to verify that this *unbounded* iterative mutation $\mathsf{iter}^{\mathsf{nd}}_{f}(\ell)$ always terminates.

With Concurrency Furthermore, we want to perform unbounded iterative mutations *concurrently* with *many threads*. We consider the following recursive function $\mathsf{iterforks}_{f,c',\ell}()$:

$$\mathsf{fun}\ \ \mathsf{iterforks}_{f,c',\ell}()\ \big\{\ \mathsf{if}\ !c' \neq 0\ \mathsf{then}\ \big(\mathsf{fork}\,\{\,\mathsf{iter}^{\mathsf{nd}}_{f}(\ell)\,\};\ c' \leftarrow !c' - 1;\ \mathsf{iterforks}_{f,c',\ell}()\,\big)\ \big\} \tag{3.10}$$

The function *forks threads* that perform the unbounded iterative mutation $\mathsf{iter}^{\mathsf{nd}}_{f}(\ell)$, where the number of forked threads is specified by the counter at c'. We want to verify that this concurrent iterative mutation $\mathsf{iterforks}_{f,c',\ell}()$ always terminates, in the sense that *all the threads terminate.*

We can even perform that with *unboundedly many threads.* We consider the following recursive function $\mathsf{iterforks}^{\mathsf{nd}}_{f,\ell}()$:

$$\mathsf{fun}\ \ \mathsf{iterforks}^{\mathsf{nd}}_{f,\ell}()\ \big\{\ \mathsf{let}\ \ c' := \mathsf{ref}\,\mathsf{ndnat}\ \ \mathsf{in}\ \ \mathsf{iterforks}_{f,c',\ell}()\ \big\} \tag{3.11}$$

It calls the function $\mathsf{iterforks}_{f,c',\ell}()$ by a counter at c' initialized with an *unbounded natural number* ndnat. We want to verify that even $\mathsf{iterforks}^{\mathsf{nd}}_{f,\ell}()$ always terminates.

3.3.2 Problem with Iris's Invariants

Let us first try to verify the termination of these functions using Iris's invariants.

Modeling the Singly Linked List What should be the assertion for the singly linked list? In order to perform mutation among multiple threads, the singly linked list should be both *sharable* and *mutable*. Also, the singly linked list should be *infinite*, because the number of elements to be mutated is unbounded. Infinite singly linked lists can be constructed from *cyclic references*, because lists we consider are sharable.

For that, we can consider the following *persistent* Iris assertion $\mathsf{ilist}^{\mathcal{N}}\ \Phi\ \ell \in iProp$ for a shared mutable infinite list of the head location $\ell \in Loc$, parameterized with the invariant predicate $\Phi\colon Loc \to iProp$:

$$\mathsf{ilist}^{\mathcal{N}}\ \Phi\ \ell \quad \triangleq \quad \boxed{\Phi\ \ell}^{\mathcal{N}} * \boxed{\exists \ell' \in Loc.\ (\ell+1) \mapsto \ell' * \mathsf{ilist}^{\mathcal{N}}\ \Phi\ \ell'}^{\mathcal{N}} \tag{3.12}$$

For the head, it persistently asserts the *invariant* $\boxed{\Phi\ \ell}^{\mathcal{N}}$ about the head location ℓ. For the tail, it persistently asserts the *invariant* that $\ell+1$ always points to a some location $\ell' \in Loc$ satisfying the list predicate $\mathsf{ilist}^{\mathcal{N}}\ \Phi\ \ell'$ itself.

Note that the self-reference to $\mathsf{ilist}^{\mathcal{N}}\ \Phi$ in (3.12) is *guarded* by Iris's invariant connective $\boxed{-}^{\mathcal{N}}$. Technically, the connective $\boxed{-}^{\mathcal{N}}$ is *contractive* because the next constructor in its model (3.2) (§ 3.1.3) is contractive. So the equation (3.12) forms *guarded recursion* and its solution *uniquely exists* (recall Theorem 2.1, § 2.1.2). This can roughly be understood as a coinductive definition.

Also note that the infinite list assertion $\mathsf{ilist}^{\mathcal{N}}\ \Phi\ \ell$ is an example of *unboundedly nested invariants* (in this case, it is even *infinitely* nested).

For example, we can think of an invariant $\Phi = \exists n \in \mathbb{Z}.\ \ell \mapsto 3n$, stating that a multiple of three is stored at ℓ. This invariant is retained by the mutation fun $f(\ell)\ \{\ \mathsf{faa}\ \ell\ 3\ \}$, which adds three to the value stored at ℓ. Indeed, we can prove the following total Hoare triple for any ℓ and n:

$$\left[\ \boxed{\exists n \in \mathbb{Z}.\ \ell \mapsto 3n}^{\mathcal{N}}\ \right]\ \mathsf{faa}\ \ell\ 3\ \left[\ \lambda\_.\ \top\ \right]_{\mathcal{N}} \tag{3.13}$$

Here, the *fetch-and-add* $\mathsf{faa}\ \ell\ n$ is an *atomic* operation that adds n to the number stored at ℓ and returns the old value, satisfying the following Hoare triple rule:

$$\left[\ \ell \mapsto k\ \right]\ \mathsf{faa}\ \ell\ n\ \left[\ \lambda v.\ v = k * \ell \mapsto (k+n)\ \right]_{\varnothing} \qquad \text{THOARE-FAA}$$

Termination Proof Fails The assertion we target for the most basic iterative mutation function $\mathsf{iter}_{f,c}(\ell)$ (3.8) is the following *total* Hoare triple entailment for any natural number $n \in \mathbb{N}$:

$$\frac{\forall \ell.\ \left[\ \boxed{\Phi\ \ell}^{\mathcal{N}}\ \right]\ f(\ell)\ \left[\ \lambda\_.\ \top\ \right]_{\mathcal{N}}}{\left[\ \mathsf{ilist}^{\mathcal{N}}\ \Phi\ \ell * c \mapsto n\ \right]\ \mathsf{iter}_{f,c}(\ell)\ \left[\ \lambda\_.\ c \mapsto 0\ \right]_{\mathcal{N}}.} \tag{3.14}$$

We can satisfy the premise with any kind of mutation, e.g., the fetch-and-add of (3.13).

Unfortunately, the entailment (3.14) cannot be proved. Consider the step case $n > 0$. We can perform the mutation $f(\ell)$ using the premise $\left[\ \boxed{\Phi\ \ell}^{\mathcal{N}}\ \right]\ f(\ell)\ \left[\ \lambda\_.\ \top\ \right]_{\mathcal{N}}$. We can also perform the decrement $c \leftarrow !c - 1$ by updating the given points-to token $c \mapsto n$. The problem occurs when we access the *tail* list $!(\ell+1)$:

$$\left[\ \mathsf{ilist}^{\mathcal{N}}\ \Phi\ \ell\ \right]\ !(\ell+1)\ \left[\ \lambda v.\ \exists \ell' \in Loc \text{ s.t. } v = \ell'.\ \triangleright \left(\ \mathsf{ilist}^{\mathcal{N}}\ \Phi\ \ell'\ \right)\ \right]_{\mathcal{N}} \tag{3.15}$$

We have the *later modality* $\triangleright$ on the tail list $\mathsf{ilist}^{\mathcal{N}}\ \Phi\ \ell'$, due to the *later modality* $\triangleright$ in the access rule THOARE-IINV and the non-timelessness of the assertion $\mathsf{ilist}^{\mathcal{N}}\ \Phi\ \ell'$, stemming from the non-timelessness of invariants $\boxed{-}^{\mathcal{N}}$. Because we are working in the total Hoare triple, we never have the chance to strip off this later modality $\triangleright$, unlike in the partial Hoare triple (recall the discussions in § 1.4). This later modality $\triangleright$ blocks us from verifying the recursive call $\mathsf{iter}_{f,c}(!(\ell+1))$, because we need to obtain the later-free $\mathsf{ilist}^{\mathcal{N}}\ \Phi\ \ell'$.

3.3.3 Solution: Nola's Later-Free Invariants

Using Nola's invariants, we can provide the later-free access rule THOARE-INV and thus prove a later-free version of (3.14), making the verification work smoothly.

Here we present our solution in a high-level manner. See § 8.3 for how verification goes in our Coq mechanization of Nola.

First Step: Determine the Syntax *nProp* For that, we first determine the syntactic data type *nProp*. We should consider what propositions we want to store into Nola's invariants. For example, informally, the following fragment of Iris propositions can be sufficient to describe the desired assertions:

$$\begin{aligned} P, Q \ ::= \ & \forall a \in A.\, P_a \mid \exists a \in A.\, P_a \mid P \wedge Q \mid P \vee Q \mid P \to Q \mid \phi \\ & \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \Box P \mid \dot{\Rrightarrow} P \mid \triangleright P \\ & \mid \ell \overset{q}{\mapsto} v \mid \boxed{P}^{\mathcal{N}} \mid \mathsf{ilist}^{\mathcal{N}}\ \Phi\ \ell \end{aligned}$$

One can also add other logical connectives one wishes to use. (There is a limitation, which we discuss in § 3.4.2.) In particular, we can freely add tokens like the points-to token $\ell \overset{q}{\mapsto} v$ and $\boxed{P}^{\mathcal{N}}$.

We express the above fragment formally as the following *data type nProp* for *syntactic logic formulas*:

$$\begin{aligned} nProp \ni P, Q \ ::= \ & \forall_A\, \Phi \mid \exists_A\, \Phi \mid P \texttt{ and } Q \mid P \texttt{ or } Q \mid P \texttt{ -> } Q \mid \phi \\ & \mid P \texttt{ * } Q \mid P \texttt{ -* } Q \mid \texttt{pers}\, P \mid \texttt{bupd}\, P \mid \texttt{|>}\, P \\ & \mid \ell \overset{q}{\texttt{|->}} v \mid \texttt{inv}^{\mathcal{N}}\, P \mid \texttt{ilist}^{\mathcal{N}}\ \Phi\ \ell \end{aligned} \tag{3.16}$$

To distinguish from Iris propositions, we use the typewriter fonts for the syntactic propositions in *nProp*. For the universal and existential quantifiers $\forall_A\,\Phi$, $\exists_A\,\Phi$, we use *higher-order abstract syntax (HOAS)*, where A is the domain set of the quantification and Φ is a function $A \longrightarrow nProp$. For the universe consistency, the set A here should be taken from a universe that does not include *nProp*.[7]

Second Step: Build the Semantics $[\![\ ]\!]$ Now that we have decided the data type *nProp* (3.16), we can add the resource algebra $\textsc{Inv}_{nProp}$ to the component cameras of the Iris propositions *iProp* (and take the ghost name $\gamma_{\textsc{Inv}}$ by WINV-ALLOC). This provides the invariant token $\mathsf{inv}^{\mathcal{N}}\, P \in iProp$ for $P \in nProp$.

Now we can construct the interpretation $[\![\ ]\!]: nProp \longrightarrow iProp$ by straightforward structural induction, interpreting each constructor of *nProp* as its intended meaning as Iris propositions:

$$[\![\forall_A\,\Phi]\!] \triangleq \forall a \in A.\, [\![\Phi\ a]\!] \qquad [\![\exists_A\,\Phi]\!] \triangleq \exists a \in A.\, [\![\Phi\ a]\!]$$

$$[\![P \texttt{ and } Q]\!] \triangleq [\![P]\!] \wedge [\![Q]\!] \qquad [\![P \texttt{ or } Q]\!] \triangleq [\![P]\!] \vee [\![Q]\!] \qquad [\![P \texttt{ -> } Q]\!] \triangleq [\![P]\!] \to [\![Q]\!]$$

$$[\![\phi]\!] \triangleq \phi$$

$$[\![P \texttt{ * } Q]\!] \triangleq [\![P]\!] * [\![Q]\!] \qquad [\![P \texttt{ -* } Q]\!] \triangleq [\![P]\!] \mathrel{-\!\!*} [\![Q]\!] \qquad [\![\texttt{pers}\, P]\!] \triangleq \Box [\![P]\!]$$

$$[\![\texttt{bupd}\, P]\!] \triangleq \dot{\Rrightarrow} [\![P]\!] \qquad [\![\texttt{|>}\, P]\!] \triangleq \triangleright [\![P]\!]$$

$$[\![\ell \overset{q}{\texttt{|->}} v]\!] \triangleq \ell \overset{q}{\mapsto} v$$

[7] This means that we cannot handle *impredicative quantifiers* like $\forall\, P \in nProp$ in this way. We discuss this problem in § 3.4.2.

$$[\![\mathtt{inv}^{\mathcal{N}}\,P]\!] \triangleq \mathsf{inv}^{\mathcal{N}}\,P \tag{3.17}$$

$$[\![\mathtt{ilist}^{\mathcal{N}}\,\Phi\,\ell]\!] \triangleq \mathsf{inv}^{\mathcal{N}}(\Phi\,\ell) * \mathsf{inv}^{\mathcal{N}}\big(\exists\,\ell'.\ (\ell+1) \mapsto \ell' * \mathtt{ilist}^{\mathcal{N}}\,\Phi\,\ell'\big) \tag{3.18}$$

The invariant constructor $\mathtt{inv}^{\mathcal{N}}\,P$ of *nProp* is interpreted as the invariant token $\mathsf{inv}^{\mathcal{N}}\,P$ (3.17). The token depends only on the syntactic proposition $P \in nProp$, not its interpretation $[\![P]\!] \in iProp$. Remarkably, we can freely *nest* the invariant constructor $\mathtt{inv}^{\mathcal{N}}\,P$, i.e., attain *nested invariants*.

The infinite singly linked list constructor $\mathtt{ilist}^{\mathcal{N}}\,\Phi\,\ell$ is interpreted as a proposition (3.18) that is analogous to (3.12) but uses Nola's invariant token $\mathsf{inv}^{\mathcal{N}}\,P$ instead of Iris's $\boxed{P}^{\mathcal{N}}$. Here, we introduced the shorthand $\exists\,a.\,P_a \triangleq \exists(\lambda a.\,P_a)$ and $\ell \mapsto v \triangleq \ell \overset{1}{\mapsto} v$.

Note that the right-hand side of (3.18) does *not* contain a recursive call of $[\![\;]\!]$. Although the tail list $\mathtt{ilist}^{\mathcal{N}}\,\Phi\,\ell'$ 'recursively' occurs in the right-hand side, it is referred to just as a *syntactic formula*, not as its semantics $[\![\mathtt{ilist}^{\mathcal{N}}\,\Phi\,\ell']\!]$. In a sense, the 'recursive' occurrence of $\mathtt{ilist}^{\mathcal{N}}$ is *guarded* by the invariant connective $\mathtt{inv}^{\mathcal{N}}$.

Also note that the infinite list assertion $\mathtt{ilist}^{\mathcal{N}}\,\Phi\,\ell \in nProp$ thus interpreted can be regarded as an example of *unboundedly nested invariants* (or even *infinitely* nested).

Final Step: Verify with Nola's Invariants Now that we have the semantics $[\![\;]\!]\colon nProp \to iProp$, we can instantiate the world satisfaction $\mathsf{Winv}\,[\![\;]\!]$ of Nola's invariant mechanism with the above-given interpretation $[\![\;]\!]\colon nProp \to iProp$.

Now we can verify the termination of the iterative mutation $\mathsf{iter}_{f,c}(\ell)$ (3.8) without any difficulty. The following total Hoare triple entailment holds for any natural number $n \in \mathbb{N}$:

$$\frac{\forall \ell.\ \big[\,\mathsf{inv}^{\mathcal{N}}(\Phi\,\ell)\,\big]\ f(\ell)\ \big[\lambda\_.\,\top\big]^{\mathsf{Winv}\,[\![\;]\!]}_{\mathcal{N}}}{\big[\,[\![\mathtt{ilist}^{\mathcal{N}}\,\Phi\,\ell]\!] * c \mapsto n\,\big]\ \mathsf{iter}_{f,c}(\ell)\ \big[\,\lambda\_.\ c \mapsto 0\,\big]^{\mathsf{Winv}\,[\![\;]\!]}_{\mathcal{N}}} \tag{3.19}$$

The proof of (3.19) simply goes by induction over $n \in \mathbb{N}$. The key to success is that we can prove the following assertion for accessing the tail list at $!(\ell + 1)$, using THOARE-LOAD and THOARE-INV:

$$\big[\,[\![\mathtt{ilist}^{\mathcal{N}}\,\Phi\,\ell]\!]\,\big]\ !(\ell+1)\ \big[\lambda v.\ \exists \ell' \in Loc \text{ s.t. } v = \ell'.\ [\![\mathtt{ilist}^{\mathcal{N}}\,\Phi\,\ell']\!]\big]^{\mathsf{Winv}\,[\![\;]\!]}_{\mathcal{N}} \tag{3.20}$$

Unlike (3.15), we do not suffer from the later modality $\triangleright$ on the tail list $[\![\mathtt{ilist}^{\mathcal{N}}\,\Phi\,\ell']\!]$, thanks to the later-free access rule THOARE-INV.

We can satisfy the premise of (3.19) with any kind of mutation. For example, we can consider the fetch-and-add like (3.13):

$$\big[\,\mathsf{inv}^{\mathcal{N}}\big(\exists\,n.\,\ell \mapsto 3n\big)\,\big]\ \mathsf{faa}\ \ell\ 3\ \big[\,\lambda\_.\,\top\big]^{\mathsf{Winv}\,[\![\;]\!]}_{\mathcal{N}}$$

Note that we can construct infinite singly linked lists $\mathtt{ilist}^{\mathcal{N}}\,\Phi\,\ell$ from cyclic references, for example:

$$\mathsf{inv}^{\mathcal{N}}(\Phi\,\ell) * (\ell+1) \mapsto \ell \ \vDash \Rrightarrow^{\mathsf{Winv}\,[\![\;]\!]}_{\varnothing}\ [\![\mathtt{ilist}^{\mathcal{N}}\,\Phi\,\ell]\!]$$

$$\mathsf{inv}^{\mathcal{N}}(\Phi\,\ell) * (\ell+1) \mapsto \ell' * \mathsf{inv}^{\mathcal{N}}(\Phi\,\ell') * (\ell'+1) \mapsto \ell \ \vDash \Rrightarrow^{\mathsf{Winv}\,[\![\;]\!]}_{\varnothing}\ [\![\mathtt{ilist}^{\mathcal{N}}\,\Phi\,\ell]\!]$$

We can prove them using the recursive allocation rule INV-ALLOC-REC. Also, we can add an element to an infinite singly linked list:

$$\mathsf{inv}^{\mathcal{N}}(\Phi\,\ell) * (\ell+1) \mapsto \ell' * [\![\mathtt{ilist}^{\mathcal{N}}\,\Phi\,\ell']\!] \ \vDash \Rrightarrow^{\mathsf{Winv}\,[\![\;]\!]}_{\varnothing}\ [\![\mathtt{ilist}^{\mathcal{N}}\,\Phi\,\ell]\!]$$

Based on the termination of $\mathsf{iter}_{f,c}(\ell)$ (3.19), we can easily verify the termination of the other functions we considered. First, we can verify the termination of the unbounded iterative mutation by the function $\mathsf{iter}^{\mathsf{nd}}_{f}(\ell)$ (3.9):

$$\frac{\forall \ell.\ \big[\,\mathsf{inv}^{\mathcal{N}}(\Phi\ \ell)\,\big]\ f(\ell)\ \big[\lambda\_.\top\big]^{\mathsf{Winv}[\![\,]\!]}_{\mathcal{N}}}{\big[\,[\![\mathtt{ilist}^{\mathcal{N}}\ \Phi\ \ell]\!]\,\big]\ \mathsf{iter}^{\mathsf{nd}}_{f}(\ell)\ \big[\lambda\_.\top\big]^{\mathsf{Winv}[\![\,]\!]}_{\mathcal{N}}} \tag{3.21}$$

To prove this, we just combine (3.19) with THOARE-ALLOC the following rule for taking a non-deterministic natural number:

$$\big[\top\big]\ \mathsf{ndnat}\ \big[\,\lambda v.\ \exists n \in \mathbb{N}.\ v = n\,\big]_{\varnothing} \qquad \text{THOARE-NDNAT}$$

We can also verify the termination of concurrent iterative mutation by the function $\mathsf{iterforks}_{f,c',\ell}()$ (3.10), proving the following for any natural number $m \in \mathbb{N}$:

$$\frac{\forall \ell.\ \big[\,\mathsf{inv}^{\mathcal{N}}(\Phi\ \ell)\,\big]\ f(\ell)\ \big[\lambda\_.\top\big]^{\mathsf{Winv}[\![\,]\!]}_{\mathcal{N}}}{\big[\,[\![\mathtt{ilist}^{\mathcal{N}}\ \Phi\ \ell]\!] * c' \mapsto m\,\big]\ \mathsf{iterforks}_{f,c',\ell}()\ \big[\lambda\_.\top\big]^{\mathsf{Winv}[\![\,]\!]}_{\mathcal{N}}} \tag{3.22}$$

The proof goes by induction on $m \in \mathbb{N}$, using (3.21) and the following rule for forking a thread (similar to THOARE-FORK in § 1.2.3):

$$\frac{\big[P\big]\ e\ \big[\lambda\_.\top\big]_{\mathcal{E}}}{\big[P\big]\ \mathsf{fork}\,\{\,e\,\}\ \big[\lambda\_.\top\big]_{\mathcal{E}}} \qquad \text{THOARE-FORK}$$

Finally, we can verify the termination of concurrent iterative mutation among an unbounded number of threads by the function $\mathsf{iterforks}^{\mathsf{nd}}_{f,\ell}()$ (3.11):

$$\frac{\forall \ell.\ \big[\,\mathsf{inv}^{\mathcal{N}}(\Phi\ \ell)\,\big]\ f(\ell)\ \big[\lambda\_.\top\big]^{\mathsf{Winv}[\![\,]\!]}_{\mathcal{N}}}{\big[\,[\![\mathtt{ilist}^{\mathcal{N}}\ \Phi\ \ell]\!]\,\big]\ \mathsf{iterforks}^{\mathsf{nd}}_{f,\ell}()\ \big[\lambda\_.\top\big]^{\mathsf{Winv}[\![\,]\!]}_{\mathcal{N}}}$$

The proof goes by combining (3.22) with THOARE-ALLOC and THOARE-NDNAT.

3.4 Paradoxes and Expressivity

As we have seen, Nola's later-free invariant mechanism is parameterized by *nProp*, the syntactic data type of propositions that can be stored into invariants, and its semantics $[\![\,]\!]\colon \mathit{nProp} \to \mathit{iProp}$. By enriching *nProp* and $[\![\,]\!]$, we can express more Iris propositions in *nProp* and get the power to store them in Nola's invariants. Still, the fact that the data type *nProp* and its interpretation $[\![\,]\!]$ are *well-defined* imposes a limitation on what can be expressed in *nProp*. On the other hand, that limitation naturally protects the later-free invariant mechanism from *paradoxes*. This section clarifies the limit of *nProp*'s expressivity by discussing the paradoxes of later-free invariants (§ 3.4.1) and the definability of the interpretation $[\![\,]\!]$ (§ 3.4.2).

3.4.1 Paradoxes of Later-Free Invariants

Old Paradox There is a known paradox of later-free invariants by Krebbers et al. (2017a, § 5).

They assume an invariant connective $\boxed{P} \in \mathit{iProp}$ ($P \in \mathit{iProp}$) that is persistent and supports the following *later-free* rules for allocation and access:

$$P \vDash \Rrightarrow_{\bullet} \boxed{P} \quad \text{IINV-ALLOC}' \qquad \frac{P * Q \vDash \Rrightarrow_{\varnothing} (P * R)}{\boxed{P} * Q \vDash \Rrightarrow_{\bullet} R} \quad \text{IINV-ACC-NOLATER}$$

Here they assume two types of masks, empty $\varnothing$ and full $\bullet$.[8] For the fancy update $\Rrightarrow_{\mathcal{E}}$, they assume the following rules:

$$\frac{P \vDash Q}{\Rrightarrow_{\mathcal{E}} P \vDash \Rrightarrow_{\mathcal{E}} Q} \qquad P \vDash \Rrightarrow_{\mathcal{E}} P \qquad \Rrightarrow_{\mathcal{E}} \Rrightarrow_{\mathcal{E}} P = \Rrightarrow_{\mathcal{E}} P$$
$$(\Rrightarrow_{\mathcal{E}} P) * Q \vDash \Rrightarrow_{\mathcal{E}} (P * Q) \qquad \Rrightarrow_{\varnothing} P \vDash \Rrightarrow_{\bullet} P \tag{3.23}$$

They also assume two tokens $\dbox{\mathrm{S}}^{\gamma}, \dbox{\mathrm{F}}^{\gamma} \in iProp$ satisfying the following rules:

$$\vDash \Rrightarrow_{\varnothing} \exists \gamma.\, \dbox{\mathrm{S}}^{\gamma} \quad \textsc{s-new} \qquad \dbox{\mathrm{S}}^{\gamma} \vDash \Rrightarrow_{\varnothing} \dbox{\mathrm{F}}^{\gamma} \quad \textsc{s-}\Rrightarrow\textsc{-f}$$
$$\dbox{\mathrm{S}}^{\gamma} * \dbox{\mathrm{F}}^{\gamma} \vDash \bot \quad \textsc{s-f-}\bot \qquad \dbox{\mathrm{F}}^{\gamma} \vDash \dbox{\mathrm{F}} * \dbox{\mathrm{F}}^{\gamma} \quad \textsc{f-dup}$$

Intuitively, the token $\dbox{\mathrm{S}}^{\gamma}$ exclusively asserts that γ is at the *'start'* state and the token $\dbox{\mathrm{F}}^{\gamma}$ non-exclusively asserts thatγ is at the *'finished'* state. The state is initially set to the start state (S-NEW) and can change to the finished state (S-$\Rrightarrow$-F). The state and finished states do not co-exist (S-F-$\bot$). A witness of the finished state can be duplicated (F-DUP).

Under these assumptions, they prove the following, introducing the falsehood $\bot$ under the fancy update $\Rrightarrow_{\bullet}$:

$$\vDash \Rrightarrow_{\bullet} \bot.$$

This contradicts the adequacy of the fancy update modality Theorem 3.4.

To prove this, they constructed the following invariant:

$$\boxed{\dbox{\mathrm{S}}^{\gamma} \lor (\dbox{\mathrm{F}}^{\gamma} * \Box\, \mathrm{big}^{\gamma})} \quad \text{where}$$
$$\mathrm{big}^{\gamma} \triangleq \exists P \in iProp.\ \Box\big(P \mathrel{-\!\!*} \Rrightarrow_{\bullet} \bot\big) * \boxed{\dbox{\mathrm{S}}^{\gamma} \lor (\dbox{\mathrm{F}}^{\gamma} * \Box P)}$$

Unfortunately, this invariant is quite involved. The proposition stored in the invariant contains an *impredicative existential quantifier* $\exists P \in iProp$, a *fancy update modality* $\Rrightarrow_{\bullet}$, and an *invariant* connective $\boxed{-}$. We argue that the real source of the contradiction is unclear from this paradox.

New Paradox Remarkably, we can significantly simplify the above paradox. We just construct the following invariant:

$$\boxed{\dbox{\mathrm{S}}^{\gamma} \lor \Box \Rrightarrow_{\bullet} \bot}$$

Now the real source of the contradiction is pretty clear: the *fancy update modality* $\Rrightarrow_{\bullet}$ stored inside the invariant.

We clarify the statement of our new paradox.

Theorem 3.9 (New Paradox of Later-Free Invariants)**.** *Assume that there is a fancy update modality* $\Rrightarrow_{\mathcal{E}} P$ *(*$\mathcal{E} \in \{\varnothing, \bullet\}, P \in iProp$*) satisfying the rules of* (3.23)*. Also, assume that there are two tokens* $\dbox{\mathrm{S}}^{\gamma}, \dbox{\mathrm{F}}^{\gamma} \in iProp$ *satisfying* S-NEW, S-$\Rrightarrow$-F, S-F-$\bot$ *and*

$$\dbox{\mathrm{F}}^{\gamma} \textit{ is persistent} \quad \textsc{f-persist}$$

We define the bad *proposition* $\mathrm{bad}^{\gamma} \in iProp$ *as follows:*

$$\mathrm{bad}^{\gamma} \triangleq \dbox{\mathrm{S}}^{\gamma} \lor \Box \Rrightarrow_{\bullet} \bot.$$

Finally, assume that an invariant connective $\boxed{\mathrm{bad}^{\gamma}} \in iProp$ *for the bad proposition satisfying the following rules:*

$$\boxed{\mathrm{bad}^{\gamma}} \textit{ is persistent} \quad \textsc{iinvbad-persist} \qquad \mathrm{bad}^{\gamma} \vDash \Rrightarrow_{\bullet} \boxed{\mathrm{bad}^{\gamma}} \quad \textsc{iinv-alloc}'\textsc{-bad}$$

[8] This amounts to fixing the invariant's namespace $\mathcal{N}$ to the universal namespace *InvName* and interpreting the full mask $\bullet$ as the universal namespace *InvName*.

$$\frac{\mathrm{bad}^{\gamma} * Q \vDash \Rrightarrow_{\varnothing} \left(\mathrm{bad}^{\gamma} * R\right)}{\boxed{\mathrm{bad}^{\gamma}} * Q \vDash \Rrightarrow_{\bullet} R} \quad \text{IINV-ACC-NOLATER-BAD}$$

Then we can prove the following:

$$\vDash \Rrightarrow_{\bullet} \bot.$$

Notably, the invariant connective that the paradox uses is limited to $\boxed{\mathrm{bad}^{\gamma}}$ for the 'bad' proposition $\mathrm{bad}^{\gamma} \triangleq \boxed{\mathrm{S}}^{\gamma} \vee \Box \Rrightarrow_{\bullet} \bot$, which just contains the fancy update $\Rrightarrow_{\bullet}$. Compared to the original paradox, we slightly strengthen the assumption about the token $\boxed{\mathrm{F}}^{\gamma}$, assuming that it is persistent F-PERSIST, not just duplicable F-DUP.

High-Level Idea of the New Paradox At the high level, the paradox is analogous to the following variant of *Landin's knot* Code 1.4 (§ 1.4):

```
let rbad : ref (option (unit -> bot)) = ref None in
rbad := Some (fun _ -> unwrap (!rbad) ());
unwrap (!rbad) ()
```

Code 3.1: Landin's knot with `ref (option (unit -> bot))`

This program Code 3.1 employs a *higher-order shared mutable reference* `rbad : ref (option (unit -> bot))`. Its body type is `option (unit -> bot)`, whose element is either `None` or `Some f` for a *closure* `f : unit -> bot`, whose execution is *provably non-terminating* due to the *empty type* `bot`. The function `unwrap` unwraps `Some a` into `a`, throwing an exception for `None`.

The program Code 3.1 goes as follows. First, the reference `rbad` is created with the initial value `None`. Then the value stored in `rbad` is updated into `Some (fun _ -> unwrap (!rbad) ())`. This is *well-typed*, because `unwrap (!rbad)` has the type `unit -> bot`. Finally, we call `unwrap (!rbad) ()`, which causes an *infinite loop*. This program is provably non-terminating (by checking that all `unwrap`s succeed), because the return type of this program is the empty type `bot`.

Let us connect the paradox to this variant of Landin's knot Code 3.1. Roughly speaking, the shared invariant connective $\boxed{-}$ corresponds to the shared mutable reference type `ref`, the fancy update modality $\Rrightarrow_{\bullet}$ corresponds to the closure type `->`, and the contradiction $\bot$ corresponds to the empty type `bot`. The first $\boxed{\mathrm{S}}^{\gamma}$ and second disjuncts $\Box \Rrightarrow_{\bullet} \bot$ of the bad proposition bad^{γ} respectively correspond to `None` and `Some f` for a divergent closure `f : unit -> bot`. At the high level, the contradiction $\Box \Rrightarrow_{\bullet} \bot$ is caused by an *infinite loop of logical reasoning*, caused by the *transitivity* of the fancy update $\Rrightarrow_{\bullet}$.

Proof of the New Paradox With this high-level idea in mind, we can prove this new paradox.

Proof of Theorem 3.9. First, combining S-NEW and IINV-ALLOC′-BAD, we can prove the following for creating the invariant $\boxed{\mathrm{bad}^{\gamma}}$ for the bad proposition bad^{γ}:

$$\vDash \Rrightarrow_{\bullet} \left(\exists \gamma.\ \boxed{\mathrm{bad}^{\gamma}}\right) \tag{3.24}$$

Assuming the invariant $\boxed{\mathrm{bad}^{\gamma}}$, a witness of the finished state $\boxed{\mathrm{F}}^{\gamma}$ causes a contradiction under the fancy update:

$$\boxed{\mathrm{bad}^{\gamma}} * \boxed{\mathrm{F}}^{\gamma} \vDash \Rrightarrow_{\bullet} \bot \tag{3.25}$$

To prove (3.25), we first modify the right-hand side into $\Rrightarrow_\bullet \Rrightarrow_\bullet \bot$ by the *transitivity* of the fancy update $\Rrightarrow_\bullet$. Then we get access to the content bad^γ with the *later-free* invariant access rule IINV-ACC-NOLATER-BAD, turning the goal into the following:

$$\mathrm{bad}^\gamma * [\mathrm{F}]^\gamma \models \Rrightarrow_\varnothing \left(\mathrm{bad}^\gamma * \Rrightarrow_\bullet \bot\right).$$

We branch on the disjunction $\mathrm{bad}^\gamma \triangleq [\mathrm{S}]^\gamma \vee \square \Rrightarrow_\bullet \bot$. The first disjunct $[\mathrm{S}]^\gamma$ is immediately rejected by S-F-$\bot$, thanks to the finished-state witness $[\mathrm{F}]^\gamma$. For the second disjunct $\square \Rrightarrow_\bullet \bot$, we introduce the empty-mask fancy update $\Rrightarrow_\varnothing$ and prove the following:

$$\square \Rrightarrow_\bullet \bot \models \mathrm{bad}^\gamma * \Rrightarrow_\bullet \bot \tag{3.26}$$

Note that the premise $\square \Rrightarrow_\bullet \bot$ is *persistent*. The first conjunct bad^γ is constructed by choosing its second disjunct. The second conjunct $\Rrightarrow_\bullet \bot$ follows by eliminating the persistence modality $\square$ ($\square$-ELIM, § 2.1.1).

By the persistence of $\boxed{\mathrm{bad}^\gamma}$ and $[\mathrm{F}]^\gamma$, we can introduce the persistence modality $\square$ to the conclusion of (3.25) (PERSIST-$\square$-INTRO, § 2.1.1):

$$\boxed{\mathrm{bad}^\gamma} * [\mathrm{F}]^\gamma \models \square \Rrightarrow_\bullet \bot \tag{3.27}$$

Then we can prove a contradiction only assuming the invariant $\boxed{\mathrm{bad}^\gamma}$:

$$\boxed{\mathrm{bad}^\gamma} \models \Rrightarrow_\bullet \bot \tag{3.28}$$

To prove (3.28), again we first modify the right-hand side into $\Rrightarrow_\bullet \Rrightarrow_\bullet \bot$ by the fancy update's *transitivity*. We also modify the left-hand side into $\boxed{\mathrm{bad}^\gamma} * \boxed{\mathrm{bad}^\gamma}$ by the persistence and get access to the content bad^γ with the invariant access rule IINV-ACC-NOLATER-BAD, turning the goal into the following:

$$\mathrm{bad}^\gamma * \boxed{\mathrm{bad}^\gamma} \models \Rrightarrow_\varnothing \left(\mathrm{bad}^\gamma * \Rrightarrow_\bullet \bot\right).$$

Again we branch on the disjunction $\mathrm{bad}^\gamma \triangleq [\mathrm{S}]^\gamma \vee \square \Rrightarrow_\bullet \bot$. For the second disjunct $\square \Rrightarrow_\bullet \bot$, the proof is finished by (3.26). For the first disjunct $[\mathrm{S}]^\gamma$, we first turn this start-state token into the finished-state witness $[\mathrm{F}]^\gamma$ by S-$\Rrightarrow$-F with an empty-mask fancy update $\Rrightarrow_\varnothing$. Then out of bad^γ and $[\mathrm{F}]^\gamma$, we get $\square \Rrightarrow_\bullet \bot$ by (3.27), which finishes the proof.

Finally, we can prove the desired contradiction

$$\models \Rrightarrow_\bullet \bot$$

simply by combining (3.24) and (3.28). $\square$

Note that the *transitivity* of the fancy update $\Rrightarrow_\bullet$ is the key to this paradox. Iris's later-requiring invariant mechanism avoids this paradox because the fancy update under the later modality $\triangleright \Rrightarrow_\bullet$ is *not transitive*.

3.4.2 Expressivity

The fact that the data type for syntactic propositions *nProp* and its semantic interpretation $[\![\ ]\!]$: *nProp* $\to$ *iProp* are *'well-defined'* imposes a *limitation* on what can be expressed in *nProp*. At the same time, that limitation naturally avoids the paradoxes of later-free invariants like Theorem 3.9 we discussed in § 3.4.1. Our general observation is as follows: the soundness of Nola-style sharing is ensured by the absence of *circularity* in the definitions of *nProp* and $[\![\ ]\!]$.

General design of *nProp* From the perspective of Nola's invariant, the naive later-free invariant of THOARE-IINV-NAIVE (§ 1.4) amounts to setting *nProp* $:=$ *iProp* and $[\![\ ]\!] := \lambda P.\, P$. But unfortunately, setting *nProp* $:=$ *iProp* indeed causes *circularity* because *iProp* directly depends on *nProp*.

On the other hand, we can set *nProp* anything as long as no circularity occurs. Sometimes *nProp* can contain quite *semantic* components. We already saw that any pure proposition $\phi \in Prop$ can be added to *nProp* (3.16). We can go further. For example, suppose we have a certain basic state $State^*$, consisting of the heap memory state $Loc \xrightarrow{\text{fin}} Val$ etc. Then we can add to *nProp* any basic SL propositions $P^* \in State^* \to Prop$, i.e., predicates over the basic state $State^*$. If the actual global state *State* for *iProp* is, say, the product $State^* \times \textsc{Inv}\ nProp$ of the basic state and the state for Nola's invariant, we can naturally interpret basic SL propositions by $[\![P^*]\!] \triangleq \lambda(s, \_).\ P^*\, s$.

How the Paradox Is Avoided First, let us discuss how the paradox Theorem 3.9 we showed in § 3.4.2 is avoided by the Nola framework. In order to attain the counterpart of the bad proposition invariant $\boxed{\mathsf{bad}^\gamma}$ in Nola, the data type *nProp* should contain a *bad proposition* bad^γ to *nProp*, which is interpreted $[\![\mathsf{bad}^\gamma]\!] \in iProp$ as follows:

$$[\![\mathsf{bad}^\gamma]\!] \quad =_? \quad \boxed{\mathsf{S}}^{\gamma} \lor \Box \mathrel{|\!\Rrightarrow}_{\bullet}^{\mathsf{Wiinv}\,[\![\ ]\!]} \bot. \tag{3.29}$$

We can define $\bullet$ as the universal set *InvName*. By creating an invariant $\mathsf{inv}^{\mathcal{N}}\, \mathsf{bad}^\gamma \in iProp$, we could prove the contradiction $\vDash\ \mathrel{|\!\Rrightarrow}_{\bullet}^{\mathsf{Wiinv}\,[\![\ ]\!]} \bot$.

But fortunately, it is *impossible* to *define* the semantic interpretation $[\![\ ]\!]\colon nProp \to iProp$ so that $[\![\mathsf{bad}^\gamma]\!]$ is interpreted as above (3.29). The right-hand side of (3.29) contains the *extended fancy update* $\mathrel{|\!\Rrightarrow}_{\bullet}^{\mathsf{Wiinv}\,[\![\ ]\!]}$, or more specifically Nola's world satisfaction $\mathsf{Wiinv}\,[\![\ ]\!]$. This depends on the interpretation $[\![\ ]\!]\colon nProp \to iProp$ over the *whole set* of propositions *nProp*, which includes even the left-hand side itself $[\![\mathsf{bad}^\gamma]\!]$. Because this introduces a malignant cyclic definition, the equation (3.29) makes the interpretation $[\![\ ]\!]$ *ill-defined*.

Fancy Update More specifically, the syntactic data type *nProp* of propositions cannot express the *extended fancy update modality* ${}_{\mathcal{E}}{\mathrel{|\!\Rrightarrow}}_{\mathcal{E}'}^{\mathsf{Wiinv}\,[\![\ ]\!]}$ employed for Nola's invariants. This is because the modality directly contains Nola's *world satisfaction* $\mathsf{Wiinv}\,[\![\ ]\!]$, which depends on the global interpretation $[\![\ ]\!]$.[9] If the data type *nProp* should have a fancy update constructor ${}_{\mathcal{E}}\mathsf{fupd}_{\mathcal{E}'}\, P$ interpreted as

$$[\![{}_{\mathcal{E}}\mathsf{fupd}_{\mathcal{E}'}\, P]\!] \quad \triangleq_? \quad {}_{\mathcal{E}}{\mathrel{|\!\Rrightarrow}}_{\mathcal{E}'}^{\mathsf{Winv}\,[\![\ ]\!]}\, [\![P]\!], \tag{3.30}$$

then the interpretation $[\![\ ]\!]$ would become *ill-defined*, because the right-hand side depends on $[\![\ ]\!]$, which includes the left-hand side $[\![{}_{\mathcal{E}}\mathsf{fupd}_{\mathcal{E}'}\, P]\!]$.

Hoare Triples For the same reason, *nProp* cannot express the new *Hoare triples* $\{P\}\, e\, \{\Psi\}_{\mathcal{E}}^{\mathsf{Winv}\,[\![\ ]\!]}$, $[P]\, e\, [\Psi]_{\mathcal{E}}^{\mathsf{Winv}\,[\![\ ]\!]}$ with Nola's world satisfaction $\mathsf{Wiinv}\,[\![\ ]\!]$, which internally use the extended fancy update ${}_{\mathcal{E}}{\mathrel{|\!\Rrightarrow}}_{\mathcal{E}'}^{\mathsf{Wiinv}\,[\![\ ]\!]}$. The constructors $\mathsf{phoare}_{\mathcal{E}}\ P\ e\ \Psi$ and $\mathsf{thoare}_{\mathcal{E}}\ P\ e\ \Psi \in nProp$ intepreted as follows would make the interpretation $[\![\ ]\!]$ *ill-defined*:

$$[\![\mathsf{phoare}_{\mathcal{E}}\ P\ e\ \Psi]\!] \quad \triangleq_? \quad \{[\![P]\!]\}\, e\, \{\lambda v.\, [\![\Psi\, v]\!]\}_{\mathcal{E}}^{\mathsf{Winv}\,[\![\ ]\!]}$$

[9] Note that *nProp* can express the *basic update* modality $\dot{\Rrightarrow}$ or the *original fancy update* ${}_{\mathcal{E}}{\mathrel{|\!\Rrightarrow}}_{\mathcal{E}'}$ for Iris's invariants without any difficulty.

$$[\![\texttt{thoare}_{\mathcal{E}}\ P\ e\ \Psi]\!] \quad \triangleq_? \quad \big[[\![P]\!]\big]\ e\ \big[\lambda v.\,[\![\Psi\ v]\!]\big]_{\mathcal{E}}^{\mathsf{Winv}\,[\![\ ]\!]}.$$

Notably, the paradox of Landin's knot (1.17) (§ 1.4) is avoided because the total Hoare triple cannot be directly expressed in *nProp*.

Impredicative Quantifiers Also, the syntactic data type *nProp* of propositions cannot generally express *impredicative quantifiers* such as $\exists\, X \in nProp.\ P_X$. For example, suppose that *nProp* includes a formula $\exists\, X.\ X \texttt{ -> } \bot$ interpreted as follows:

$$[\![\exists\, X.\ X \texttt{ -> } \bot]\!] \quad \triangleq_? \quad \exists X \in nProp.\ [\![X]\!] \to \bot. \tag{3.31}$$

Unfortunately, this breaks the well-definedness of $[\![\ ]\!]$, because the right-hand side contains the interpretation $[\![X]\!]$ of any syntactic propositions $X \in nProp$ whatsoever, which includes the left-hand side $[\![\exists\, X. \neg X]\!]$.

We believe that this limitation is fundamental for avoiding paradoxes. A later-free version of *named propositions*, another mechanism for storing a proposition (Dodds et al., 2016), causes a *paradox* leading to $\models\ \dot{\Rrightarrow}\bot$, if the named proposition connective allows an *impredicative quantifier* to be stored (Jung et al., 2018b, § 3.3). Also, Nola's world satisfaction $\mathsf{Winv}\,[\![\ ]\!]$ is constructed by an *impredicative quantifier* over $I : InvName \overset{\text{fin}}{\rightharpoonup} nProp$ (see (3.7), § 3.2.3 for details):

$$\mathsf{Winv}\,[\![\ ]\!] \quad \triangleq \quad \exists I.\ \cdots * \mathop{\huge *}_{\iota \in \operatorname{dom} I} \Big(\big([\![I\ \iota]\!] * \cdots \big) \lor \cdots \Big)$$

Generally allowing impredicative quantifiers in *nProp* would allow constructing the world satisfaction $\mathsf{Winv}\,[\![\ ]\!]$ in *nProp*, which would lead to our paradox Theorem 3.9 (§ 3.4.1).

Still, an impredicative quantifier is problematic only when the body of the quantifier refers to the *semantics* of the proposition bound by the quantifier. If the body refers only to the syntactic data of the bounded proposition, then the quantifier can be safely added to *nProp*. For example, it is safe to have a constructor $\exists\, X.\ \texttt{inv}^{\mathcal{N}}\, X \in nProp$ interpreted as follows:

$$[\![\exists\, X.\ \texttt{inv}^{\mathcal{N}}\, X]\!] \quad \triangleq \quad \exists X \in nProp.\ \mathsf{inv}^{\mathcal{N}}\, X.$$

This is because the invariant connective $\mathsf{inv}^{\mathcal{N}}\, X$ refers only to the *syntactic data* of the shared proposition X. In other words, the invariant connective $\mathsf{inv}^{\mathcal{N}}$ *guards* the occurrence of X. Recall the infinite list assertion $\texttt{ilist}^{\mathcal{N}}\ \Phi\ \ell$ in § 3.3.3, where the guard by $\mathsf{inv}^{\mathcal{N}}$ made the equation for the interpretation $[\![\texttt{ilist}^{\mathcal{N}}]\!]$ (3.18) not recursive with respect to $[\![\ ]\!]$.

Later Modality Recovers All Actually, we can recover the fancy update modality and impredicative quantifiers in general under the guard of the *later modality* $\triangleright$.

For example, we can safely add to *nProp* a logical connective $\texttt{|>}\ {}_{\mathcal{E}}\texttt{fupd}_{\mathcal{E}'}\ P$, the composite of the later modality $\texttt{|>}$ and the fancy update modality ${}_{\mathcal{E}}\texttt{fupd}_{\mathcal{E}'}\ P$, interpreted as follows:

$$[\![\texttt{|>}\ {}_{\mathcal{E}}\texttt{fupd}_{\mathcal{E}'}\ P]\!] \quad \triangleq \quad \triangleright\ {}_{\mathcal{E}}\!\Rrightarrow_{\mathcal{E}'}^{\mathsf{Winv}\,[\![\ ]\!]}\ [\![P]\!] \tag{3.32}$$

The equation (3.32) is a valid definition, unlike the interpretation of the fancy update $[\![{}_{\mathcal{E}}\texttt{fupd}_{\mathcal{E}'}\ P]\!]$ (3.30). This is because the occurrence of the global interpretation $[\![\ ]\!]$ in (3.32) is *guarded* by the later modality $\triangleright$, which is a *contractive* mapping (recall Theorem 2.1, § 2.1.2).

Similarly, we can also safely add the constructor $\texttt{|>}\ \texttt{phoare}_{\mathcal{E}}\ P\ e\ \Phi$ for the partial Hoare triple under the later modality, interpreted as follows:

$$[\![\texttt{|>}\ \texttt{phoare}_{\mathcal{E}}\ P\ e\ \Psi]\!] \quad \triangleq \quad \triangleright \big\{[\![P]\!]\big\}\ e\ \big\{\lambda v.\,[\![\Psi\ v]\!]\big\}_{\mathcal{E}}^{\mathsf{Winv}\,[\![\ ]\!]}$$

This is indeed useful, because the later modality on the partial Hoare triple can be stripped off during execution (recall STEP-PHOARE, § 1.4).

We can safely add the constructor $\rhd\, \texttt{thoare}_{\mathcal{E}}\ P\ e\ \Phi$ for the total Hoare triple under the later modality as well, but it is less useful because we have no chance to strip off the later modality in verifying total Hoare triples (recall discussions in § 1.4).

We can also support general impredicative quantifiers under the later modality. For example, we can safely add the constructor $\rhd\, \exists\, X.\ X \texttt{ -> } \bot$, interpreted as follows:

$$[\![\rhd\, \exists\, X.\ X \texttt{ -> } \bot]\!] \quad \triangleq \quad \triangleright\, \exists X \in \mathit{nProp}.\ [\![X]\!] \to \bot.$$

This is valid unlike (3.31), thanks to the guard of the later modality $\triangleright$ on the reference to the interpretation $[\![X]\!]$.

Nola's invariant mechanism arguably *subsumes* Iris's invariant mechanism in expressivity, because the data type *nProp* can effectively express any kind of Iris propositions $\triangleright P$ under the later modality $\triangleright$. While Iris's invariant mechanism *unconditionally* puts the later modality $\triangleright$ to all propositions, Nola's invariant mechanism calls for the later modality *only on 'problematic' propositions*, such as the fancy update modality and impredicative quantifiers, which can lead to *paradoxes* without the later modality.

Another Workaround: Stratification We can also share propositions like the total Hoare triple by *stratifying* the separation logic propositions. Such stratification has long been known (Ahmed et al., 2002).

For example, we can consider two representations of separation logic propositions, nProp_0 and nProp_1, and let nProp_1 contain the total Hoare triple $\texttt{thoare}^0\ P\ e\ \Psi$ that supports invariants on nProp_0 (but not nProp_1). We can enjoy invariants on both nProp_0 and nProp_1 by adding both $\textsc{Inv}\,\mathit{nProp}_0$ and $\textsc{Inv}\,\mathit{nProp}_1$ to the global camera $\tilde{\mathcal{G}}$. The interpretations can be constructed without circularity by constructing $[\![\ ]\!]_1 \colon \mathit{nProp}_1 \longrightarrow \mathit{iProp}$ after constructing $[\![\ ]\!]_0 \colon \mathit{nProp}_0 \longrightarrow \mathit{iProp}$:

$$[\![\texttt{thoare}^0\ P\ e\ \Psi]\!]_1 \quad \triangleq \quad \big[[\![P]\!]_1\big]\ e\ \big[[\![\Psi]\!]_1\big]^{\mathsf{Winv}\,[\![\ ]\!]_0}$$

This avoids the paradox of Landin's knot (1.17) (§ 1.4), because the total Hoare triple $\texttt{thoare}^0\ P\ e\ \Psi$ in the higher-order invariant over nProp_1 can access only invariants over nProp_0 and not the higher-order invariant itself.

We can naturally consider more levels of stratification to achieve more expressivity. Nola's parameterization over *nProp* and $[\![\ ]\!]$ allows for such stratification in general. We revisit this technique later in § 5.2 for semantically modeling and verifying a stratified type system § 5.1.

Chapter 4

Semantic Alteration by Derivability

柔能制剛
The soft can overcome the hard

The *Three Strategies*

Chapter 4 presents a novel, general technique for *semantic alteration* of the content propositions of the logical connectives for propositional sharing (§ 4.1). The key idea is to *parameterize* the semantics of propositions $[\![\ ]\!] \colon nProp \to iProp$ with an undecided *derivability* predicate δ to be used for semantic alteration (§ 4.2). Unlike a traditional approach where such a derivability predicate δ should be syntactically constructed up-front, we have found a general construction of the well-behaved derivability predicate that one can obtain *for free* once one builds the parameterized *semantics* of judgments (§ 4.3). Going further, we also present an advanced model of the invariant that allows even the *merger* of invariants (§ 4.4).

This chapter focuses on the invariant mechanism presented in § 3.2.2. The application of the derivability technique to the borrow machinery is later discussed in § 6.4.

4.1 Goal: Semantic Alteration

The direct model of the invariant connective $\mathtt{inv}^{\mathcal{N}}\,P$ considered so far ((3.17), § 3.3.3)

$$[\![\mathtt{inv}^{\mathcal{N}}\,P]\!] \quad \triangleq \quad \mathtt{inv}^{\mathcal{N}}\,P \tag{4.1}$$

is not satisfactory because the inner formula P cannot be modified in a *semantic* way.

Examples For a simple example, we expect the following equality to hold

$$[\![\mathtt{inv}^{\mathcal{N}}(P * Q)]\!] \;=\; [\![\mathtt{inv}^{\mathcal{N}}(Q * P)]\!] \tag{4.2}$$

because the separating conjunction is commutative. But this does not hold for the naive model (4.1) because the ghost state of Nola's invariant uses 'syntactic' agreement over *nProp*, agnostic of its semantics $[\![\ ]\!]$.

We also expect the following entailment to hold

$$[\![\mathtt{inv}^{\mathcal{N}}(P * Q)]\!] \vDash [\![\mathtt{inv}^{\mathcal{N}}\,P]\!] \tag{4.3}$$

because P is a 'part' of the situation described by $P * Q$.

We also want semantic alteration to work for nested connectives. For example, we expect the following equality to hold

$$[\![\mathtt{inv}^{\mathcal{N}'}\,\mathtt{inv}^{\mathcal{N}}\,(P * Q)]\!] \;=\; [\![\mathtt{inv}^{\mathcal{N}'}\,\mathtt{inv}^{\mathcal{N}}\,(Q * P)]\!] \tag{4.4}$$

by applying (4.2) to the inner invariant.

Naive Model How can we support such *semantic alteration* of the content of an invariant? Naively, we want to define the interpretation of the invariant $\mathtt{inv}^{\mathcal{N}}\, P$ as follows:

$$[\![\mathtt{inv}^{\mathcal{N}}\, P]\!] \quad \triangleq_? \quad \exists\, Q \text{ s.t. } \square \big([\![Q]\!] \twoheadrightarrow \big([\![P]\!] * ([\![P]\!] \twoheadrightarrow [\![Q]\!]) \big) \big).\ \ \mathtt{inv}^{\mathcal{N}}\, Q \tag{4.5}$$

We existentially quantify the actual syntactic proposition $Q \in nProp$ used for the inner syntactic agreement of the invariant. Also, we assert the relationship between the exposed proposition $P \in nProp$ and the inner proposition Q. To express the fact that P is a 'part' of Q (recall (4.3)), we use a *semantic conversion*

$$[\![Q]\!] \twoheadrightarrow \big([\![P]\!] * ([\![P]\!] \twoheadrightarrow [\![Q]\!]) \big), \tag{4.6}$$

which splits $[\![Q]\!]$ into $[\![P]\!]$ and the rest that turns back into $[\![Q]\!]$ when combined with $[\![P]\!]$. We put the semantic conversion under the persitence modality $\square$ because we want the proposition $[\![\mathtt{inv}^{\mathcal{N}}\, P]\!]$ to be persistent:

$$[\![\mathtt{inv}^{\mathcal{N}}\, P]\!] \ \text{ is persistent}$$

Note that we consider the separating implication (4.6) under the persistence modality $\square$ instead of the entailment

$$[\![Q]\!] \vDash [\![P]\!] * ([\![P]\!] \twoheadrightarrow [\![Q]\!])$$

to allow the semantic conversion to *dependent on resources.*[1]

If the naive model of (4.5) should work, it would support the expected properties such as (4.2), (4.3) and (4.4).

But unfortunately, *this model does not work* as a definition of the interpretation function $[\![\ ]\!]$. This is because it has an ill-formed recursive call $[\![Q]\!]$, where Q is not necessarily structurally smaller than $\mathtt{inv}^{\mathcal{N}}\, P$. We want to find a good substitute for the *semantic conversion* (4.6) that can be used to define the interpretation of the invariant $[\![\mathtt{inv}^{\mathcal{N}}\, P]\!]$ *without causing an invalid recursive call.*

4.2 First Step: Parameterization by Derivability

Parameterization by Derivability As the first step, we *parameterize* our proposition interpretation $[\![\ ]\!]\colon nProp \to iProp$ by a *derivability predicate* $\delta\colon Judg \to iProp$ that expresses the semantic conversion, where the *syntactic* data type for *judgments Judg* here has the following form:

$$Judg \ni J \ ::= \ P \propto Q \tag{4.7}$$

A judgment $P \propto Q$ means that the proposition $Q \in nProp$ is semantically a part of the proposition $P \in nProp$.

The new proposition semantics $[\![\ ]\!]_\delta\colon nProp \to iProp$ parameterized with the derivability predicate δ is defined like the following:

$$[\![\ell \mathrel{\mathtt{|\!\!->}} v]\!]_\delta \ \triangleq\ \ell \mapsto v \qquad\qquad [\![P \star Q]\!]_\delta \ \triangleq\ [\![P]\!]_\delta * [\![Q]\!]_\delta.$$

[1] For an interesting example of the power of resource-dependent semantic conversion, we can consider the following semantic alteration on the borrower connective $\mathtt{bor}$ later introduced in § 6.3.2: $\alpha \sqsubseteq \beta * \beta \sqsubseteq \alpha * [\![\mathtt{inv}^{\mathcal{N}}(\mathtt{bor}^{\alpha}\, P)]\!] \vDash [\![\mathtt{inv}^{\mathcal{N}}(\mathtt{bor}^{\beta}\, P)]\!]$. This modifies the lifetime of the borrower from α to β under the mutual lifetime inclusion $\sqsubseteq$ between α and β.

Also, resource-dependent semantic conversion is the key to the advanced model of the invariant connective (4.21) we present later in § 4.4.

$$[\![\mathtt{inv}^{\mathcal{N}}\,P]\!]_\delta \quad \triangleq \quad \exists\, Q \text{ s.t.} \,\Box\, \delta\,(Q \propto P).\ \ \mathtt{inv}^{\mathcal{N}}\,Q \tag{4.8}$$

A basic token like the points-to token $\ell \mapsto v$ simply ignores δ. A basic connective like the separating conjunction $*$ is defined by inheriting δ to its subformulas. The derivability predicate δ is used only by the interpretation of the invariant connective $[\![\mathtt{inv}^{\mathcal{N}}\,P]\!]_\delta$. Unlike (4.5), this new model of the invariant (4.8) uses $\delta\,(Q \propto P)$ for the semantic conversion, which does *not* cause an invalid recursive call of $[\![\ ]\!]$. As a result, this gives a totally *valid* definition of the proposition interpretation $[\![\ ]\!]$.

After defining the parameterized proposition interpretation $[\![\ ]\!]_\delta \colon \mathit{nProp} \to \mathit{iProp}$, we can define the *judgment interpretation* $[\![\ ]\!]^{+}_{\delta} \colon \mathit{Judg} \to \mathit{iProp}$, again parameterized over the derivability predicate $\delta \colon \mathit{nProp} \to \mathit{iProp}$, as follows:[2]

$$[\![P \propto Q]\!]^{+}_{\delta} \quad \triangleq \quad [\![P]\!]_\delta \twoheadrightarrow \big(\,[\![Q]\!]_\delta * ([\![Q]\!]_\delta \twoheadrightarrow [\![P]\!]_\delta)\,\big). \tag{4.9}$$

Sound Derivability We say a derivability predicate δ is *sound* if it satisfies, for any $P, Q \in \mathit{nProp}$,

$$\delta\,(Q \propto P) \ \vDash\ [\![Q \propto P]\!]^{+}_{\delta}.$$

Suppose we have a derivability predicate $\delta \colon \mathit{nProp} \to \mathit{iProp}$ that is *sound*. Then the enriched invariant $[\![\mathtt{inv}^{\mathcal{N}}\,P]\!]_\delta$ satisfies

$$[\![\mathtt{inv}^{\mathcal{N}}\,P]\!]_\delta \ \vDash\ \exists\, Q \text{ s.t.} \,\Box\,\big(\,[\![Q]\!]_\delta \twoheadrightarrow \big(\,[\![P]\!]_\delta * ([\![P]\!]_\delta \twoheadrightarrow [\![Q]\!]_\delta)\,\big)\big).\ \ \mathtt{inv}^{\mathcal{N}}\,Q.$$

Combining this and INV-ACC, we can derive the following proof rule for accessing an enriched invariant, setting $[\![\ ]\!] \triangleq [\![\ ]\!]_{\mathsf{der}}$:

$$\frac{[\![P]\!] * Q \ \vDash {\Rrightarrow}^{\mathsf{Winv}\,[\![\ ]\!]}_{\mathcal{E}}\ (\,[\![P]\!] * R\,)}{[\![\mathtt{inv}^{\mathcal{N}}\,P]\!] * Q \ \vDash {\Rrightarrow}^{\mathsf{Winv}\,[\![\ ]\!]}_{\mathcal{N} \uplus \mathcal{E}}\ R}\ \ \mathtt{inv}\text{-ACC}$$

Or more generally, we have the following rule like INV-ACC-CH:

$$[\![\mathtt{inv}^{\mathcal{N}}\,P]\!] \ \vDash\ {}_{\mathcal{N}}{\Rrightarrow}^{\mathsf{Winv}\,[\![\ ]\!]}_{\varnothing}\ \Big(\,[\![P]\!] * \big(\,[\![P]\!] \twoheadrightarrow {}_{\varnothing}{\Rrightarrow}^{\mathsf{Winv}\,[\![\ ]\!]}_{\mathcal{N}}\ \top\,\big)\Big) \quad \mathtt{inv}\text{-ACC-CH}$$

Generalization We generalize this situation. We build a data type for *judgments Judg*. A *derivability predicate* δ is any Iris predicate $\delta \colon \mathit{Judg} \to \mathit{iProp}$. We can build the parameterized *judgment interpretation* $[\![\ ]\!]^{+}_{-} \colon (\mathit{Judg} \to \mathit{iProp}) \to (\mathit{Judg} \to \mathit{iProp})$, the interpretation of judgments *Judg* parameterized over the derivability predicate $\mathit{Judg} \to \mathit{iProp}$. We say a derivability predicate δ is *sound* if it satisfies, for any judgment $J \in \mathit{Judg}$,

$$\delta\, J \ \vDash\ [\![J]\!]^{+}_{\delta}.$$

Challenge: Build a Useful Sound Derivability The next challenge is to build a *sound* derivability predicate $\mathsf{der} \colon \mathit{nProp} \to \mathit{iProp}$ that is *'useful'*.

The idealistic situation would be that der is exactly the *fixed point* of the parameterized judgment interpretation $[\![\ ]\!]^{+}_{-} \colon (\mathit{Judg} \to \mathit{iProp}) \to (\mathit{Judg} \to \mathit{iProp})$:

$$\mathsf{der}\, J \quad \triangleq \quad [\![J]\!]^{+}_{\mathsf{der}}. \tag{4.10}$$

Unfortunately, this is not possible in general because the occurrence of der in $[\![\ ]\!]^{+}_{\mathsf{der}}$ is neither monotone nor guarded. If der were to satisfy (4.10), the proposition interpretation $[\![\ ]\!]_{\mathsf{der}}$ would satisfy the desirable equation (4.5). So we want a derivability predicate der that, roughly speaking, *under-approximates* the expected semantics $\lambda\, J.\ [\![J]\!]^{+}_{\delta}$ *as tightly as possible.*

[2] For a more radical approach, we can also set *Judg* to *nProp* itself, setting $[\![\ ]\!]^{+}_{\delta} \triangleq [\![\ ]\!]_\delta$, and use $\delta\,(P \twoheadrightarrow (Q * (Q \twoheadrightarrow P)))$ instead of $\delta\,(P \propto Q)$ in (4.8).

In a traditional approach, a 'useful' derivability predicate der should be *syntactically* constructed, considering all the wanted proof rules *upfront*. But that is not quite acceptable. We want to get a useful derivability predicate der *for free* once we define the judgment semantics $[\![\ ]\!]^+_- : (\mathit{Judg} \to \mathit{iProp}) \to (\mathit{Judg} \to \mathit{iProp})$. How can we attain this goal?

First Attempt: Universal Quantification As the first attempt, we can define the derivability predicate der as follows:

$$\mathsf{der}\, J \quad \triangleq \quad \forall \delta.\ [\![J]\!]^+_\delta. \tag{4.11}$$

Because we do not know δ in advance when we define der, we *universally quantify* a derivability predicate δ for the semantics $[\![J]\!]^+_\delta$. By definition, this derivability predicate der is always sound.

The der defined as (4.11) already behaves quite nicely. It supports semantic alteration like (4.2) and (4.3) for $[\![\ ]\!]_{\mathsf{der}}$, namely:

$$[\![\mathtt{inv}^{\mathcal{N}}(P * Q)]\!]_{\mathsf{der}} \ = \ [\![\mathtt{inv}^{\mathcal{N}}(Q * P)]\!]_{\mathsf{der}} \qquad [\![\mathtt{inv}^{\mathcal{N}}(P * Q)]\!]_{\mathsf{der}} \vDash [\![\mathtt{inv}^{\mathcal{N}}\, P]\!]_{\mathsf{der}}.$$

This is because der thus defined satisfies the *transitivity*

$$\mathsf{der}\,(P \propto Q) \ * \ \mathsf{der}\,(Q \propto R) \ \vDash \ \mathsf{der}\,(P \propto R).$$

and because semantic judgments like the following hold for any δ:

$$\vDash [\![(P * Q) \propto (Q * P)]\!]^+_\delta \qquad \vDash [\![(P * Q) \propto P]\!]^+_\delta \tag{4.12}$$

The remaining problem is that we cannot use this definition of der to prove semantic alteration on *nested* connective like (4.4). This is because alteration on invariants like $[\![\mathtt{inv}^{\mathcal{N}}(P * Q)]\!]_\delta = [\![\mathtt{inv}^{\mathcal{N}}(Q * P)]\!]_\delta$ does not hold for *any* derivability predicates δ: $\mathit{Judg} \to \mathit{iProp}$ whatsoever. We should update the definition of der (4.11) so that we appropriately *restrict* the range of the universally quantified derivability predicate δ.

4.3 Our Key Achievement: General Derivability Construction

General Derivability Construction Remarkably, we have found a novel *general* way to construct a well-behaved derivability predicate der, regardless of the choice of the judgment data type *Judg* and its interpretation $[\![\ ]\!]^+_-$. Our construction is as follows.

Definition 4.1 (General Derivability Construction)**.** Given the judgment data type *Judg* and its parameterized interpretation $[\![\ ]\!]^+_- : (\mathit{Judg} \to \mathit{iProp}) \to (\mathit{Judg} \to \mathit{iProp})$, we define the *evolution* relation between derivability predicates $\rightsquigarrow : (\mathit{Judg} \to \mathit{iProp}) \times (\mathit{Judg} \to \mathit{iProp}) \to \mathit{iProp}$, the set of *good* derivability predicates $\mathit{Deriv} \subseteq (\mathit{Judg} \to \mathit{iProp})$, and the *best* derivability predicate $\mathsf{der} : \mathit{Judg} \to \mathit{iProp}$ as follows:

$$\delta \rightsquigarrow \delta' \quad \triangleq \quad \square \big(\forall J.\ \delta\, J \to ([\![J]\!]^+_{\delta'} \land \delta'\, J)\big) \tag{4.13}$$

$$\delta \in \mathit{Deriv} \quad \triangleq_\mu \quad \forall J.\ \big(\forall\, \delta' \in \mathit{Deriv} \text{ s.t. } \delta \rightsquigarrow \delta'.\ [\![J]\!]^+_{\delta'}\big) \vDash \delta\, J \tag{4.14}$$

$$\mathsf{der}\, J \quad \triangleq_\mu \quad \forall\, \delta \in \mathit{Deriv} \text{ s.t. } \mathsf{der} \rightsquigarrow \delta.\ [\![J]\!]^+_\delta \tag{4.15}$$

The domain of universal quantification in der (4.15) is *restricted* to the set of *good* derivability predicates *Deriv*, improving on the previous definition (4.11). Again, the idealistic situation would be that *Deriv* is the singleton set {der}, but generally we cannot do so before we define der. Instead, we define *Deriv* as a set of derivability predicates that behave pretty like der.

In the definition of der (4.15), the universally quantified derivability predicate δ is associated with der using an *evolution* relation der $\rightsquigarrow \delta$. The definition of *Deriv* (4.14) also follows this pattern. The evolution relation $\delta \rightsquigarrow \delta'$ (4.13) persistently asserts that the former δ can be transformed into the judgment semantics by the latter δ', $[\![J]\!]^+_{\delta'}$, and also into the latter δ' itself.

A key trick is to use the *least fixed point* $\triangleq_\mu$ for defining both *Deriv* (4.14) and der (4.15), carefully designing their definitions so that their self-reference is *monotone*, i.e., in a *positive* position. In the definition of der (4.15), it is crucial that the 'association' der $\rightsquigarrow \delta$ between the universal quantified derivability predicate δ and der is *antitone* in der. Also, the definition of *Deriv* (4.14) should use the entailment $\vDash$ instead of the equality $=$ for the monotonicity of the self-reference to *Deriv*.

General Properties The set of good derivability predicates *Deriv* is designed so that it contains the best derivability predicate der:

Lemma 4.2. der $\in$ *Deriv.*

Proof. Clear by definition. □

And importantly, we have designed the best derivability predicate der so that it is *sound*, extending the idea of (4.11). The proof is a bit tricky, using *strong induction.*

Theorem 4.3 (Soundness of der). *The best derivability predicate* der *is sound, that is,* der $J \vDash [\![J]\!]^+_{\mathsf{der}}$ *holds for any* $J \in$ *Judg.*

Proof. By *strong induction* on the *least fixed point structure* $\triangleq_\mu$ of der (4.15), the goal der $J \vDash [\![J]\!]^+_{\mathsf{der}}$ reduces to the following:

$$\Big(\forall\, \delta \in \mathit{Deriv}.\ \big((\lambda J.\ [\![J]\!]^+_{\mathsf{der}} \land \mathsf{der}\ J) \rightsquigarrow \delta\big) \mathrel{-\!\!*} [\![J]\!]^+_{\delta}\Big) \vDash [\![J]\!]^+_{\mathsf{der}}.$$

First, we instantiate δ in the right-hand side with der, using der $\in$ *Deriv* Lemma 4.2. Then it suffices to supply the premise $(\lambda J.\ [\![J]\!]^+_{\mathsf{der}} \land \mathsf{der}\ J)$ of the separating implication, which holds by definition. □

We have the following useful proof rules for proving a *derivability* $\delta\, J$ for a good derivability predicate $\delta \in$ *Deriv*.

Lemma 4.4 (Utilities for *Deriv*). *We have the following rules:*

$$\big(\forall\, \delta \in \mathit{Deriv}.\ [\![J]\!]^+_{\delta}\big) \vDash \forall\, \delta \in \mathit{Deriv}.\ \delta\, J \quad \textsc{deriv-intro}$$

$$\big(\forall\, \delta \in \mathit{Deriv}.\ [\![J']\!]^+_{\delta} \mathrel{-\!\!*} [\![J]\!]^+_{\delta}\big) \vDash \forall\, \delta \in \mathit{Deriv}.\ \delta\, J' \mathrel{-\!\!*} \delta\, J \quad \textsc{deriv-map}$$

$$\big(\forall\, \delta \in \mathit{Deriv}.\ [\![J']\!]^+_{\delta} * [\![J'']\!]^+_{\delta} \mathrel{-\!\!*} [\![J]\!]^+_{\delta}\big) \vDash \forall\, \delta \in \mathit{Deriv}.\ \delta\, J' * \delta\, J'' \mathrel{-\!\!*} \delta\, J \quad \textsc{deriv-map}_2$$

Proof. DERIV-INTRO holds by the definition of *Deriv*.

To prove DERIV-MAP, we first apply the universally quantified entailment of *Deriv* to $\delta\, J$ on the right-hand side, introducing $\delta' \in$ *Deriv*. We also instantiate the universal quantifier of the left-hand side with δ'. Now the goal is reduced to the following:

$$\big([\![J']\!]^+_{\delta'} \mathrel{-\!\!*} [\![J]\!]^+_{\delta'}\big) * (\delta \rightsquigarrow \delta') \vDash \delta\, J' \mathrel{-\!\!*} [\![J]\!]^+_{\delta'}.$$

To prove the right-hand side, we use the premise $\delta \rightsquigarrow \delta'$ to turn $\delta\, J$ into $[\![J]\!]^+_{\delta'}$ and then apply the first conjunct to $[\![J]\!]^+_{\delta'}$ to get the goal $[\![J']\!]^+_{\delta'}$.

The proof of DERIV-MAP$_2$ is similar to that of DERIV-MAP. □

Examples For the judgment data type *Judg* of (4.7) and the judgment semantics $[\![\ ]\!]^{+}_{-}$ of (4.9) we have specifically considered, we can derive the reflexivity (4.16) and transitivity (4.17) of any derivability predicate $\delta \in \mathit{Deriv}$, by applying DERIV-INTRO and DERIV-MAP$_2$ respectively:

$$\models\ \delta\,(P \propto P) \tag{4.16}$$

$$\delta\,(P \propto Q)\ *\ \delta\,(Q \propto R)\ \models\ \delta\,(P \propto R) \tag{4.17}$$

Thanks to the reflexivity (4.16), we can turn a direct invariant token $\mathsf{inv}^{\mathcal{N}}\,P$ into the enriched invariant $[\![\mathsf{inv}^{\mathcal{N}}\,P]\!]_\delta$ for any $\delta \in \mathit{Deriv}$:

$$\mathsf{inv}^{\mathcal{N}}\,P\ \models\ [\![\mathsf{inv}^{\mathcal{N}}\,P]\!]_\delta$$

Combining this with INV-ALLOC, we get the following rule for any $\delta \in \mathit{Deriv}$:

$$[\![P]\!]_\delta\ \models \mathrel{\Rrightarrow}^{\mathsf{Winv}\,[\![\ ]\!]_\delta}_{\varnothing}\ [\![\mathsf{inv}^{\mathcal{N}}\,P]\!]_\delta \quad \textsf{inv-ALLOC}$$

Thanks to the transitivity (4.17), we can alter the content proposition of the enriched invariant according to a derivability $\delta\,(P \propto Q)$ for any $\delta \in \mathit{Deriv}$:

$$\Box\ \delta\,(P \propto Q)\ *\ [\![\mathsf{inv}^{\mathcal{N}}\,P]\!]_\delta\ \models\ [\![\mathsf{inv}^{\mathcal{N}}\,Q]\!]_\delta \quad \textsf{inv-ALTER}$$

Applying DERIV-INTRO to the derivability $\delta\,(P \propto Q)$ in the premise of inv-ALTER, we get the following strategy for proving a semantic alteration of an invariant:

$$\Box\left(\forall \delta' \in \mathit{Deriv}.\ [\![J]\!]^{+}_{\delta'}\right)\ *\ [\![\mathsf{inv}^{\mathcal{N}}\,P]\!]_\delta\ \models\ [\![\mathsf{inv}^{\mathcal{N}}\,Q]\!]_\delta \tag{4.18}$$

Using (4.18), we can prove semantic alteration like (4.2) and (4.3) for the semantics $[\![\ ]\!]_\delta$ on any $\delta \in \mathit{Deriv}$, thanks to the semantic judgments (4.12):

$$[\![\mathsf{inv}^{\mathcal{N}}(P \star Q)]\!]_\delta\ =\ [\![\mathsf{inv}^{\mathcal{N}}(Q \star P)]\!]_\delta \tag{4.19}$$

$$[\![\mathsf{inv}^{\mathcal{N}}(P \star Q)]\!]_\delta\ \models\ [\![\mathsf{inv}^{\mathcal{N}}\,P]\!]_\delta$$

Now, unlike the simpler model of (4.11), we have also achieved the power to prove semantic alteration for *nested* invariants like (4.4) on any $\delta \in \mathit{Deriv}$:

$$[\![\mathsf{inv}^{\mathcal{N}'}\ \mathsf{inv}^{\mathcal{N}}\,(P \star Q)]\!]_\delta\ =\ [\![\mathsf{inv}^{\mathcal{N}'}\ \mathsf{inv}^{\mathcal{N}}\,(Q \star P)]\!]_\delta \tag{4.20}$$

To prove (4.20), we apply (4.18) and derive from (4.19) the following semantic judgment for *any* $\delta \in \mathit{Deriv}$:

$$\models\ [\![\mathsf{inv}^{\mathcal{N}}\,(P \star Q)\ \propto\ \mathsf{inv}^{\mathcal{N}}\,(Q \star P)]\!]^{+}_{\delta}$$

Here, the key is to keep the *universal quantification* over good derivability predicates $\delta \in \mathit{Deriv}$ in (4.19).

Inductive Semantic Alteration Our approach to semantic alteration works also for *recursive* propositions. Recall the infinite singly linked list proposition $\mathtt{ilist}^{\mathcal{N}}\ \Phi\ \ell$ introduced in § 3.3.3. First, we update the interpretation of this proposition as follows, modifying (3.18):

$$[\![\mathtt{ilist}^{\mathcal{N}}\ \Phi\ \ell]\!]_\delta\quad \triangleq\quad [\![\mathsf{inv}^{\mathcal{N}}(\Phi\ \ell)]\!]_\delta\ *\ [\![\mathsf{inv}^{\mathcal{N}}\big(\exists \ell'.\ (\ell+1) \mapsto \ell'\ \star\ \mathtt{ilist}^{\mathcal{N}}\ \Phi\ \ell'\big)]\!]_\delta.$$

We can semantically alter the content predicate of this proposition according to a derivability for any $\delta \in \mathit{Deriv}$, just like inv-ALTER:

$$\Box\big(\forall \ell.\ \delta\,(\Phi\,\ell \propto \Psi\,\ell)\ *\ \delta\,(\Psi\,\ell \propto \Phi\,\ell)\big)\ *\ [\![\mathtt{ilist}^{\mathcal{N}}\ \Phi\ \ell]\!]_\delta\ \models\ [\![\mathtt{ilist}^{\mathcal{N}}\ \Psi\ \ell]\!]_\delta$$

The proof goes by *induction* over the *least fixed point structure* $\triangleq_\mu$ of the set *Deriv* (4.14) (Definition 4.1).

To facilitate such induction, we can use the technique of *parameterized induction*, the dual of better-known *parameterized coinduction* (Winskel, 1989; Moss, 2001; Hur et al., 2013).

Definition 4.5 (Parameterized *Deriv*)**.** Given the judgment data type *Judg* and its parameterized interpretation $[\![\ ]\!]^{+}_{-} \colon (\mathit{Judg} \to \mathit{iProp}) \to (\mathit{Judg} \to \mathit{iProp})$, we define the set of *good* derivability predicates $\mathit{Deriv}_\phi \subseteq (\mathit{Judg} \to \mathit{iProp})$ *parameterized* over the *induction hypothesis* $\phi \colon (\mathit{Judg} \to \mathit{iProp}) \to \mathit{Prop}$ as follows:

$$\delta \in \mathit{Deriv}_\phi \quad \triangleq_\mu \quad \forall J.\ \big(\forall\, \delta' \in \mathit{Deriv}_\phi \text{ s.t. } \phi\, \delta' \text{ and } \delta \rightsquigarrow \delta'.\ [\![J]\!]^{+}_{\delta'}\big) \vDash \delta\, J.$$

Now Deriv_ϕ is parameterized over the induction hypothesis ϕ. The original *Deriv* corresponds to $\mathit{Deriv}_{\lambda\_.\top}$. The set Deriv_ϕ is monotone over the induction hypothesis parameter ϕ (DERIV*-MONO). Also, we can *accumulate* the induction goal into the induction hypothesis parameter (DERIV*-ACC).

Lemma 4.6 (Modify ϕ of Deriv_ϕ)**.** *For any pure predicates $\phi, \psi \colon (\mathit{Judg} \to \mathit{iProp}) \to \mathit{Prop}$, the following rules hold:*

$$\frac{\forall \delta.\ \phi\,\delta \to \psi\,\delta}{\mathit{Deriv}_\phi \subseteq \mathit{Deriv}_\psi}\ \text{DERIV}^*\text{-MONO} \qquad \frac{\forall\,\delta \in \mathit{Deriv}_{\phi\wedge\psi}.\ \phi\,\delta}{\forall\,\delta \in \mathit{Deriv}_\psi.\ \phi\,\delta}\ \text{DERIV}^*\text{-ACC}$$

Here we write $\phi \wedge \psi$ for the pointwise conjunction $\lambda\delta.\ \phi\,\delta \wedge \psi\,\delta$.

The utility proof rules of Lemma 4.4 are modified as follows.

Lemma 4.7 (Utilities for Deriv_ϕ)**.** *For any $\phi \colon (\mathit{Judg} \to \mathit{iProp}) \to \mathit{Prop}$, we have the following rules:*

$$\big(\forall\,\delta \in \mathit{Deriv}_\phi \text{ s.t. } \phi\,\delta.\ [\![J]\!]^{+}_{\delta}\big) \vDash \forall\,\delta \in \mathit{Deriv}_\phi.\ \delta\,J \quad \text{DERIV}^*\text{-INTRO}$$

$$\big(\forall\,\delta \in \mathit{Deriv}_\phi \text{ s.t. } \phi\,\delta.\ [\![J']\!]^{+}_{\delta} \mathrel{-\!\!*} [\![J]\!]^{+}_{\delta}\big) \vDash \forall\,\delta \in \mathit{Deriv}_\phi.\ \delta\,J' \mathrel{-\!\!*} \delta\,J \quad \text{DERIV}^*\text{-MAP}$$

$$\begin{array}{r}\big(\forall\,\delta \in \mathit{Deriv}_\phi \text{ s.t. } \phi\,\delta.\ [\![J']\!]^{+}_{\delta} * [\![J'']\!]^{+}_{\delta} \mathrel{-\!\!*} [\![J]\!]^{+}_{\delta}\big) \vDash \\ \forall\,\delta \in \mathit{Deriv}_\phi.\ \delta\,J' * \delta\,J'' \mathrel{-\!\!*} \delta\,J\end{array} \quad \text{DERIV}^*\text{-MAP}_2$$

4.4 Advanced Model

Finally, as an advanced topic, we briefly discuss a further improvement on the model of the invariant connective. This advanced model admits the following *merger* of invariants for any $\delta \in \mathit{Deriv}$:

$$\frac{\mathcal{N} + \mathcal{N}' \subseteq \mathcal{N}''}{[\![\mathtt{inv}^{\mathcal{N}}\,P]\!]_\delta * [\![\mathtt{inv}^{\mathcal{N}'}\,Q]\!]_\delta \vDash [\![\mathtt{inv}^{\mathcal{N}''}(P * Q)]\!]_\delta}\ \mathtt{inv}\text{-MERGE}$$

Here, $\mathcal{N}$ and $\mathcal{N}'$ are disjoint namespaces and $\mathcal{N}''$ is a namespace that includes both namespaces.

Model For that, we update the judgment data type *Judg* into the following:

$$\mathit{Judg} \ni J \quad ::= \quad \propto_{\mathcal{N}} P$$

And define the interpretation of the invariant connective $\mathtt{inv}^{\mathcal{N}}\, P$ as follows:

$$[\![\mathtt{inv}^{\mathcal{N}}\, P]\!]_{\delta} \quad \triangleq \quad \square\ \delta\,(\propto_{\mathcal{N}} P) \tag{4.21}$$

Notably, the invariant token $\mathrm{inv}^{\mathcal{N}}\, P$ does not appear at all in this definition.

Finally, we define the judgment interpretation $[\![\ ]\!]^{+}_{\delta}$ as follows:

$$[\![\propto_{\mathcal{N}} P]\!]^{+}_{\delta} \quad \triangleq \quad {}_{\mathcal{N}}\!\!\Rrightarrow^{\mathrm{Winv}\,[\![\ ]\!]_{\delta}}_{\varnothing} \Big([\![P]\!]_{\delta} * \big([\![P]\!]_{\delta} \mathrel{-\!\!*} {}_{\varnothing}\!\!\Rrightarrow^{\mathrm{Winv}\,[\![\ ]\!]_{\delta}}_{\mathcal{N}} \top \big) \Big) \tag{4.22}$$

This model (4.22) is quite tricky, using *mask-changing* fancy updates in the style of inv-acc-ch. We get the content $[\![P]\!]_{\delta}$ by consuming the namespace $\mathcal{N}$ and recover the namespace $\mathcal{N}$ by restoring $[\![P]\!]_{\delta}$. This advanced model (4.22) is very close to the *accessor*-style model of the invariant used in the latest versions of Iris (Iris Team, 2023a).

This enriched model is designed so that the invariant access rules inv-acc and inv-acc-ch hold.

The allocation rule inv-alloc holds for this enriched model of the invariant, because the following holds for any δ by inv-acc-ch:

$$\mathrm{inv}^{\mathcal{N}}\, P \ \vDash \ [\![\propto_{\mathcal{N}} P]\!]^{+}_{\delta}$$

Here, we benefit a lot from the fact that the semantic conversion $\square\ \delta\,(\propto_{\mathcal{N}} P)$ of the invariant connective (4.21) is *resource-dependent*.

The merger rule inv-merge we showed above follows from deriv-map$_2$, combining two accessors of the semantic judgments of the invariants.

How the Paradox Is Avoided As discussed in § 3.4.2, we cannot *directly* store the fancy update $\Rrightarrow$ inside an invariant, because that leads to a contradiction by the paradox Theorem 3.9 presented in § 3.4.1. But the advanced invariant model considered above (4.21) seems to contain fancy updates through the judgment $\propto_{\mathcal{N}}$ (4.22).[3] This does not lead to a contradiction, thanks to the indirection by the derivability predicate.

To see this, let us consider a syntactic proposition $\mathsf{bad}^{\gamma}_{*} \in \mathit{nProp}$ and a judgment $\mathtt{falsy} \in \mathit{Judg}$ interpreted as follows, for a construction analogous to the paradox Theorem 3.9:

$$[\![\mathsf{bad}^{\gamma}_{*}]\!]_{\delta} \quad \triangleq \quad \boxed{\mathtt{S}}^{\gamma} \vee \square\ \delta\ \mathtt{falsy}$$

$$[\![\mathtt{falsy}]\!]^{+}_{\delta} \quad \triangleq \quad \Rrightarrow^{\mathrm{Winv}\,[\![\ ]\!]_{\delta}}_{\bullet} \bot$$

The proposition $\boxed{\mathtt{S}}^{\gamma}$ represents the start state, just like Theorem 3.9. Here, we write $\bullet$ for *InvName*. The judgment $\mathtt{falsy}$ represents the contradiction under the fancy update. From the parameterized judgment semantics $[\![\ ]\!]^{+}$, we get the best derivability predicate der. We can create a Nola invariant $\mathrm{inv}^{\bullet}\ \mathsf{bad}^{\gamma}_{*}$ just like (3.24) in the proof of Theorem 3.9:

$$\vDash\ \Rrightarrow^{\mathrm{Winv}\,[\![\ ]\!]_{\mathrm{der}}}_{\bullet} \big(\exists \gamma.\ \mathrm{inv}^{\bullet}\ \mathsf{bad}^{\gamma}_{*} \big)$$

[3] For a more radical approach, extending the idea of Footnote 2, we can even set *Judg* to syntactic propositions *nProp* extended with the fancy update ${}_{\mathcal{E}}\mathtt{fupd}_{\mathcal{E}'}\ P$, Hoare triples, etc. (recall § 3.4.2), because the judgment interpretation $[\![\ ]\!]^{+}_{\delta} \colon \mathit{Judg} \to \mathit{iProp}$ can depend on the global proposition interpretation $[\![\ ]\!]_{\delta} \colon \mathit{nProp} \to \mathit{iProp}$ and thus express Nola's world satisfactions.

Accessing this invariant under the finished state $\boxed{\mathrm{F}}^{\gamma}$ leads to a contradiction after an extended fancy update $\Mapsto^{\mathsf{Winv}\,[\![\ ]\!]_{\mathsf{der}}}$, like (3.25):

$$\mathsf{inv}^{\bullet}\,\mathsf{bad}_{*}^{\gamma} \ * \ \boxed{\mathrm{F}}^{\gamma} \ \vDash \Mapsto_{\bullet}^{\mathsf{Winv}\,[\![\ ]\!]_{\mathsf{der}}} \ \bot \tag{4.23}$$

To cause the contradiction

$$\vDash \Mapsto_{\bullet}^{\mathsf{Winv}\,[\![\ ]\!]_{\mathsf{der}}} \bot \tag{4.24}$$

like our paradox Theorem 3.9, the following entailment should hold, like (3.28):

$$\mathsf{inv}^{\bullet}\,\mathsf{bad}_{*}^{\gamma} \ \vDash \Mapsto_{\bullet}^{\mathsf{Winv}\,[\![\ ]\!]_{\mathsf{der}}} \ \bot$$

To prove this, we would open the invariant $\mathsf{inv}^{\bullet}\,\mathsf{bad}_{*}^{\gamma}$ to get the content $[\![\mathsf{bad}_{*}^{\gamma}]\!]_{\mathsf{der}} = \boxed{\mathrm{S}}^{\gamma} \lor \Box\,\mathrm{der}\,\texttt{falsy}$. The proof is indeed done for the case where the content is the second disjunct, because $\mathrm{der}\,\texttt{falsy}$ implies $[\![\texttt{falsy}]\!]_{\mathsf{der}}^{+} = \Mapsto_{\bullet}^{\mathsf{Winv}\,[\![\ ]\!]_{\delta}} \bot$ by the soundness of der (Theorem 4.3). For the case where the content is the first disjunct $\boxed{\mathrm{S}}^{\gamma}$, the start state, we would turn it into the finished state $\boxed{\mathrm{F}}^{\gamma}$ to get a contradiction. But the entailment (4.23) is *insufficient*. We need to *close the invariant* by making the second disjunct, and for that we need to prove the following entailment stronger than (4.23):

$$\mathsf{inv}^{\bullet}\,\mathsf{bad}_{*}^{\gamma} \ * \ \boxed{\mathrm{F}}^{\gamma} \ \vDash \ \mathrm{der}\,\texttt{falsy} \tag{4.25}$$

Fortunately, the entailment (4.25) does not hold, thanks to the under-approximation by the derivability predicate der. In this way, the contradiction (4.24) is avoided in our situation, unlike the paradox Theorem 3.9.

Chapter 5

Case Study: Strong Normalization under a Stratified Type System

Well-typed programs cannot "go wrong"

Robin Milner, *A Theory of Type Polymorphism in Programming*

This chapter presents a case study of our framework, verifying *strong normalization* (i.e., termination) of functional programs under a stratified type system that supports higher-order references. Section 5.1 presents our target type system. Section 5.2 verifies the strong normalization guarantee of this type system by constructing a logical relation using Nola's later-free invariants.

5.1 Our Target Type System

It is well known that unrestricted higher-order shared mutable references cause non-termination by Landin's knot Code 1.4 (§ 1.4). To ensure termination, we design a type system that supports higher-order references but systematically avoids situations like Landin's knot. Our type system *stratifies* the set of types, annotating types with *levels* $i \in \mathbb{N}$.

Types Our types have the following syntax:

$$\begin{aligned} \mathit{Typ}_i \ni T_i, U_i, V_i \ ::=\ & \texttt{nat} \mid \texttt{bool} \mid \texttt{unit} \mid T_i \times U_i \mid T_i \wedge U_i \\ \mid\ & \texttt{ref}^o_k\, T_k \mid T_i \to_j U_i \ \ (j \le i) \\ \mid\ & \texttt{rec}\, X_j.\, T_j \ \ (j \le i) \mid \forall X_k.\, T_i \mid \exists X_k.\, T_i \\ \mid\ & X_j \ \ (\text{either } j = i \text{ and guarded or } j < i) \end{aligned}$$

The data type Typ_i for syntactic types is indexed by the *level* $i \in \mathbb{N}$. We can always convert a type T_i into a type T_j of a higher level $j \ge i$. We omit level annotations when they are not relevant.

As usual, our type system has the *value types* $\texttt{nat}, \texttt{bool}, \texttt{unit}$ for the natural numbers, booleans and unit value as well as *pair types* $T_i \times U_i$. Our type system also supports *intersection types* $T_i \wedge U_i$.

A key feature of our type system is the *shared mutable reference type* $\texttt{ref}^o_k\, T_k$. Notably, its body type T_k can have *any* level $k \in \mathbb{N}$, regardless of the level i of the reference type. The subscript k in $\texttt{ref}^o_k$ clarifies the level of the body type T_k. The superscript number $o \in \mathbb{Z}$ is the *offset* of the location of the reference's body from the reference's location value.

Another key feature of our type system is the *function type* (a.k.a. *closure type*) $T_i \to_j U_i \in \mathit{Typ}_i$. The arrow $\to_j$ of the function type is annotated with the *level* j

for executing the function, which should be no higher than the level i of the function type (i.e., $j \le i$). This level j means that the *references* accessed during the function execution is limited to those $\mathtt{ref}^o_k\ V$ whose body type level k is *lower than* j (i.e., $k < j$). Also, the function type *ensures* that the function execution *always terminates* for any argument value.

Our type system also supports *recursive types* $\mathtt{rec}\ X_j.\ T_j$, *universal types* $\forall X_k.\ T_i$, and *existential types* $\exists X_k.\ T_i$. We write X_i for a type variable of level i. Any occurrence of a type variable X_j in a type Typ_i of the level i should be bound by a variable binder (of a recursive, universal, or existential type) and also satisfy either of the following constraints:

Guarded the type variable's level j equals the surrounding level i (i.e., $j = i$), and the occurrence of the type variable is guarded by the reference type constructor $\mathtt{ref}^o$; or

Lower-level the type variable's level j is strictly lower than the surrounding level i (i.e., $j < i$).

For example, the type $\mathtt{list}_i\ T_i$ for the *shared mutable infinite singly linked list* of the element type T_i, corresponding to the separation logic predicates $\mathrm{ilist}^{\mathcal{N}}\ \Phi$ / $\mathtt{ilist}^{\mathcal{N}}\ \Phi\ \ell$ in the verification example of § 3.3, can be expressed in our type system as follows:

$$\mathtt{list}_i\ T_i \quad \triangleq \quad \mathtt{rec}\ X_i.\ \mathtt{ref}^0_i\ T_i \land \mathtt{ref}^1_i\ X_i \tag{5.1}$$

It is defined as the recursive type $\mathtt{rec}$, whose body is the intersection $\land$ of the reference to the head $\mathtt{ref}^0_i\ T_i$ and the reference to the tail $\mathtt{ref}^1_i\ X_i$, where the type variable X_i recursively refers to the list type $\mathtt{list}_i\ T_i$ itself. Here, the recursive occurrence of the variable X_i is valid, because it is guarded by the reference type constructor $\mathtt{ref}_1$.

Subtyping Our type system has the *subtyping judgment* $T_i \le U_j$, saying that any object of the type T_i can always be retyped as U_j. Notably, the levels of the two types T_i, U_j can be different.

Subtyping is reflexive and transitive:

$$T \le T \quad \le\text{-REFL} \qquad\qquad \frac{T \le U \quad U \le V}{T \le V}\ \le\text{-TRANS}$$

The pair type and intersection types satisfy the standard subtyping rules:

$$\frac{T \le T' \quad U \le U'}{T \times U \ \le\ T' \times U'}\ \le\text{-}\times \qquad \frac{V \le T \quad V \le U}{V \le T \land U}\ \le\text{-}\land\text{-INTRO}$$

$$T \land U \ \le\ T \quad \le\text{-}\land\text{-ELIM}_\mathrm{L} \qquad T \land U \ \le\ U \quad \le\text{-}\land\text{-ELIM}_\mathrm{R}$$

The reference type satisfies the following subtyping rule:

$$\frac{T \le U \quad U \le T \quad i \le j}{\mathtt{ref}^o_i\ T \ \le\ \mathtt{ref}^o_j\ U}\ \le\text{-}\mathtt{ref}$$

This rule says that, if the types T and U are a subtype of each other and the level i is no higher than the level j, then a reference of the body type T and body level i can be retyped as a reference of the body type U and body level j, keeping the offset o.[1]

[1] We also allow using the subtyping rule for the reference type $\le$-$\mathtt{ref}$ *coinductively*. This means that we can assume the final proof goal to prove the premises $T \le U$, $U \le T$ of this rule.

We can use such coinductive reasoning for subtyping on *recursive types* whose self-reference is guarded by the reference type. For example, we can prove $\mathtt{list}_i\ T \le \mathtt{list}_j\ U$ under the premises $T \le U$, $U \le T$ and $i \le j$ for the infinite singly linked list type $\mathtt{list}_i\ T$ (5.1).

The function type satisfies the following subtyping rule:

$$\frac{T' \leq T \quad U \leq U' \quad i \leq j}{T \to_i U \;\leq\; T' \to_j U'}\ \ \leq\text{-}\to$$

Note that the level of function execution can be relaxed (i.e., get higher).

We also have the following standard subtyping rules for recursive types, universal types, and existential types:

$$\mathtt{rec}\ X_j.\, T_j \;\leq\; T_j\left[\,\mathtt{rec}\ X_j.\, T_j \,/\, X_j\,\right] \quad \leq\text{-}\mathtt{rec}\text{-}\textsc{unfold}$$

$$T_j\left[\,\mathtt{rec}\ X_j.\, T_j \,/\, X_j\,\right] \;\leq\; \mathtt{rec}\ X_j.\, T_j \quad \leq\text{-}\mathtt{rec}\text{-}\textsc{fold}$$

$$\frac{\forall V_k.\ \left(U \;\leq\; T_i\,[V_k/X_k]\right)}{U \;\leq\; \forall X_k.\, T_i}\ \leq\text{-}\forall\text{-}\textsc{intro} \qquad \forall X_k.\, T_i \;\leq\; T_i\,[V_k/X_k] \quad \leq\text{-}\forall\text{-}\textsc{elim}$$

$$T_i\,[V_k/X_k] \;\leq\; \exists X_k.\, T_i \quad \leq\text{-}\exists\text{-}\textsc{intro} \qquad \frac{\forall V_k.\ \left(T_i\,[V_k/X_k] \;\leq\; U\right)}{\exists X_k.\, T_i \;\leq\; U}\ \leq\text{-}\exists\text{-}\textsc{elim}$$

Typing Judgments Our type system uses the *typing judgment* of the form $\Gamma \vdash e :_i T$. The *type context* Γ is a list $\overline{x\colon U}$ of typed objects. The *execution level* $i \in \mathbb{N}$ of the typing judgment means that the *references* accessed during the execution of the program e are limited to those $\mathtt{ref}^{o}_{j}\ V$ whose body type level j is *lower than* i (i.e., $j < i$). Notably, our typing judgment *ensures strong normalization*, i.e., ensures that the program e *always terminates*. In summary, our typing judgment $\Gamma \vdash e :_i T$ says that, under the type context Γ, the program e *always terminates*, without accessing references of levels no lower than i, and produces a value of the type T.

The objects in the type context can freely be reordered, discarded, and duplicated, i.e., our type system is *not substructural*:

$$\frac{\overline{x\colon U} \vdash e :_i T \qquad \{\overline{x\colon U}\} \subseteq \{\overline{x'\colon U'}\}}{\overline{x'\colon U'} \vdash e :_i T}\ \textsc{ty-}\subseteq$$

The typing judgment is monotone over the execution level and can be modified with respect to subtyping:

$$\frac{\Gamma \vdash e :_i T \qquad i \leq j}{\Gamma \vdash e :_j T}\ \textsc{ty-lev} \qquad \frac{\overline{x\colon U} \vdash e :_i T \qquad \overline{U' \leq U} \qquad T \leq T'}{\overline{x\colon U'} \vdash e :_i T'}\ \textsc{ty-}\leq$$

We have standard rules for constants, pairs, and control flows:

$$\frac{n \in \mathbb{N}}{\vdash n :_0 \mathsf{nat}}\ \textsc{ty-nat} \qquad \frac{b \in \mathbb{B}}{\vdash b :_0 \mathsf{bool}}\ \textsc{ty-bool} \qquad \vdash () :_0 \mathtt{unit} \quad \textsc{ty-unit}$$

$$\frac{\Gamma \vdash e :_i T \qquad \Gamma \vdash e' :_i U}{\Gamma \;\vdash\; (e, e') :_i\ T \times U}\ \textsc{ty-pair}$$

$$\frac{\Gamma \vdash e :_i T \times U}{\Gamma \;\vdash\; e.1 :_i\ T}\ \textsc{ty-fst} \qquad \frac{\Gamma \vdash e :_i T \times U}{\Gamma \;\vdash\; e.2 :_i\ U}\ \textsc{ty-snd}$$

$$\frac{\Gamma \vdash e :_i T \qquad x\colon T, \Gamma \vdash e' :_i U}{\Gamma \;\vdash\; \left(\mathsf{let}\, x := e\ \mathsf{in}\ e'\right) :_i\ U}\ \textsc{ty-let} \qquad \frac{\Gamma \vdash e :_i T \qquad \Gamma \vdash e' :_i U}{\Gamma \;\vdash\; (e; e') :_i\ U}\ \textsc{ty-seq}$$

$$\frac{\Gamma \vdash e :_i \mathsf{bool} \quad \Gamma \vdash e_t :_i T \quad \Gamma \vdash e_f :_i T}{\Gamma \;\vdash\; \left(\mathsf{if}\ e\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f\right) :_i\ T}\ \textsc{ty-if}$$

Note that the level i of the typing judgment is propagated to subexpressions.

We have the following rule for a non-deterministic natural number ndnat:

$$\vdash \mathsf{ndnat} :_0 \mathsf{nat} \quad \textsc{ty-ndnat}$$

Our type system also supports *concurrency* by thread forking $\mathsf{fork}\,\{\, e \,\}$:

$$\frac{\Gamma \vdash e :_i T}{\Gamma \,\vdash\, \mathsf{fork}\,\{\, e \,\} :_i \mathsf{unit}} \quad \textsc{ty-fork}$$

Our type system provides the following typing rules for allocating, reading from, and writing to a *shared mutable reference* ref:

$$\frac{\Gamma \vdash e :_j T \qquad i < j}{\Gamma \,\vdash\, \mathsf{ref}\, e :_j \mathsf{ref}^0_i\, T} \quad \textsc{ty-ref} \qquad \frac{\Gamma \vdash e :_j \mathsf{ref}^0_i\, T \qquad i < j}{\Gamma \,\vdash\, !e :_j T} \quad \textsc{ty-load}$$

$$\frac{\Gamma \,\vdash\, e :_j \mathsf{ref}^0_i\, T \qquad \Gamma \vdash e' :_j T \qquad i < j}{\Gamma \,\vdash\, e \leftarrow e' :_j \mathsf{unit}} \quad \textsc{ty-store}$$

Importantly, the level j of execution should be more than the body level i of the reference type $\mathsf{ref}^0_i\, T$. Similarly, our type system provides the following rule for the fetch-and-add primitive faa:

$$\frac{\Gamma \vdash e :_j \mathsf{ref}^0_i\, \mathsf{nat} \qquad \Gamma \vdash e' :_j \mathsf{nat} \qquad i < j}{\Gamma \,\vdash\, \mathsf{faa}\; e\; e' :_j \mathsf{nat}} \quad \textsc{ty-faa}$$

We also have the following rule for the reference offset operation:

$$\frac{\Gamma \vdash e :_j \mathsf{ref}^o_i\, T}{\Gamma \,\vdash\, e + o' :_j \mathsf{ref}^{o-o'}_i\, T}$$

The type system has the following standard rules for creating and calling a function:

$$\frac{x \colon T,\; \Gamma \,\vdash\, e :_i U}{\Gamma \,\vdash\, \mathsf{fun}\,(x)\,\{\, e \,\} :_0 T \to_i U} \quad \textsc{ty-fun} \qquad \frac{\Gamma \vdash e :_i T \to_i U \qquad \Gamma \vdash e' :_i T}{\Gamma \,\vdash\, e(e') :_i U} \quad \textsc{ty-call}$$

Note that a function $\mathsf{fun}\,(x)\,\{\, e \,\}$ (or *closure*) can capture the type context Γ.

Recall that our typing judgment ensures *strong normalization*. This means that we should not support general recursion. Instead, we introduce combinators for recursions that are *guaranteed to terminate*. For an interesting example, we introduce the following *higher-order function* for *function iteration* $\mathsf{fiter}(f)((n, x))$, which applies the argument function f to the value x iteratively n times:

$$\mathsf{fun}\;\; \mathsf{fiter}(f)\; \Big\{\; \mathsf{fun}_{\mathit{self}}((n, x))\; \Big\{\; \mathsf{if}\;\; n \neq 0\;\; \mathsf{then}\;\; \mathit{self}\big((n-1, f(x))\big)\;\; \mathsf{else}\;\; x \;\Big\}\;\Big\}$$

Here, we write $\mathsf{fun}_{\mathit{self}}$ for a recursive function that binds itself to a variable self. We can add the following typing rule for this function fiter:

$$\frac{\Gamma \vdash e :_j T \to_i T}{\Gamma \,\vdash\, \mathsf{fiter}\, e :_j \mathsf{nat} \times T \to_i T} \quad \textsc{ty-fiter}$$

For example, for the shared mutable infinite list type $\mathtt{list}_i\, T$ defined above (5.1), our type system can derive the following typing judgment, assuming $i < j$:

$$\begin{aligned} &f : \mathsf{ref}^0_i\, T \to_j \mathsf{unit} \;\;\vdash \\ &\qquad \mathsf{fiter}\,\big(\mathsf{fun}\,(\ell)\,\{\, f(\ell);\; !(\ell + 1) \,\}\big) :_0 \mathsf{nat} \times \mathtt{list}_i\, T \to_j \mathtt{list}_i\, T. \end{aligned}$$

The function $\mathsf{fiter}\,\big(\mathsf{fun}\,(\ell)\,\{\, f(\ell);\; !(\ell + 1) \,\}\big)$ applies the function $f \colon \mathsf{ref}^0_i\, T \to_j \mathsf{unit}$ to the first n-th elements of the singly linked list starting at ℓ. Notably, the function f can *mutate* the list elements. Recall that our function type and typing judgment ensure *termination*. Like in the verification example of § 3.3, this typing judgment verifies that the iteration *always terminates* for any function f that always terminates.

5.2 Verifying Strong Normalization with Nola's Invariants

Now let us verify the termination guarantee of our type system.

Problem with Iris Invariants A standard technique to verify a type system is to construct *logical relations*, i.e., semantic predicates parameterized over types. For local reasoning about state mutation, it is common to construct such predicates in separation logic. For example, RustBelt (Jung et al., 2018a) (reviewed in § 6.1.2) verified Rust's ownership type system by modeling Rust's types as *separation logic predicates* in Iris. Timany et al. (2023) illustrate how to construct logical relations for a simple type system with references in Iris.

We would obtain the following by directly applying such logical relation construction to our type system, ignoring the levels altogether:[2]

$$[\![\overline{v\colon U} \vdash e\colon T]\!] \;\triangleq\; \big[\ast\,\overline{[\![U]\!]\,v}\big]\; e\; \big[[\![T]\!]\big]_{\mathcal{N}} \qquad [\![T \le U]\!] \;\triangleq\; \forall v.\; \square\big([\![T]\!]\,v \mathrel{-\!\!*} [\![U]\!]\,v\big)$$

$$[\![\mathsf{nat}]\!]\,v \;\triangleq\; v \in \mathbb{N} \qquad [\![T \times U]\!]\,v \;\triangleq\; \exists u, u' \text{ s.t. } v = (u, u').\; [\![T]\!]\,u \ast [\![U]\!]\,u'$$

$$[\![\mathsf{ref}^{o}\,T]\!]\,v \;\triangleq\; \exists \ell \in \mathit{Loc} \text{ s.t. } v = \ell.\; \boxed{\exists w.\; (\ell + o) \mapsto w \ast [\![T]\!]\,w}^{\mathcal{N}}$$

$$[\![T \to U]\!]\,v \;\triangleq\; \forall u.\; \big[[\![T]\!]\,u\big]\; v\,u\; \big[[\![U]\!]\big]_{\mathcal{N}}$$

Here, each type T is interpreted as an Iris predicate over values $[\![T]\!]\colon \mathit{Val} \to \mathit{iProp}$. The interpretation $[\![\;]\!]\colon \mathit{Typ} \to \mathit{Val} \to \mathit{iProp}$ is defined by induction over the structure of the types Typ. The typing judgment is interpreted as a *total* Hoare triple, because our goal is to verify *total correctness*, including the termination guarantee. To verify each typing rule of the type system, we just need to prove its semantics version calculated from the semantics $[\![\overline{v\colon U} \vdash e\colon T]\!]$, $[\![T \le U]\!]$.

This traditional approach works for most typing rules, except those for *shared mutable references*, such as TY-LOAD. Accessing the content P of an Iris invariant $\boxed{P}^{\mathcal{N}}$ comes with the later modality $\triangleright$ (recall THOARE-IINV, § 3.1). But unfortunately, the total Hoare triple $[P]\; e\; [\Psi]_{\mathcal{E}}$ has no chance to strip off the later modality $\triangleright$ unlike the partial Hoare triple, as discussed in § 1.4. To complete verification, we replace Iris's invariants $\boxed{-}^{\mathcal{N}}$ with *Nola's later-free invariants* $\mathsf{inv}^{\mathcal{N}}$.

First Model with Nola's Invariants Fortunately, Nola's approach can naturally take advantage of the *syntactic* hierarchy in our leveled types Typ_i. Notably, we can employ *multiple instances* of Nola's invariant mechanisms.

First, we construct a syntactic proposition data type nProp_i for *each body type level* $i \in \mathbb{N}$ as follows:

$$\mathit{nProp}_i \ni P_i^{*} \;::=\; \ell \mapsto T_i \quad (\ell \in \mathit{Loc},\; T_i \in \mathit{Typ}_i)$$

This is because the proposition to be stored in the invariant is limited to the form $\exists w.\,(\ell \mapsto w) \ast [\![T_i]\!]\,w$, which is parameterized by the location $\ell \in \mathit{Loc}$ and the type $T_i \in \mathit{Typ}_i$.

Next, we add the resource algebra $\textsc{Inv}_{\mathit{nProp}_i}$ for the invariant mechanism of each

[2] We model here the judgments $\overline{v\colon U} \vdash e\colon T$ and $T \le U$ as a persistent separation logic proposition, which can depend on persistent resources. Another possible design choice is to model the judgments as a pure proposition, like $[\![\overline{v\colon U} \vdash e\colon T]\!] \triangleq\; \vDash \big[\ast\,\overline{[\![U]\!]\,v}\big]\; e\; \big[[\![T]\!]\big]_{\mathcal{N}}$ and $[\![T \le U]\!] \triangleq \forall v.\; \big([\![T]\!]\,v \vDash [\![U]\!]\,v\big)$. By choosing the former design, we have the ability to extend our type system with dynamically created relations, such as lifetime inclusion $\alpha \sqsubseteq \beta$ for Rust-style borrows (see § 6.5.1).

level $i \in \mathbb{N}$ as the component RA of the global camera $\tilde{\mathcal{G}}$:[3]

$$\textsc{Inv}_{nProp_0},\ \textsc{Inv}_{nProp_1},\ \textsc{Inv}_{nProp_2},\ \ldots$$

To use Nola's later-free invariant mechanism, we define the semantic interpretation $[\![\ ]\!]^*_i \colon nProp_i \to iProp$ of the custom syntactic proposition $nProp_i$ as follows:

$$[\![\ell \mapsto T]\!]^*_i \quad \triangleq \quad \exists w.\ \ell \mapsto w \ast [\![T]\!]_i\, w \tag{5.2}$$

where $[\![\ ]\!]_i \colon Typ_i \to Val \to iProp$ is the semantic interpretation of our types $T_i \in Typ_i$ of the level i, which we discuss below.

The definition of the type semantics $[\![T]\!]_i$ is straightforward in many cases:

$$[\![\mathtt{nat}]\!]_i\, v \ \triangleq\ v \in \mathbb{N} \qquad [\![\mathtt{bool}]\!]_i\, v \ \triangleq\ v \in \mathbb{B} \qquad [\![\mathtt{unit}]\!]_i\, v \ \triangleq\ v = ()$$
$$[\![T \times U]\!]_i\, v \ \triangleq\ \exists u, u' \text{ s.t. } v = (u, u').\ [\![T]\!]_i\, u \ast [\![U]\!]_i\, u'$$
$$[\![T \land U]\!]_i\, v \ \triangleq\ [\![T]\!]_i\, v \ast [\![U]\!]_i\, v$$

The semantics of the reference type $\mathtt{ref}^o_k\ T$ is now defined as follows:

$$[\![\mathtt{ref}^o_k\ T]\!]_i\, v \ \triangleq\ \exists \ell \in Loc \text{ s.t. } v = \ell.\ \mathsf{inv}^{\mathcal{N}}_k\left((\ell + o) \mapsto T\right) \tag{5.3}$$

Here we write inv_k for the invariant connective from the resource algebra $\textsc{Inv}_{nProp_k}$ of the level k. Recall that $\cdots \mapsto T$ will be interpreted by $[\![\ ]\!]^*_k$ given by (5.2). Notably, the interpretation of the reference type $[\![\mathtt{ref}^o_k\ T]\!]_i$ *does not depend on* the interpretation $[\![T]\!]_k$ of the body type. Thanks to this, the reference type's body type level k can be any level, regardless of the surrounding level.

We can also interpret the recursive, universal, and existential types as follows:

$$[\![\mathtt{rec}\ X_j.\ T_j]\!]_i\, v \ \triangleq\ [\![\, T_j\, [\, \mathtt{rec}\ X_j.\ T_j \,/\, X_j \,]\,]\!]_j\, v$$
$$[\![\forall X_k.\ T]\!]_i\, v \ \triangleq\ \forall V_k.\ [\![\, T\, [V_k/X_k]\,]\!]_i\, v \qquad [\![\exists X_k.\ T]\!]_i\, v \ \triangleq\ \exists V_k.\ [\![\, T\, [V_k/X_k]\,]\!]_i\, v$$

The variable substitution does not cause infinite loops in evaluating the interpretation $[\![\ ]\!]_i$, thanks to the *guardedness condition* on the occurrences of type variables X. If X occurs in a type of a level j strictly lower than i, then it is interpreted under $[\![\ ]\!]_j$, which is safe to call recursively. If X is guarded by the reference type, then it will not be semantically interpreted, because the reference type's interpretation (5.3) does not depend on the body type's interpretation.

The semantics of the function type $T \to_j U$ requires a care, because it involves an extended total Hoare triple with a *custom world satisfaction*:

$$[\![T \to_j U]\!]_i\, v \ \triangleq\ \forall u.\ \left[[\![T]\!]_i\, u\right]\ v(u)\ \left[[\![U]\!]_i\right]^{\ast_{k<j} \mathsf{Winv}_k\, [\![\ ]\!]^*_k}_{\mathcal{N}} \tag{5.4}$$

The custom world satisfaction for the function type $T \to_j U$ of the execution level j is $\ast_{k<j} \mathsf{Winv}_k\, [\![\ ]\!]^*_k$, the separating conjunction of the world satisfactions $\mathsf{Winv}_k\, [\![\ ]\!]^*_k$ for the levels $k < j$. Here, we write Winv_k for the world satisfaction predicate from the RA $\textsc{Inv}_{nProp_k}$ of the level k. This recursion equation (5.4) is well-formed, because $\mathsf{Winv}_k\, [\![\ ]\!]^*_k$ depends on the type interpretation $[\![\ ]\!]_k$ and its level k is strictly lower than i thanks to the constraints $k < j$ and $j \le i$.

[3] Unfortunately, in the current Coq mechanization of Iris, only a finite number of component RAs are allowed for the global camera. So we parameterized our proof with the maximum level $L \in \mathbb{N}$ to be used. This should not be a restriction because the levels used for one typing judgment are always bounded.

Now that we have completed the definition of the type interpretations $\llbracket\ \rrbracket_i \colon \mathit{Typ}_i \to \mathit{Val} \to \mathit{Prop}$, we can define the semantics of the typing judgment $\Gamma \vdash e :_i T$ and subtyping judgment $T \le U$ as follows:

$$\llbracket \overline{v \colon U} \vdash e :_i T \rrbracket \quad \triangleq \quad \Big[\ast\, \overline{\llbracket U \rrbracket\, v} \Big]\ e\ \Big[\llbracket T \rrbracket \Big]_{\mathcal{N}}^{\ast_{k<i} \mathsf{Winv}_k \llbracket\ \rrbracket_k^*}$$

$$\llbracket T \le U \rrbracket \quad \triangleq \quad \forall v.\ \Box \big(\llbracket T \rrbracket\, v \mathrel{-\!\!\ast} \llbracket U \rrbracket\, v \big)$$

For the total Hoare triple of the typing judgment, we use the custom world satisfaction $\ast_{k<i} \mathsf{Winv}_k \llbracket\ \rrbracket_k^*$, just like for the function type.

Final Model with Derivability We can prove most typing rules with the model introduced above. But unfortunately, we cannot prove *the subtyping rule for the reference type* $\le$-`ref`, because the proposition of the invariant token $\mathsf{inv}^{\mathcal{N}}$ cannot be directly changed.

To solve this problem, we update the model of our type system by applying the *derivability* technique of Chapter 4. To begin with, we set the syntactic data type for judgments *Judg* as follows:

$$\mathit{Judg} \ni J \ ::= \ T_i \mathrel{\dot{\le}} U_j \quad (T_i \in \mathit{Typ}_i;\ U_j \in \mathit{Typ}_j;\ i, j \in \mathbb{N})$$

We introduce a new syntactic subtyping judgment $T_i \mathrel{\dot{\le}} U_j \in \mathit{Judg}$.

Then we parameterize our type interpretation $\llbracket\ \rrbracket_i^\delta \colon \mathit{Typ}_i \to \mathit{Val} \to \mathit{iProp}$ with the *derivability predicate* $\delta \colon \mathit{Judg} \to \mathit{iProp}$. For the most part, we just replace $\llbracket\ \rrbracket_i$ with $\llbracket\ \rrbracket_i^\delta$ in the above definitions:

$$\llbracket \mathsf{nat} \rrbracket_i^\delta\, v \ \triangleq \ v \in \mathbb{N} \qquad \llbracket \mathsf{bool} \rrbracket_i^\delta\, v \ \triangleq \ v \in \mathbb{B} \qquad \llbracket \mathsf{unit} \rrbracket_i^\delta\, v \ \triangleq \ v = ()$$

$$\llbracket T \times U \rrbracket_i^\delta\, v \ \triangleq \ \exists u, u' \text{ s.t. } v = (u, u').\ \llbracket T \rrbracket_i^\delta\, u \ast \llbracket U \rrbracket_i^\delta\, u'$$

$$\llbracket T \wedge U \rrbracket_i^\delta\, v \ \triangleq \ \llbracket T \rrbracket_i^\delta\, v \ast \llbracket U \rrbracket_i\, v$$

$$\llbracket \mathsf{rec}\ X_j.\ T_j \rrbracket_i^\delta\, v \ \triangleq \ \llbracket T_j\, [\, \mathsf{rec}\ X_j.\ T_j \,/\, X_j \,] \rrbracket_j^\delta\, v$$

$$\llbracket \forall\, X_k.\ T \rrbracket_i^\delta\, v \ \triangleq \ \forall V_k.\ \llbracket T\, [V_k / X_k] \rrbracket_i^\delta\, v \qquad \llbracket \exists\, X_k.\ T \rrbracket_i^\delta\, v \ \triangleq \ \exists V_k.\ \llbracket T\, [V_k / X_k] \rrbracket_i^\delta\, v$$

The function type's interpretation (5.4) is also modified similarly, just replacing $\llbracket\ \rrbracket_i$ with $\llbracket\ \rrbracket_i^\delta$:

$$\llbracket T \to_j U \rrbracket_i^\delta\, v \quad \triangleq \quad \forall u.\ \Big[\llbracket T \rrbracket_i^\delta\, u \Big]\ v\, u\ \Big[\llbracket U \rrbracket_i^\delta \Big]_{\mathcal{N}}^{\ast_{k<j} \mathsf{Winv}_k \llbracket\ \rrbracket_k^{\delta *}}$$

Here, $\llbracket\ \rrbracket_i^{\delta *} \colon \mathit{nProp}_i \to \mathit{iProp}$ is defined as follows, just by replacing $\llbracket\ \rrbracket_i$ with $\llbracket\ \rrbracket_i^\delta$ in (5.2):

$$\llbracket \ell \mapsto T_i \rrbracket_i^{\delta *} \quad \triangleq \quad \exists w.\ \ell \mapsto w \ast \llbracket T_i \rrbracket_i^\delta\, w$$

Most significantly, we modify the interpretation of the reference type $\mathsf{ref}_k^o\ T$ from (5.3) as follows, using the *derivability predicate* δ:

$$\llbracket \mathsf{ref}_k^o\ T \rrbracket_i^\delta\, v \quad \triangleq \quad \exists U, k' \le k.\ \Box\delta(T \mathrel{\dot{\le}} U) \ast \Box\delta(U \mathrel{\dot{\le}} T) \ast \mathsf{inv}_{k'}^{\mathcal{N}} \big((\ell + o) \mapsto U \big) \tag{5.5}$$

Notably, this interpretation (5.5) does not depend on the type semantics of the body type level $\llbracket\ \rrbracket_k^\delta$. The relation between the types T and U is described by the *derivability* assertions $\delta(T \mathrel{\dot{\le}} U)$ and $\delta(U \le T)$, which do not directly depend on the type semantics.

Once we construct the parameterized type interpretation $[\![\ ]\!]_i^\delta$, we can define the parameterized *judgment interpretation* $[\![\ ]\!]_\delta^+ \colon \mathit{Judg} \longrightarrow \mathit{iProp}$ as follows:

$$[\![T_i \mathrel{\dot{\leq}} U_j]\!]_\delta^+ \quad \triangleq \quad \forall v.\ \big(\, [\![T_i]\!]_i^\delta\, v \mathrel{-\!\!*} [\![U_j]\!]_j^\delta\, v \,\big)$$

This automatically gives the best derivability predicate der and the set of good derivability predicates *Deriv* by Definition 4.1 (§ 4.3). By DERIV-INTRO and DERIV-MAP$_2$, we can derive the reflexivity (5.6) and transitivity (5.7) of the derivability for a good derivability predicate $\delta \in \mathit{Deriv}$:

$$\vDash\ \delta\,(T \mathrel{\dot{\leq}} T) \tag{5.6}$$

$$\delta\,(T \mathrel{\dot{\leq}} U)\ *\ \delta\,(U \mathrel{\dot{\leq}} V)\ \vDash\ \delta\,(T \mathrel{\dot{\leq}} V) \tag{5.7}$$

We modify the semantics of the subtyping relation $T \leq U$ as follows:

$$[\![T \leq U]\!] \quad \triangleq \quad \forall \delta \in \mathit{Deriv},\, v.\ \square\big(\, [\![T]\!]^\delta\, v \mathrel{-\!\!*} [\![U]\!]^\delta\, v \,\big)$$

Importantly, it is *universally quantified* over good derivability predicates $\delta \in \mathit{Deriv}$. Thanks to this, by DERIV-INTRO (§ 4.3), we can turn $[\![T \leq U]\!]$ into a derivability $\delta(T \mathrel{\dot{\leq}} U)$ for any good derivability predicate $\delta \in \mathit{Deriv}$:

$$[\![T \leq U]\!]\ \vDash\ \square\,\delta\,(T \mathrel{\dot{\leq}} U) \tag{5.8}$$

We update the semantics of the typing judgment $\Gamma \vdash e :_i T$ as follows:

$$[\![\overline{v \colon U} \vdash e :_i T]\!] \quad \triangleq \quad \Big[\mathop{\ast} \overline{[\![U]\!]^{\mathsf{der}}\, v} \Big]\, e\, \Big[[\![T]\!]^{\mathsf{der}} \Big]^{\ast_{k<i}\ \mathsf{Winv}_k\, [\![\ ]\!]_k^{\mathsf{der}*}}_{\mathcal{N}}$$

For the typing judgment, we use the type semantics $[\![\ ]\!]^{\mathsf{der}}$ by the *best derivability predicate* der. We use the predicate der because it satisfies the *soundness* by Theorem 4.3 (§ 4.3), which means the following in our setting:

$$\mathsf{der}\,(T \mathrel{\dot{\leq}} U)\ \vDash\ \forall v.\ [\![T]\!]^{\mathsf{der}}\, v \mathrel{-\!\!*} [\![U]\!]^{\mathsf{der}}\, v \tag{5.9}$$

Soundness of Our Target Type System Finally, using the model above, we can prove the soundness of our target type system.

Lemma 5.1 (Persistence of Type Semantics)**.** *For any type $T \in \mathit{Typ}_i$ of any level $i \in \mathbb{N}$, any value v, and any derivability predicate $\delta \colon \mathit{Judg} \longrightarrow \mathit{iProp}$, the Iris proposition $[\![T]\!]_i^\delta\, v$ is* persistent.

Proof. Clear by construction. □

Theorem 5.2 (Soundness of Typing Rules)**.** *The typing rules of our type system are sound with respect to the above-defined semantics $[\![\ ]\!]$.*

Proof. By straightforwardly checking each typing rule. We can copy typed objects $v\colon T$ (i.e., duplicate $[\![T]\!]^{\mathsf{der}}\, v$) by the previous lemma Lemma 5.1.

For the typing rules that access references such as TY-LOAD, we use Nola's later-free invariant access rule THOARE-INV and the soundness (5.9) of the best derivability predicate der.

For the reference subtyping rule ≤-ref, we turn the subtyping assumptions into derivability assertions by (5.8) and combine them with the derivability assertions inside the reference type model (5.5) using the transitivity (5.7).[4] □

[4] In order to justify *coinductive* use of the reference subtyping rule ≤-ref, we slightly modify the semantics of the subtyping judgment $[\![T \leq U]\!]$ to update the domain for the derivability predicate δ from *Deriv* to its variant Deriv_ϕ parameterized with the induction hypothesis ϕ (Definition 4.5), where ϕ models the set of coinductive hypotheses for the subtyping judgment $T \leq U$.

Corollary 5.3 (Termination Adequacy)**.** *If the typing judgment* $\vdash e :_i T$ *holds for some level* i *and type* T*, then the execution of the program* e *always terminates.*

Proof. By the previous theorem Theorem 5.2, the given typing judgment implies the total Hoare triple $\left[\top\right] e \left[\lambda\_.\, \top\right]_{\mathcal{N}}^{\ast_{k<i} \mathsf{Winv}_k \llbracket\, \rrbracket^*_k}$. This holds by the termination adequacy of the extended total Hoare triple Theorem 3.7 (§ 3.2.1), allocating the custom world satisfaction by performing the update of WINV-ALLOC over the levels $k < i$. □

Chapter 6

Later-Free Rust-Style Borrows

Neither a borrower nor a lender be

Polonius, *Hamlet* by William Shakespeare

This chapter presents the later-free Rust-style *borrow* mechanism of our framework. The mechanism allows non-step-indexed separation logic to support the features of RustBelt's lifetime logic (Jung et al., 2018a), which can model and reason about Rust-style borrows in a general, semantic way.

This chapter is organized as follows. Section 6.1 reviews Rust's borrows and RustBelt's lifetime logic. Section 6.2 explains the design of our borrow mechanism. Section 6.3 presents our proof rules for lifetimes and borrows. Section 6.4 discusses the semantic alteration by the derivability technique of Chapter 4. Section 6.5 briefly explains our semantic model for the lifetime and borrow mechanisms.

6.1 Background

6.1.1 Rust's Borrows

Rust is a programming language that uses a strong *ownership type system*, as introduced in § 1.2.2. A key feature of Rust is the *lifetime*-based *borrowing*, which derives from an idea in *region*-based ownership management (Tofte and Talpin, 1997; Gay and Aiken, 1998), especially of the Cyclone programming language (Grossman et al., 2002).[1] Borrows are very commonly used in Rust, as we briefly saw in the example Code 1.2 in § 1.2.2.

General Idea First, we give a general idea of a borrow in Rust.

Suppose an alias `a : T` to an object typed `T`. The alias has the *full ownership* of the object as described by the ownership type `T`.

Then we can create a *mutable reference* `&mut a : &'a mut T` to the object by temporarily *borrowing* the ownership from `a`. The time period for which the borrowed ownership is active is called the *lifetime* `'a` and is statically managed by the type system. The lifetime is determined at the time when the borrow is created.

Notably, there is *no direct communication* between the borrower (mutable reference) `&mut a : &'a mut T` and the lender (original owner) `a : T` ever after the borrow is created. While the lifetime `'a` is ongoing, the borrower can freely *mutate* the object

[1] Cyclone (Grossman et al., 2002) is a safe dialect of the C programming language that ensures memory safety with a strong type system under a spirit similar to Rust. Cyclone's type system manages ownership based on *memory regions* being aware of their *lifetimes*. Extending this idea, Rust directly considers *lifetimes* of objects and borrows, around which one can see regions implicitly managed.

using the ownership, *throw away* any fragments of the ownership at any time, and *split up* into multiple smaller borrowers.

Finally, at the time the lifetime `'a` ends, the lender *automatically* retrieves the ownership of the whole object typed `T`, summing up all ownership fragments thrown away by the borrowers during the borrowing.

Borrow Subdivision A key feature of the borrowing machinery is *subdivision* of a borrower into multiple smaller borrowers, throwing away some ownership fragments.

For example, suppose a mutable reference to a vector `v : &'a mut Vec<T>`. We can convert it into a mutable *iterator* `v.iter_mut() : IterMut<'a, T>`. Using it, we can iterate over the vector and get the *mutable references* to every element of the vector `v.iter_mut().collect() : Vec<&'a mut T>`. At the same time, for the conversion into a mutable iterator, we *throw away* the power to freely mutate the length and the memory block address and capacity, or throws away the ownership of the fields for that information.

Reborrowing Another key feature of the borrowing machinery is *reborrowing*. Suppose a mutable reference `a : &'a mut T` that borrows an object `o : T` under the lifetime `'a`. We can *reborrow* the ownership from this mutable reference under a *shorter* lifetime `'b` to create a new mutable reference `b : &'b mut T`. The reborrowed mutable reference `a` recovers its ownership after the lifetime `'b` of the reborrower `b` ends. The hierarchy of reborrows can be unboundedly deep.

Reborrows are very commonly used in Rust, often implicitly. For example, when we perform `v.push(a)` on a mutable reference to a vector `v : &'a mut Vec<T>`, the `push` method implicitly *reborrows* the mutable reference `v` under a shorter lifetime `'b` that lives only during the method call.

Also, reborrows can happen with *borrow subdivision*. For example, suppose we dereference a nested mutable reference `b : &'b mut &'a mut T`. Then we get `*b : &'b mut T` that *reborrows* from the inner reference `&'a mut T` under a shorter lifetime `'b`. We also throw away a fragment of `b`'s ownership, because we lose the power to mutate what the outer reference `b` points to. Therefore, this dereference is a combination of reborrowing and borrow subdivision.

Borrow as a Contract Rust-style borrows can be seen as a flexible *contract* between the borrowers and the lender, allowing them to *share* the mutable state under the access control by the lifetime. In Rust, the *contracts* are described by Rust's *ownership types* `T`. The mutable reference type `&'a mut T` expresses the contract that it gets access to the object only while the lifetime `'a` is ongoing and that an object typed `T` will be stored at the time the lifetime `'a` ends.

Automatic Lifetime Inference Note that the lifetimes of borrows are *automatically inferred* by Rust's compiler. In the earliest versions of Rust, lifetimes were limited to lexical scopes, typically introduced by curly braces `{ ... }`. But the latest versions of Rust support *non-lexical lifetimes* (Matsakis, 2017, 2022), allowing lifetimes to be much more flexible regions. Rust's compiler automatically infers non-lexical lifetimes by a clever static analysis of the program's control-flow graph, similar to more classical *live-variable analysis* (Aho et al., 2006, § 9.2.5). Rust's compiler is also carefully engineered to provide user-friendly error messages when borrows are violating the ownership principle, like Code 1.3 (§ 1.2.2).

6.1.2 RustBelt's Lifetime Logic

Lifetime Logic RustBelt's *lifetime logic* (Jung et al., 2018a, § 5; Jung, 2020, Chapter 11) semantically modeled Rust-style borrows presented above as an advanced form of *propositional sharing* in the *Iris separation logic*.[2]

RustBelt's lifetime logic models the mutable reference `&α mut T` using a new proposition called the *full borrow* $\&^{\alpha}_{\mathsf{full}} P \in iProp$, which can get access to the content $P \in iProp$ while the lifetime α is ongoing.[3] Roughly speaking, the full borrow $\&^{\alpha}_{\mathsf{full}} P$ can be seen as an advanced version of Iris's invariant $\boxed{P}$ (§ 1.3.1, § 3.3.2). The full borrow connective can be freely nested to model nested mutable reference types. Also, the lifetime logic supports *subdivision* and *reborrowing* of the full borrow, modeling Rust's borrowing machinery.

RustBelt Using the lifetime logic, *RustBelt* (Jung et al., 2018a; Jung, 2020, Part II) established a semantic foundation for Rust's ownership type system. It formally verified the memory and thread safety of well-typed Rust programs for a realistic subset of Rust with various Rust APIs, including `Rc`, `Arc`, `Cell`, `RefCell`, `Mutex`, `RwLock`, and a new Rust API `GhostCell` (Yanovski et al., 2021).

RustBelt's approach is highly *extensible* in that we can extend the proof with a new feature/API of Rust just by proving some new lemmas without touching the existing proofs at all. Also, a real-world bug in Rust's `Mutex` API was found in the course of RustBelt's verification (Jung, 2017).

RustBelt has also been extended to a relaxed memory model (Dang et al., 2020), which led to the detection of a tricky real-world bug in Rust's `Arc` API that is problematic only under a relaxed memory model (Jourdan, 2018).

Also, RustBelt has been extended by *RustHornBelt* (Matsushita et al., 2022) to provide a semantic foundation for functional verification of Rust programs in the style of *RustHorn* (Matsushita et al., 2020, 2021). We illustrate RustHorn's approach later in § 7.1.1 and review RustHornBelt in § 7.1.2.

Problem: Later Modality The borrow mechanism of RustBelt's lifetime logic suffered from the *later modality* $\triangleright$ just like the invariant mechanism. For example, we have the following rule for getting access to the full borrow $\&^{\alpha}_{\mathsf{full}} P$:

$$[\alpha]_q \ast \&^{\alpha}_{\mathsf{full}} P \vDash \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} \Big((\triangleright P) \ast \Big((\triangleright P) \twoheadrightarrow \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} \big([\alpha]_q \ast \&^{\alpha}_{\mathsf{full}} P \big) \Big) \Big) \quad \textsc{LftL-bor-acc}$$

What we get out of a full borrow is only $\triangleright P$, the content proposition P weakened by the later modality, like the access rules PHOARE-IINV and IINV-ACC for Iris's invariants.

This is because RustBelt's lifetime logic is achieved via *indexed semantics*, where the reference to *iProp* in the resource fo borrows is guarded by the later constructor $\blacktriangleright$, just like Iris's invariants (recall (1.19) in § 1.4, or (3.1) in § 3.1.3).

6.2 Design

The overall structure of Nola's borrow mechanism is analogous to Nola's invariant mechanism, presented in § 3.2.2. The resource algebra $\textsc{Bor}_{nProp}$ for Nola's borrow mechanism is parameterized over the *syntactic data type for propositions nProp*. Also, the *world satisfaction* $\mathrm{Wbor}_M [\![\ ]\!]$ is parameterized over the *semantic interpretation* $[\![\ ]\!]$:

[2] Although it is called the lifetime 'logic', it is essentially just a *library* in Iris providing a set of separation logic propositions and proof rules for them.

[3] In mathematical expressions, we write lifetimes with Greek letters α, β, γ instead of single-quoted ASCII names `'a`, `'b`, `'c`.

nProp $\longrightarrow$ *iProp* of the syntactic propositions *nProp* (we explain the parameter M later). The difference between Nola's borrow mechanism and Nola's invariant mechanism is mainly in the 'protocols' used for propositional sharing.

Technical Comparison with RustBelt's Lifetime Logic Whereas Nola's invariant mechanism is almost identical to Iris's invariant mechanism (presented in § 3.1.2), our borrow mechanism adopts designs different from RustBelt's lifetime logic (Jung et al., 2018a), for simplicity and exploring further possibilities. We can compare our borrow mechanism with RustBelt's lifetime logic as follows.

- Our borrow mechanism provides all the core functionalities of the *full borrow* $\&^{\alpha}_{\mathsf{full}} P$ in RustBelt's lifetime logic, namely *subdivision, reborrow, and merger.*
- Aside from the *borrower token* $\mathsf{bor}^{\alpha}\, P$ that corresponds to RustBelt's full borrow $\&^{\alpha}_{\mathsf{full}} P$, we introduce two new tokens, the *lender token* $\mathsf{lend}^{\alpha}\, P$ and the *open borrower token* $\mathsf{obor}^{\alpha}\, P$.

  Roughly speaking, they correspond to the *later-free* version of subformulas appearing in proof rules of RustBelt's lifetime logic:

  $$\mathsf{lend}^{\alpha}\, P \quad\approx\quad \dagger\alpha \twoheadrightarrow\!\!\Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} P \quad \text{in } \textsc{LftL-borrow}$$

  $$\mathsf{obor}^{\alpha}_{q}\, P \quad\approx\quad P \twoheadrightarrow\!\!\Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \big(\, [\alpha]_q \ast \&^{\alpha}_{\mathsf{full}} P \,\big) \quad \text{in } \textsc{LftL-bor-acc}$$

  $$\text{or more generally}\quad \forall Q.\ Q \ast \big(\, \dagger\alpha \ast Q \twoheadrightarrow\!\!\Rrightarrow_{\varnothing} P \,\big) \twoheadrightarrow\!\!\Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \big(\, [\alpha]_q \ast \&^{\alpha}_{\mathsf{full}} Q \,\big)$$
  $$\text{in } \textsc{LftL-bor-acc-strong}$$

  So conceptually, our introduction of the lender and open borrower tokens are not significant changes.

  However, modeling such notions as tokens brings technical advantages. First, our tokens are timeless, while these subformulas of RustBelt's lifetime logic are not. More significantly, in the Nola framework, our tokens can easily be expressed as *nProp*, as they do not depend on $[\![\ ]\!]$, while Nola's counterparts of these subformulas would contain the fancy update with the world satisfaction $\mathsf{Wbor}_M\,[\![\ ]\!]$, which is hard to express in *nProp* (recall § 3.4.2).

- Our borrow mechanism does not directly provide the counterparts of the *atomic borrows* $\&^{\alpha/\mathcal{N}}_{\mathsf{at}} P$ and its variant, the *non-atomic borrow* $\&^{\alpha/p.\mathcal{N}}_{\mathsf{na}} P$, provided by RustBelt's lifetime logic.

  These propositions intuitively provide the functionality of the full borrow $\&^{\alpha}_{\mathsf{full}} P$ stored in the (atomic) invariant $\boxed{-}^{\mathcal{N}}$ and its slight variant, the non-atomic invariant $\mathsf{NaInv}^{p.\mathcal{N}}$. RustBelt could not directly store full borrows in Iris's atomic and non-atomic invariants, because Iris's invariants put the *later modality* $\triangleright$ in accessing the contents (e.g., IINV-ACC). For example, if one simply constructs an invariant $\boxed{\&^{\alpha}_{\mathsf{full}} P}^{\mathcal{N}}$ that stores the full borrow $\&^{\alpha}_{\mathsf{full}} P$, opening it gives $\triangleright \&^{\alpha}_{\mathsf{full}} P$, which is useless just like the invariant under the later modality $\triangleright \boxed{P}^{\mathcal{N}}$ is useless (recall Remark 3.5, § 3.1.3). As a workaround, RustBelt's lifetime logic split the borrow into the persistent part (called the *indexed borrow*) and the timeless part (some token).

  We do not need such a workaround, because Nola's invariant mechanism is just *later-free*. Once we build Nola's borrow mechanism with some syntactic proposition *nProp*, we can use it for the syntactic proposition for Nola's invariant mechanism to enable storing Nola's borrower token $\mathsf{bor}^{\alpha}\, P$ in Nola's later-free invariants (see also the examples in § 6.3.3). Also, we can store Nola's borrower token in Iris's invariants without problems, because the token is *timeless* unlike RustBelt's full borrow.

- For flexibility, our borrow mechanism does not assume anything on the expressivity and structure of *nProp*. In particular, *nProp* may not even have the separating conjunction $*$.

  We slightly redesigned proof rules for this. For example, RustBelt's lifetime logic has the following rule to split $\&^{\alpha}_{\mathsf{full}}(P * Q)$ into $\&^{\alpha}_{\mathsf{full}}P$ and $\&^{\alpha}_{\mathsf{full}}Q$:

  $$\&^{\alpha}_{\mathsf{full}}(P * Q) \;\vDash {\Rrightarrow}_{\mathcal{N}_{\mathrm{lft}}} \left(\&^{\alpha}_{\mathsf{full}}P \;*\; \&^{\alpha}_{\mathsf{full}}Q\right) \quad \textsc{LftL-bor-split}$$

  We do not directly adopt this rule, because *nProp* may not have the separating conjunction $*$. Instead, we made the borrow subdivision rule include this kind of splitting (obor-subdiv, § 6.3.2), enriching RustBelt's original rule (LftL-bor-acc-strong). Similarly, RustBelt's lifetime logic has the following rule that merges $\&^{\alpha}_{\mathsf{full}}P$ and $\&^{\alpha}_{\mathsf{full}}Q$ into $\&^{\alpha}_{\mathsf{full}}(P * Q)$, the inverse of LftL-bor-split:

  $$\&^{\alpha}_{\mathsf{full}}P \;*\; \&^{\alpha}_{\mathsf{full}}Q \;\vDash {\Rrightarrow}_{\mathcal{N}_{\mathrm{lft}}} \;\&^{\alpha}_{\mathsf{full}}(P * Q) \quad \textsc{LftL-bor-merge}$$

  Instead of adopting this rule directly (for the same reason with LftL-bor-split), we introduced a proof rule that combines merger and subdivision of borrowers (obor-merge-subdiv, § 6.3.2).

  It is arguably non-trivial that the borrow mechanism can be developed not assuming anything about shared propositions *nProp*, especially since *nProp* has fundamental limitations in expressing the fancy update modality and the impredicative quantifiers (recall § 3.4.2).

- Our lifetime mechanism is completely independent from our borrow mechanism. Our proof rules work with the *basic update modality* $\dot{\Rrightarrow}$ that behaves nicely (e.g., $\dot{\Rrightarrow}$-pure), while RustBelt's proof rules depend on the fancy update modality ${\Rrightarrow}_{\mathcal{N}_{\mathrm{lft}}}$ and even the later modality $\triangleright$ (e.g., LftL-begin).

- For simplicity, we always require live lifetime tokens $[\alpha]_q$ for modifying borrows, unlike the rules LftL-bor-split and LftL-bor-merge. From our inspection, we can always take live lifetime tokens for the modified borrows in verifying Rust programs, so this requirement should not matter in verification.

6.3 Proof Rules

Now we present our proof rules for lifetimes and borrows.

6.3.1 Lifetimes

Technical Comparison with RustBelt's Lifetime Logic Our proof rules for lifetimes are almost the same as RustBelt's lifetime logic.

But ours are cleaner in that they work with the *basic update modality* $\dot{\Rrightarrow}$, which behaves nicely (e.g., $\dot{\Rrightarrow}$-pure). RustBelt's proof rules depend on the *fancy update modality* $\Rrightarrow$ and even the *later modality* $\triangleright$, such as the following rule LftL-begin for creating a fresh lifetime α:

$$\vDash {\Rrightarrow}_{\mathcal{N}_{\mathrm{lft}}} \;\exists \alpha.\; [\alpha]_1 \,*\, \Box\left([\alpha]_1 \twoheadrightarrow {}_{\mathcal{N}_{\mathrm{lft}}}{\Rrightarrow}_{\varnothing} \triangleright {}_{\varnothing}{\Rrightarrow}_{\mathcal{N}_{\mathrm{lft}}} \dagger\alpha\right) \quad \textsc{LftL-begin}$$

Basics For lifetimes, we provide two tokens, the *live* $[\alpha]_q$ and *dead* $\dagger\alpha$ lifetime tokens, just like RustBelt's lifetime logic.

A *live lifetime token* $[\alpha]_q \in iProp$ asserts with the *fraction* $q \in \mathbb{Q}_{>0}$ that the lifetime $\alpha \in Lft$ is *alive*.[4] The live lifetime token is fractional and its fraction q cannot exceed 1.

$$[\alpha]_{q+r} \;=\; [\alpha]_q * [\alpha]_r \quad \textsc{lft-live-fract} \qquad \frac{\alpha \neq \top \quad q > 1}{[\alpha]_q \;=\; \bot} \quad \textsc{lft-live-over1}$$

A *dead lifetime token* $\dagger\alpha \in iProp$ *persistently* asserts that the lifetime $\alpha \in Lft$ is *dead*.

$$\dagger\alpha \text{ is persistent} \quad \textsc{lft-dead-persist}$$

The key property is that the live and dead lifetime tokens *cannot coexist*:

$$[\alpha]_q \;*\; \dagger\alpha \;\vDash\; \bot \quad \textsc{lft-live-dead-}\bot$$

This rule enables access control by lifetimes.

We can always take a fresh *live* lifetime α, getting the witness $[\alpha]_1$:

$$\vDash \dot{\Rrightarrow} \left(\exists\, \alpha \neq \top.\; [\alpha]_1\right) \quad \textsc{lft-alloc}$$

Then we can kill the lifetime α with the witness $\dagger\alpha$ any time by consuming $[\alpha]_1$:

$$\frac{\alpha \neq \top}{[\alpha]_1 \;\vDash \dot{\Rrightarrow}\; \dagger\alpha} \quad \textsc{lft-kill}$$

In this way, a lifetime dynamically forms a *time period*.

Also, both live and dead lifetime tokens are timeless:

$$[\alpha]_q \text{ is timeless} \quad \textsc{lft-live-timeless} \qquad \dagger\alpha \text{ is timeless} \quad \textsc{lft-dead-timeless}$$

Static Lifetime and Lifetime Intersection We also provide the *static lifetime* and *lifetime intersection*, just like RustBelt's lifetime logic.

The *static lifetime* $\top \in Lft$ (`'static` in Rust) is a lifetime statically known to be alive forever. For the static lifetime $\top$, the live lifetime token is free and the dead lifetime token is prohibited:

$$\vDash\; [\top]_q \quad \textsc{lft-live-}\top \qquad \dagger\top \;=\; \bot \quad \textsc{lft-dead-}\top$$

The rule LFT-KILL has the side condition $\alpha \neq \top$ to prohibit killing the static lifetime $\top$ while allowing the live lifetime token $[\top]_q$ unrestrictedly.

For convenience, we also provide the *intersection* operation $\sqcap\colon Lft \times Lft \to Lft$ over lifetimes. The intersection lifetime $\alpha \sqcap \beta$ represents a lifetime that is alive only while both α and β are alive. The intersection is commutative, associative, and unital with the static lifetime $\top$:

$$\alpha \sqcap \beta \;=\; \beta \sqcap \alpha \quad \textsc{lft-}\sqcap\textsc{-comm} \qquad (\alpha \sqcap \beta) \sqcap \gamma \;=\; \alpha \sqcap (\beta \sqcap \gamma) \quad \textsc{lft-}\sqcap\textsc{-assoc}$$

$$\alpha \sqcap \top \;=\; \alpha \quad \textsc{lft-}\sqcap\textsc{-}\top$$

Although actual Rust does not expose lifetime intersection to users, it is useful for reasoning. The lifetime intersection satisfies the following rules for the live and dead lifetime tokens:

$$[\alpha \sqcap \beta]_q \;=\; [\alpha]_q * [\beta]_q \quad \textsc{lft-live-}\sqcap \qquad \dagger(\alpha \sqcap \beta) \;=\; \dagger\alpha \lor \dagger\beta \quad \textsc{lft-dead-}\sqcap$$

Note that the lifetime intersection is *not* idempotent $\alpha \sqcap \alpha \neq \alpha$ to support the rule LFT-LIVE-$\sqcap$ soundly.

[4] For clarity, we use the name 'live lifetime token' instead of the name 'lifetime token' used by RustBelt. Also, we adopt the notation $\dagger\alpha$ instead of RustBelt's $[\dagger\alpha]$.

Eternal Lifetime Token Unlike RustBelt's lifetime logic, we newly introduce the *eternal lifetime token* $\infty\alpha \in iProp$, which *persistently* asserts that the lifetime α is *alive forever*. The token is also timeless.

$$\infty\alpha \text{ is persistent} \quad \textsc{lft-etern-persist} \qquad \infty\alpha \text{ is timeless} \quad \textsc{lft-etern-timeless}$$

Under an eternal lifetime token, we can always get a live lifetime token $[\alpha]_q$ of some fraction q:

$$\infty\alpha \ \vDash \dot{\Rrightarrow} \ (\exists q.\ [\alpha]_q) \quad \textsc{lft-etern-live}$$

We can make a lifetime eternal by consuming a live lifetime token $[\alpha]_q$:

$$[\alpha]_q \ \vDash \dot{\Rrightarrow} \ \infty\alpha \quad \textsc{lft-eternalize}$$

The lifetime token also interacts with the static lifetime and lifetime intersection:

$$\vDash \ \infty\top \quad \textsc{lft-etern-}\top \qquad \infty(\alpha \sqcap \beta) \ = \ \infty\alpha * \infty\beta \quad \textsc{lft-etern-}\sqcap$$

Lifetime Inclusion Like RustBelt's lifetime logic, we provide *lifetime inclusion* $\alpha \sqsubseteq \beta \in iProp$ between lifetimes, which *persistently* states that the lifetime α lives no longer than the lifetime β. Unlike RustBelt's lifetime logic, our lifetime inclusion is timeless.

$$\alpha \sqsubseteq \beta \text{ is persistent} \quad \textsc{lft-}\sqsubseteq\textsc{-persist} \qquad \alpha \sqsubseteq \beta \text{ is timeless} \quad \textsc{lft-}\sqsubseteq\textsc{-timeless}$$

Under the inclusion $\alpha \sqsubseteq \beta$, from a live lifetime token $[\alpha]_q$ for α, we can take out a live lifetime token $[\beta]_r$ for β of some fraction r:

$$\alpha \sqsubseteq \beta * [\alpha]_q \ \vDash \dot{\Rrightarrow} \ (\exists r.\ [\beta]_r * ([\beta]_r \twoheadrightarrow [\alpha]_q)) \quad \textsc{lft-}\sqsubseteq\textsc{-live-acc}$$

Also, under the inclusion $\alpha \sqsubseteq \beta$, if β is dead, then α is also dead:

$$\alpha \sqsubseteq \beta * \dagger\beta \ \vDash \ \dagger\alpha \quad \textsc{lft-}\sqsubseteq\textsc{-dead}$$

Lifetime inclusion is reflexive and transitive:

$$\vDash \ \alpha \sqsubseteq \alpha \quad \textsc{lft-}\sqsubseteq\textsc{-refl} \qquad \alpha \sqsubseteq \beta * \beta \sqsubseteq \gamma \ \vDash \ \alpha \sqsubseteq \gamma \quad \textsc{lft-}\sqsubseteq\textsc{-trans}$$

Lifetime inclusion also interacts with the static lifetime and lifetime intersection:

$$\vDash \ \alpha \sqsubseteq \top \quad \textsc{lft-}\sqsubseteq\textsc{-}\top$$

$$\vDash \ \alpha \sqcap \beta \sqsubseteq \alpha \quad \textsc{lft-}\sqsubseteq\textsc{-}\sqcap\textsc{-elim} \qquad \gamma \sqsubseteq \alpha * \gamma \sqsubseteq \beta \ \vDash \ \gamma \sqsubseteq \alpha \sqcap \beta \quad \textsc{lft-}\sqsubseteq\textsc{-}\sqcap\textsc{-intro}$$

Lifetime inclusion can also be *dynamically* created. First, a dead lifetime is included by any lifetime, which dynamically happens by killing a lifetime LFT-KILL:

$$\dagger\alpha \ \vDash \ \alpha \sqsubseteq \beta \quad \textsc{lft-dead-}\sqsubseteq$$

Also, an eternal lifetime includes any lifetime, which dynamically happens by eternalizing a lifetime LFT-ETERNALIZE:

$$\infty\alpha \ \vDash \ \alpha \sqsubseteq \beta \quad \textsc{lft-etern-}\sqsubseteq$$

6.3.2 Borrows

Overall Structure The overall structure of Nola's borrow mechanism is analogous to Nola's invariant mechanism, presented in § 3.2.2.

The resource algebra $\textsc{Bor}_{nProp}$ for Nola's borrow mechanism is parameterized over the *syntactic data type for propositions nProp*, just like the resource algebra $\textsc{Inv}_{nProp}$ for invariants.

Also, Nola's borrow mechanism provides a *world satisfaction* $\mathsf{Wbor}_M\, [\![\ ]\!]$ parameterized over the *semantic interpretation* $[\![\ ]\!]\colon nProp \to iProp$ of the syntactic propositions *nProp*, just like the world satisfaction of Nola's invariant mechanism $\mathsf{Winv}\, [\![\ ]\!]$.

For flexibility, the world satisfaction for borrows $\mathsf{Wbor}_M\, [\![\ ]\!]$ also takes an extra parameter $M\colon iProp \to iProp$, the modality used for *borrow subdivision* (see OBOR-SUBDIV presented later). We require M to be an *update* modality (a new notion we introduced), which we describe below.

Update Modality We say a mapping $M\colon iProp \to iProp$ is an *update modality* if it satisfies the following properties:

$$\frac{P \vDash Q}{M\,P \vDash M\,Q}\ \textsc{upd-mono} \qquad \dot{\Rrightarrow} P \vDash M\,P\ \ \dot{\Rrightarrow}\textsc{-upd}$$

$$M\,(M\,P) \vDash M\,P\ \ \textsc{upd-idemp} \qquad (M\,P) * Q \vDash M\,(P * Q)\ \ \textsc{upd-frame}$$

For example, the basic update modality $\dot{\Rrightarrow}$ is an update modality. Also, the fancy update modality $\Rrightarrow_{\mathcal{E}}$ and the extended fancy update modality $\Rrightarrow^{W}_{\mathcal{E}}$ (mask-unchanging) are an update modality.

Extended Update Modality For an *update modality* $M\colon iProp \to iProp$, we define the *extended update modality* $M^{W}\colon iProp \to iProp$ with a *custom world satisfaction* $W \in iProp$ as follows:

$$M^{W} P \quad \triangleq \quad W \mathrel{-\!\!*} M \left(W * P \right)$$

This generalizes the extended fancy update $\Rrightarrow^{W}_{\mathcal{E}}$ (§ 3.2.1) into any update modality M instead of the fancy update $\Rrightarrow_{\mathcal{E}}$. The extended update modality M^{W} is always an update modality.

The following properties hold for the extended update modality, just like the extended fancy update $\Rrightarrow^{W}_{\mathcal{E}}$:

$$M^{\top} P = M\,P\ \ \textsc{updw-}\top \qquad \frac{W' = W * W_{+}}{M^{W} P \vDash M^{W'} P}\ \textsc{updw-expand}$$

By $\dot{\Rrightarrow}$-UPD, the extended basic update modality implies the extended update modality:

$$\dot{\Rrightarrow}^{W} P \vDash M^{W} P \quad \dot{\Rrightarrow}\textsc{w-updw}$$

Also, a double extension is equal to a single extension with a separating conjunction:

$$\left(M^{W}\right)^{W'} P = M^{W * W'} P \quad \textsc{updw-updw}$$

Basic Rules Now we are ready to present the proof rules for borrows.

First, the borrow mechanism features three tokens: the *borrower token* $\mathsf{bor}^{\alpha}\, P$, *lender token* $\mathsf{lend}^{\alpha}\, P$, and *open borrower token* $\mathsf{obor}^{\alpha}\, P$.

The tokens are all timeless:

$$\mathsf{bor}^{\alpha}\, P \text{ is timeless}\ \ \textsc{bor-timeless} \qquad \mathsf{lend}^{\alpha}\, P \text{ is timeless}\ \ \textsc{lend-timeless}$$

$$\text{obor}^{\alpha}\, P \text{ is timeless} \qquad \textsc{obor-timeless}$$

By storing an interpretation of a proposition $[\![P]\!]$, we can *create a borrow* of the proposition P under any lifetime α, getting a *borrower token* $\text{bor}^{\alpha}\, P$ and a *lender token* $\text{lend}^{\alpha}\, P$:

$$[\![P]\!] \;\vDash \dot{\Rrightarrow}^{\text{Wbor}_M [\![\ ]\!]} \left(\text{bor}^{\alpha}\, P \;*\; \text{lend}^{\alpha}\, P\right) \qquad \textsc{bor-lend-new}$$

Here, we use the *extended basic update* $\dot{\Rrightarrow}^{\text{Wbor}_M [\![\ ]\!]}$ with the *world satisfaction* for the borrow mechanism $\text{Wbor}_M [\![\ ]\!]$.

A lender token $\text{lend}^{\alpha}\, P$ can *retrieve the borrowed content* $[\![P]\!]$ after the lifetime α has died:

$$\dagger\alpha \;*\; \text{lend}^{\alpha}\, P \;\vDash M^{\text{Wbor}_M [\![\ ]\!]}\, [\![P]\!] \qquad \textsc{lend-retrieve}$$

Here, we use the *extended update* $M^{\text{Wbor}_M [\![\ ]\!]}$ for the designated update modality M of the world satisfaction $\text{Wbor}_M [\![\ ]\!]$ instead of the basic update $\dot{\Rrightarrow}$. The modality M is used for *borrow subdivision* (see OBOR-SUBDIV presented later).

A borrower token $\text{bor}^{\alpha}\, P$ can *open the borrow* by storing a live lifetime token $[\alpha]_q$. In return it gets the *borrowed content* $[\![P]\!]$ with the *open borrower token* $\text{obor}^{\alpha}_q\, P$:

$$[\alpha]_q \;*\; \text{bor}^{\alpha}\, P \;\vDash M^{\text{Wbor}_M [\![\ ]\!]} \left(\text{obor}^{\alpha}_q\, P \;*\; [\![P]\!]\right) \qquad \textsc{bor-open}$$

Notably, the borrowed content $[\![P]\!]$ is not weakened by the *later modality* $\triangleright$, unlike RustBelt's lifetime logic (recall the access rule LftL-bor-acc of RustBelt's lifetime logic presented in § 6.1.2).

An opener borrower token can *close the borrow* by going the way back. It stores the *borrowed content* $[\![P]\!]$ and recover the borrower token $\text{bor}^{\alpha}\, P$ and the live lifetime token $[\alpha]_q$:

$$\text{obor}^{\alpha}_q\, P \;*\; [\![P]\!] \;\vDash \dot{\Rrightarrow}^{\text{Wbor}_M [\![\ ]\!]} \left([\alpha]_q \;*\; \text{bor}^{\alpha}\, P\right) \qquad \textsc{obor-close}$$

Converting Tokens We can also shorten a borrower token's lifetime and prolong a lender token's lifetime using lifetime inclusion $\sqsubseteq$:

$$\beta \sqsubseteq \alpha \;*\; \text{bor}^{\alpha}\, P \;\vDash\; \text{bor}^{\beta}\, P \qquad \textsc{bor-lft} \qquad\qquad \alpha \sqsubseteq \beta \;*\; \text{lend}^{\alpha}\, P \;\vDash\; \text{lend}^{\beta}\, P \qquad \textsc{lend-lft}$$

We can also shorten an open borrower token's lifetime, additionally storing a converter for the live lifetime token $[\alpha]_q \twoheadrightarrow [\beta]_r$:

$$\beta \sqsubseteq \alpha \;*\; ([\alpha]_q \twoheadrightarrow [\beta]_r) \;*\; \text{obor}^{\alpha}_q\, P \;\vDash\; \text{obor}^{\beta}_r\, P \qquad \textsc{obor-lft}$$

Also, we can freely *fake* a borrower token $\text{bor}^{\alpha}\, P$ for any dead lifetime α:

$$\dagger\alpha \;\vDash\; \text{bor}^{\alpha}\, P \qquad \textsc{bor-fake}$$

Borrow Subdivision and Merger We can *subdivide* a borrow by the following rule:

$$\begin{array}{l} \beta \sqsubseteq \alpha \;*\; \text{obor}^{\alpha}_q\, P \;*\; \mathop{\Large *}_i [\![Q_i]\!] \;*\; \left(\dagger\beta \;*\; \mathop{\Large *}_i [\![Q_i]\!] \twoheadrightarrow M\, [\![P]\!]\right) \\ \qquad\qquad \vDash \dot{\Rrightarrow}^{\text{Wbor}_M [\![\ ]\!]} \left([\alpha]_q \;*\; \mathop{\Large *}_i \text{bor}^{\beta}\, Q_i\right) \end{array} \qquad \textsc{obor-subdiv}$$

It is a richer version of the borrow closing rule OBOR-CLOSE. This rule OBOR-SUBDIV creates new *subdivided borrowers* $\text{bor}^{\beta}\, Q_i$ instead of recovering the original borrower $\text{bor}^{\alpha}\, P$. For that, it requires the contents $\mathop{\Large *}_i [\![Q_i]\!]$ of the new borrowers and the *'converter'*

$$\dagger\beta \;*\; \mathop{\Large *}_i [\![Q_i]\!] \twoheadrightarrow M\, [\![P]\!]$$

that turns the new borrowers' contents $\mathop{\Large\ast}_i \llbracket Q_i \rrbracket$ into the the original borrower's content $\llbracket P \rrbracket$ under the update modality M with an assumption that the lifetime β has died. We can designate a new lifetime β for the new borrowers as long as it is shorter than the original lifetime α. Note that OBOR-CLOSE can be derived from OBOR-SUBDIV by setting $\beta = \alpha$ and $\bar{Q} = P$.

We can also *merge* borrowers. For that, we provide the following rule for merging and subdividing borrows, enriching the rule OBOR-SUBDIV:

$$\mathop{\Large\ast}_{j} \left(\beta \sqsubseteq \alpha_j \ast \mathsf{obor}^{\alpha_j}_{q_j} P_j \right) \ast \mathop{\Large\ast}_{i} \llbracket Q_i \rrbracket \ast \left(\dagger\beta \ast \mathop{\Large\ast}_{i} \llbracket Q_i \rrbracket \mathrel{-\!\!\ast} M \left(\mathop{\Large\ast}_{j} \llbracket P_j \rrbracket \right) \right) \\ \vDash \overset{\cdot}{\Rrightarrow}^{\mathsf{Wbor}_M \llbracket\,\rrbracket} \left(\mathop{\Large\ast}_{j} [\alpha_j]_{q_j} \ast \mathop{\Large\ast}_{i} \mathsf{bor}^{\beta}\, Q_i \right)$$

OBOR-MERGE-SUBDIV

It simply allows multiple open borrowers $\mathsf{obor}^{\alpha_j}_{q_j} P_j$ instead of just one open borrower $\mathsf{obor}^{\alpha}_{q} P$ in the rule OBOR-SUBDIV.

Reborrowing We can also *reborrow* a borrower by the following rule:

$$[\alpha]_q \ast \mathsf{bor}^{\alpha} P \vDash M^{\mathsf{Wbor}_M \llbracket\,\rrbracket} \left([\alpha]_q \ast \mathsf{bor}^{\alpha \sqcap \beta} P \ast (\dagger\beta \mathrel{-\!\!\ast} \mathsf{bor}^{\alpha} P) \right) \quad \text{BOR-REBORROW}$$

This rules *reborrows* a borrower token $\mathsf{bor}^{\alpha} P$ under the *intersection* lifetime $\alpha \sqcap \beta$. It creates a new borrower token $\mathsf{bor}^{\alpha \sqcap \beta} P$ and a 'promise' $\dagger\beta \mathrel{-\!\!\ast} \mathsf{bor}^{\alpha} P$ to obtain the original borrower token $\mathsf{bor}^{\alpha} P$ back after the lifetime β has died. As a 'catalyst', the rule also requires a live lifetime token $[\alpha]_q$ of some fraction q. Note that the lifetime intersection $\sqcap$: $\mathit{Lft} \times \mathit{Lft} \to \mathit{Lft}$ serves a lot for expressing this rule.

Lender Splitting We can also *split* a lender token $\mathsf{lend}^{\alpha} P$ into multiple lender tokens $\mathop{\Large\ast}_i \mathsf{lend}^{\alpha} Q_i$ by the following rule:

$$\mathsf{lend}^{\alpha} P \ast \left(\llbracket P \rrbracket \mathrel{-\!\!\ast} M \left(\mathop{\Large\ast}_{i} \llbracket Q_i \rrbracket \right) \right) \vDash \overset{\cdot}{\Rrightarrow}^{\mathsf{Wbor}_M \llbracket\,\rrbracket} \left(\mathop{\Large\ast}_{i} \mathsf{lend}^{\alpha} Q_i \right) \quad \text{LEND-SPLIT}$$

It additionally takes a converter $\llbracket P \rrbracket \mathrel{-\!\!\ast} M\, (\mathop{\Large\ast}_i \llbracket Q_i \rrbracket)$ from the original content $\llbracket P \rrbracket$ to the new contents $\mathop{\Large\ast}_i \llbracket Q_i \rrbracket$ under the update modality M.

Note that RustBelt's lifetime logic does not support lender splitting.

World Satisfaction Allocation We have the following rule for initializing the borrow mechanism and acquiring the world satisfaction $\mathsf{Wbor}_M \llbracket\,\rrbracket$, analogous to the rule WINV-ALLOC for the invariant mechanism:

$$\vDash \overset{\cdot}{\Rrightarrow} \left(\exists \gamma_{\mathrm{BOR}}.\ \forall \llbracket\,\rrbracket, M.\ \mathsf{Wbor}_M \llbracket\,\rrbracket \right) \quad \text{WBOR-ALLOC.}$$

It takes a fresh *ghost name* γ_{BOR}, by which the tokens $\mathsf{bor}^{\alpha} P$, $\mathsf{lend}^{\alpha} P$, $\mathsf{obor}^{\alpha} P$ and the world satisfaction $\mathsf{Wbor}_M \llbracket\,\rrbracket$ are implicitly parameterized. Notably, the obtained world satisfaction $\mathsf{Wbor}_M \llbracket\,\rrbracket$ is *universally quantified* over the semantic interpretation $\llbracket\,\rrbracket$: $\mathit{nProp} \to \mathit{iProp}$ as well as the update modality M.

6.3.3 Examples

Let us instantiate our borrow mechanism with *nProp* of Chapter 3 extended with a borrower connective $\mathsf{bor}^{\alpha} P$ interpreted as the borrower token:

$$\llbracket \mathsf{bor}^{\alpha} P \rrbracket \quad \triangleq \quad \mathsf{bor}^{\alpha} P$$

Mutable Reference to a Pair For a simple example, a mutable reference to a pair $\ell\colon$ `&α mut (T, U)` can be expressed as follows, modeling the types `T`, `U` as predicates $T, U\colon \mathit{Loc} \to \mathit{nProp}$ and assuming that the size of `T` is k:

$$\mathsf{bor}^{\alpha}\,\big(T\,\ell \;*\; U\,(\ell+k)\big)$$

For example, when `T` is an integer type `int`, then we can set $T\,\ell \triangleq \exists n \in \mathbb{Z}.\ \ell \mapsto n$ and $k = 1$.

We can split the mutable reference $\ell\colon$ `&α mut (T, U)` into the mutable references to the first component $\ell\colon$ `&α mut T` and the second component $(\ell+k)\colon$ `&α mut U`:

$$[\alpha]_q \,*\, \mathsf{bor}^{\alpha}\,\big(T\,\ell \;*\; U\,(\ell+k)\big) \;\vDash M^{\mathrm{Wbor}_M\,[\![\,]\!]}\;\big([\alpha]_q \,*\, \mathsf{bor}^{\alpha}\; T\,\ell \,*\, \mathsf{bor}^{\alpha}\; U\,(\ell+k)\big)$$

We can prove this using OBOR-SUBDIV.

Conversely, we can also merge $\ell\colon$ `&α mut T` and $(\ell+k)\colon$ `&α mut U` into $\ell\colon$ `&α mut (T, U)`:

$$[\alpha]_q \,*\, \mathsf{bor}^{\alpha}\; T\,\ell \,*\, \mathsf{bor}^{\alpha}\; U\,(\ell+k) \;\vDash M^{\mathrm{Wbor}_M\,[\![\,]\!]}\;\big([\alpha]_q \,*\, \mathsf{bor}^{\alpha}\,\big(T\,\ell \;*\; U\,(\ell+k)\big)\big)$$

We can prove this using OBOR-MERGE-SUBDIV.

Nested Mutable Reference For a more advanced example, a *nested mutable reference* $\ell\colon$ `&α mut &β mut T` can be expressed as follows, modeling the type `T` as a predicate $T\colon \mathit{Loc} \to \mathit{nProp}$:

$$\mathsf{bor}^{\alpha}\,\big(\exists \ell'.\ \ell \mapsto \ell' \;*\; \mathsf{bor}^{\beta}\,(T\,\ell')\big).$$

The location value ℓ' of the inner mutable reference is existentially quantified inside the borrower token to allow mutation.

For a small but non-trivial example, let us consider the *dereference* of the nested mutable reference $!\ell\colon$ `&(α ⊓ β) mut T`. As described in § 6.1.1, this *reborrows* the inner borrow `&β mut T` under a shorter lifetime $\alpha \sqcap \beta$. Also, it *subdivides* the outer borrow `&α mut &β mut T`, throwing away the power to mutate the location of the outer borrow.

Using our borrow mechanism, we can verify this dereference of the nested mutable reference under the *total* Hoare triple:

$$\begin{gathered}\Big[\,[\alpha \sqcap \beta]_q \,*\, \mathsf{bor}^{\alpha}\,\big(\exists \ell'.\ \ell \mapsto \ell' \;*\; \mathsf{bor}^{\beta}\,(T\,\ell')\big)\Big] \\ !\ell\ \Big[\,\lambda v.\ \exists \ell' \text{ s.t. } v = \ell'.\ [\alpha \sqcap \beta]_q \,*\, \mathsf{bor}^{\alpha\sqcap\beta}(T\,\ell')\,\Big]^{\mathrm{Wbor}_{\dot{\Rrightarrow}}\,[\![\,]\!]}\end{gathered} \tag{6.1}$$

For simplicity, we use the basic update $M = \dot{\Rrightarrow}$ for the world satisfaction. Importantly, in the total Hoare triple of (6.1), the resulting inner borrow is *free of the later modality* $\triangleright$, enabling accessing further inside the borrow. Therefore, our later-free borrow mechanism enables liveness verification of nested borrows, as we observed for our later-free invariant mechanism in the singly linked list example § 3.3.3.

The verification of this dereference goes as follows.

Proof of (6.1). For convenience, we name the content proposition of the outer borrower token `refbor` $\triangleq \exists \ell'.\ \ell \mapsto \ell' \,*\, \mathsf{bor}^{\beta}\,(T\,\ell')$. Note that the live lifetime token $[\alpha \sqcap \beta]_q$ is equal to $[\alpha]_q * [\beta]_q$ by LFT-LIVE-⊓.

By BOR-OPEN, we open the outer borrower bor^{α} `refbor` storing $[\alpha]_q$, getting an *open borrower token* obor^{α}_q `refbor` and the content

$$[\![\mathtt{refbor}]\!] \quad = \quad \exists \ell'.\ \ell \mapsto \ell' \;*\; \mathsf{bor}^{\beta}\,(T\,\ell').$$

We destruct the existential quantifier to get the location $\ell' \in \mathit{Loc}$. We perform the dereference $!\ell$ using the points-to token $\ell \mapsto \ell'$.

Then we *reborrow* the inner borrow $\mathsf{bor}^{\beta}\ (T\ \ell')$ under the lifetime α to get the target mutable reference $\mathsf{bor}^{\alpha\sqcap\beta}\ (T\ \ell')$ as well as the promise

$$\dagger\alpha \twoheadrightarrow \mathsf{bor}^{\beta}\ (T\ \ell').$$

We use the live lifetime token $[\beta]_q$ as a catalyst for this.

Finally, we should recover the lifetime token $[\alpha]$. For that, we *subdivide* the open borrower token $\mathsf{obor}^{\alpha}_{q}\ \mathtt{refbor}$, getting no borrows, i.e., setting $\bar{Q}$ to the empty list. For that, we should construct the converter

$$\dagger\alpha \twoheadrightarrow \dot{\Rrightarrow} [\![\mathtt{refbor}]\!].$$

We can create this from the points-to token $\ell \mapsto \ell'$ and the promise $\dagger\alpha \twoheadrightarrow \mathsf{bor}^{\beta}\ (T\ \ell')$ obtained by the reborrow. The assumption $\dagger\alpha$ of the converter can be used to feed the promise to get the wanted borrower token $\mathsf{bor}^{\beta}\ (T\ \ell')$ for $[\![\mathtt{refbor}]\!]$. □

Shared Reference to a Mutex-Guarded Object We can also model the *shared reference type* to a mutex-guarded object `&α Mutex<T>` in Rust. The reference `&α Mutex<T>` can get access to the content object `T` by acquiring the mutex lock. Also, the reference can be *freely shared* among multiple threads.

Unlike the model (1.15) in § 1.3.1, we now consider the *lifetime* α, which limits the time period when the shared reference can get access to the object. After the lifetime α ends, the lender retrieves the full ownership of the object.

By combining Nola's invariants with Nola's borrows, we can express a shared reference to a mutex-guarded object ℓ: `&α Mutex<T>` as follows:

$$\mathsf{inv}^{\mathcal{N}} \Big(\mathsf{bor}^{\alpha} \Big(\big(\ell \mapsto \mathsf{false} * T\,(\ell+1)\big) \ \mathsf{or}\ \ \ell \mapsto \mathsf{true}\Big)\Big) \tag{6.2}$$

Here, we modeled the Rust type `T` as a predicate $T\colon \mathit{Loc} \to \mathit{nProp}$. We simply store the borrower token inside the invariant. This direct approach was not possible with Iris's invariants and RustBelt's borrows, due to the problems around the *later modality* $\triangleright$, and thus RustBelt introduced a special borrow called the *atomic borrow* $\&^{\alpha/\mathcal{N}}_{\mathrm{at}} P$ as a workaround for that (recall the discussion in § 6.2).

To be precise, we should be able to *shorten the lifetime* α of the reference type `&α Mutex<T>`. The current model (6.2) does not support that, because the lifetime α is fixed inside the invariant connective. For that, we slightly relax the model as follows:

$$\exists \alpha'.\ \alpha \sqsubseteq \alpha' * \mathsf{inv}^{\mathcal{N}} \Big(\mathsf{bor}^{\alpha'} \Big(\big(\ell \mapsto \mathsf{false} * T\,(\ell+1)\big) \ \mathsf{or}\ \ \ell \mapsto \mathsf{true}\Big)\Big) \tag{6.3}$$

Here, the lifetime α' represents the 'actual' lifetime of the borrow.

Furthermore, if we want to make sure that the lender retrieves the content $T\ (\ell+1)$ after the lifetime α' ends regardless of the flag at ℓ, we modify the model (6.3) so that the content $T\ (\ell+1)$ is under the borrow:

$$\exists \alpha'.\ \alpha \sqsubseteq \alpha' * \mathsf{inv}^{\mathcal{N}} \Big(\mathsf{bor}^{\alpha'} \Big(\big(\ell \mapsto \mathsf{false} * \mathsf{bor}^{\alpha'}(T\,(\ell+1))\big) \ \mathsf{or}\ \ \ell \mapsto \mathsf{true}\Big)\Big) \tag{6.4}$$

This model can be seen as a Nola version of RustBelt's model for the shared reference type to a mutex-guarded object `&α Mutex<T>` (Jung, 2020, § 13.2). Let us name this Iris assertion (6.4) as $\mathsf{refmutex}^{\alpha,\mathcal{N}}_{\ell}\ T$.

By storing the content $[\![T\ (\ell+1)]\!]$ with the ownership of the flag set to false, we can create the shared reference to the mutex-guarded object $\mathsf{refmutex}^{\alpha,\mathcal{N}}_{\ell}\ T$ with a promise to retrieve the content $[\![T\ (\ell + 1)]\!]$ with the ownership of the flag after the lifetime α ends:

$$[\alpha]_q \ast [\![T\,(\ell+1)]\!] \ast \ell \mapsto b \ \vDash \Rrightarrow^{\mathsf{Winv}\,[\![\,]\!] \,\ast\, \mathsf{Wbor}_{\Rrightarrow}\,[\![\,]\!]}_{\mathcal{N}} \Big(\, [\alpha]_q \ast \mathsf{refmutex}^{\alpha,\mathcal{N}}_{\ell}\, T \ast \mathsf{lend}^{\alpha}\,\big(\, T\,(\ell+1) \ast \exists\, b.\ \ell \mathrel{\text{|->}} b \,\big)\Big)$$

Here, we use the custom world satisfaction $\mathsf{Winv}\,[\![\,]\!] \ast \mathsf{Wbor}_{\Rrightarrow}\,[\![\,]\!]$, which consists of the world satisfactions for Nola's invariant and borrow mechanisms. To prove this, we first create the lender $\mathsf{lend}^{\alpha}\,\big(\, T\,(\ell+1) \ast \exists\, b.\ \ell \mathrel{\text{|->}} b \,\big)$ and the borrower by BOR-LEND-NEW, storing $[\![T\ (\ell + 1)]\!]$ and $\ell \mapsto b$. Then we split the borrower into the component borrowers $\mathsf{bor}^{\alpha}\,(T\,(\ell+1))$ and $\mathsf{bor}^{\alpha}\,(\exists\, b.\ \ell \mathrel{\text{|->}} b)$ by BOR-OPEN and OBOR-SUBDIV (with the help of the live lifetime token $[\alpha]_q$). Then we apply BOR-OPEN and OBOR-SUBDIV again to turn the second borrower $\mathsf{bor}^{\alpha}\,(\exists\, b.\ \ell \mathrel{\text{|->}} b)$ into

$$\mathsf{bor}^{\alpha}\,\Big(\,\big(\,\ell \mathrel{\text{|->}} \mathsf{false} \ast \mathsf{bor}^{\alpha}(T\,(\ell+1))\,\big) \ \text{or}\ \ \ell \mathrel{\text{|->}} \mathsf{true}\,\Big),$$

by storing the first borrower $\mathsf{bor}^{\alpha}\,(T\,(\ell+1))$ for the new content. Finally, we store the resulting enriched borrower token to create the invariant for $\mathsf{refmutex}^{\alpha,\mathcal{N}}_{\ell}\ T$ by INV-ALLOC. The promise is created from the lender tokens for the inner and outer borrows, using LEND-RETRIEVE.

With a shared reference to the mutex-guarded object $\mathsf{refmutex}^{\alpha,\mathcal{N}}_{\ell}\ T$, a thread can try to acquire the mutex lock with *compare-and-swap* cas, and when it succeeds, the content $[\![T\ (\ell + 1)]\!]$ is transferred to the thread:

$$\Big[\, [\alpha]_q \ast \mathsf{refmutex}^{\alpha,\mathcal{N}}_{\ell}\ T \,\Big]\ \ \mathsf{cas}(\ell, \mathsf{false}, \mathsf{true})$$
$$\Big[\, \lambda v.\ \ [\alpha]_q \ast \Big(\, v = \mathsf{false}\ \lor\ \big(\, v = \mathsf{true} \ast [\![T\,(\ell+1)]\!]\,\big)\Big)\Big]^{\mathsf{Winv}\,[\![\,]\!] \,\ast\, \mathsf{Wbor}_{\Rrightarrow}\,[\![\,]\!]}_{\mathcal{N}}$$

Again, we use the custom world satisfaction $\mathsf{Winv}\,[\![\,]\!] \ast \mathsf{Wbor}_{\Rrightarrow}\,[\![\,]\!]$ for Nola's invariant and borrow mechanisms. We use a live lifetime token $[\alpha]_q$, observing that the lifetime α is alive, to get access to the content of the borrows.

Also, under $\mathsf{refmutex}^{\alpha,\mathcal{N}}_{\ell}\ T$, the thread can release the lock by storing back the content $[\![T\ (\ell + 1)]\!]$:

$$\Big[\, [\alpha]_q \ast \mathsf{refmutex}^{\alpha,\mathcal{N}}_{\ell}\ T \ast [\![T\,(\ell+1)]\!] \,\Big]\ \ \ell \leftarrow \mathsf{false}\ \Big[\, \lambda\_.\ [\alpha]_q \,\Big]^{\mathsf{Winv}\,[\![\,]\!] \,\ast\, \mathsf{Wbor}_{\Rrightarrow}\,[\![\,]\!]}_{\mathcal{N}}$$

Notably, the access to this shared reference is completely *free of the later modality* $\triangleright$, enabling further access to the content even under the total Hoare triple.

Usual Shared Reference Using Nola's borrows and invariants, we can express usual shared reference types in Rust like &α `int` and &α (`int`, `bool`) without difficulty. For this, we can introduce the *shared points-to token* $\ell \overset{\alpha}{\mapsto} v \in iProp$, defined as follows:

$$\ell \overset{\alpha}{\mapsto} v \quad \triangleq \quad \exists \alpha'.\ \alpha \sqsubseteq \alpha' \ast \mathsf{inv}^{\mathcal{N}}\Big(\exists\, q.\ \mathsf{bor}^{\alpha'}\,\big(\,\ell \overset{q}{\mathrel{\text{|->}}} v\,\big)\Big) \tag{6.5}$$

Here we choose some namespace $\mathcal{N}$ for the shared points-to token. This is an invariant that stores a borrower token $\mathsf{bor}^{\alpha'}\,\big(\,\ell \overset{q}{\mathrel{\text{|->}}} v\,\big)$ over a fractional points-to token $\ell \overset{q}{\mathrel{\text{|->}}} v$ of some fraction q. Notably, the shared points-to token $\ell \overset{\alpha}{\mapsto} v$ is *persistent*, unlike the usual points-to token $\ell \overset{q}{\mapsto} v$.

We can model an integer shared reference $\ell\colon$ &α `int` as follows using the shared points-to token:

$$\exists\, n \in \mathbb{Z}.\ \ \ell \overset{\alpha}{\mapsto} n$$

Similarly, we can model a shared reference to a pair of an integer and a boolean ℓ: `&α (int, bool)` as follows:

$$\exists\, n \in \mathbb{Z},\, b \in \mathbb{B}.\ \ \ell \overset{\alpha}{\mapsto} n \ * \ (\ell + 1) \overset{\alpha}{\mapsto} b$$

By storing a fractional points-to token $\ell \overset{q}{\mapsto} v$, we can create a shared points-to token $\ell \overset{\alpha}{\mapsto} v$ and a lender token $\mathsf{lend}^{\alpha}\,(\ell \overset{q}{\mathrel{\texttt{|->}}} v)$:

$$\ell \overset{q}{\mapsto} v \ \ \models \Rrightarrow^{\mathsf{Winv}[\![\ ]\!] \,*\, \mathsf{Wbor}_{\Rrightarrow}[\![\ ]\!]}_{\varnothing} \ \big(\, \ell \overset{\alpha}{\mapsto} v \ * \ \mathsf{lend}^{\alpha}\,(\ell \overset{q}{\mathrel{\texttt{|->}}} v)\,\big)$$

We first borrow $\ell \overset{q}{\mathrel{\texttt{|->}}} v$ by BOR-LEND-NEW to create a borrower token $\mathsf{bor}^{\alpha}\,(\ell \overset{q}{\mathrel{\texttt{|->}}} v)$ and a lender token $\mathsf{lend}^{\alpha}\,(\ell \overset{q}{\mathrel{\texttt{|->}}} v)$. Then we store the borrower token to create the invariant for the shared points-to token $\ell \overset{\alpha}{\mapsto} v$ by INV-ALLOC.

Under the shared points-to token $\ell \overset{\alpha}{\mapsto} v$, we can always *take out a borrower token* for a points-to token $\ell \overset{q'}{\mathrel{\texttt{|->}}} v$ of some fraction q':

$$[\alpha]_r \ * \ \ell \overset{\alpha}{\mapsto} v \ \ \models \dot{\Rrightarrow}^{\mathsf{Winv}[\![\ ]\!] \,*\, \mathsf{Wbor}_{\Rrightarrow}[\![\ ]\!]}_{\mathcal{N}} \ \big(\, [\alpha]_r \ * \ \exists q'.\ \mathsf{bor}^{\alpha}\,(\ell \overset{q'}{\mathrel{\texttt{|->}}} v)\,\big)$$

To prove this, we first get access to the invariant's content by INV-ACC, which gives a borrower token $\mathsf{bor}^{\alpha'}\,\big(\ell \overset{q}{\mathrel{\texttt{|->}}} v\big)$ for some q and some lifetime α' that is no shorter than α. Then we *subdivide* this borrow by BOR-OPEN and OBOR-SUBDIV to create *two* borrower tokens $\mathsf{bor}^{\alpha'}\,\big(\ell \overset{q/2}{\mathrel{\texttt{|->}}} v\big)$, $\mathsf{bor}^{\alpha'}\,\big(\ell \overset{q/2}{\mathrel{\texttt{|->}}} v\big)$ for the half $q/2$ of the original fraction q. We can restore the invariant's content using one of the two borrower tokens and can output the other.

Using a borrower token $\mathsf{bor}^{\alpha}\,(\ell \overset{q}{\mathrel{\texttt{|->}}} v)$ over a fractional points-to token, we can read from the location ℓ:[5]

$$\big[\, [\alpha]_r \ * \ \mathsf{bor}^{\alpha}\,(\ell \overset{q}{\mathrel{\texttt{|->}}} v)\,\big] \ \ !\ell \ \ \big[\, \lambda v'.\ v' = v \ * \ [\alpha]_r\,\big]^{\mathsf{Winv}[\![\ ]\!] \,*\, \mathsf{Wbor}_{\Rrightarrow}[\![\ ]\!]}_{\mathcal{N}}$$

Extending the idea of the shared points-to token (6.5), we can generally express the *fractured borrower token* $\mathsf{fbor}^{\alpha}\ T$ for any fractional predicate $T\colon \mathbb{Q} \longrightarrow nProp$ instead of just the points-to predicate $\lambda q.\, \ell \overset{q}{\mapsto} v$, just like the *fractured borrow* $\&^{\alpha}_{\mathrm{frac}}\,\Phi$ provided by RustBelt's lifetime logic.

6.4 Semantic Alteration by Derivability

We can support *semantic alteration* of the proposition P of the borrower, lender and open borrower connectives by the general *derivability* construction presented in Chapter 4.

First, we add to the judgment data type *Judg* the following constructor:

$$\mathit{Judg} \ni J \ ::= \ \cdots \ \mid \ P \Rightarrow Q$$

The judgment $P \Rightarrow Q$ simply means that the proposition P can be converted into the proposition Q.

Next, we construct the *interpretation* $[\![\ ]\!]_{\delta}\colon nProp \longrightarrow iProp$ *parameterized* with the derivability predicate $\delta\colon \mathit{Judg} \longrightarrow iProp$ (§ 4.2). The syntactic borrower, lender and open borrower assertions $\mathsf{bor}^{\alpha}\,P$, $\mathsf{lend}^{\alpha}\,P$, $\mathsf{obor}^{\alpha}\,P \in nProp$ can be interpreted as follows:

$$[\![\mathsf{bor}^{\alpha}\,P]\!]_{\delta} \quad \triangleq \quad \exists Q.\ \delta\,(P \Rightarrow Q) \ * \ \delta\,(Q \Rightarrow P) \ * \ \mathsf{bor}^{\alpha}\,Q$$

[5] Although we consider here an atomic read, we can also similarly perform a *non-atomic* read $!^{\mathsf{na}}\ell$ using the borrower token $\mathsf{bor}^{\alpha}\,(\ell \overset{q}{\mathrel{\texttt{|->}}} v)$.

$$\begin{aligned}
[\![\mathsf{lend}^{\alpha}\, P]\!]_{\delta} &\triangleq \exists Q.\ \delta\,(Q \Rightarrow P) \,*\, \mathsf{lend}^{\alpha}\, Q \\
[\![\mathsf{obor}^{\alpha}_{q}\, P]\!]_{\delta} &\triangleq \exists Q.\ \delta\,(P \Rightarrow Q) \,*\, \mathsf{obor}^{\alpha}_{q}\, Q
\end{aligned}$$

Unlike the model of the invariant (4.8) (§ 4.3), we do not need the persistence modality $\square$ on the derivability assertions $\delta\,(P \Rightarrow Q)$ and $\delta\,(Q \Rightarrow P)$, because the borrower, lender and open borrower assertions are not persistent.

Now we can define the semantics $[\![\ ]\!]^{+}_{\delta}\colon \mathit{Judg} \longrightarrow \mathit{iProp}$ of judgments *Judg parameterized* with the derivability predicate $\delta\colon \mathit{Judg} \longrightarrow \mathit{iProp}$. For the judgment $P \Rightarrow Q$, we define the semantics as follows:

$$[\![P \Rightarrow Q]\!]^{+}_{\delta} \quad\triangleq\quad [\![P]\!]_{\delta} \mathrel{-\!\!*} \dot{\Rrightarrow}\, [\![Q]\!]_{\delta}.$$

Now we automatically get the *best derivability predicate* $\mathsf{der} \in \mathit{Judg} \longrightarrow \mathit{iProp}$ and the set of *good derivability predicates* $\mathit{Deriv} \subseteq \mathit{Judg} \longrightarrow \mathit{iProp}$ by Definition 4.1 (§ 4.3).

Now the borrower, lender and open borrower connectives support the following rules for semantic alteration by derivability, which hold for any good derivability predicate $\delta \in \mathit{Deriv}$, in a similar way to inv-ALTER (§ 4.3):

$$\begin{aligned}
\delta\,(P \Rightarrow Q) \,*\, \delta\,(Q \Rightarrow P) \,*\, [\![\mathsf{bor}^{\alpha}\, P]\!]_{\delta} &\vDash [\![\mathsf{bor}^{\alpha}\, Q]\!]_{\delta} \quad \text{bor-ALTER} \\
\delta\,(P \Rightarrow Q) \,*\, [\![\mathsf{lend}^{\alpha}\, P]\!]_{\delta} &\vDash [\![\mathsf{lend}^{\alpha}\, Q]\!]_{\delta} \quad \text{lend-ALTER} \\
\delta\,(Q \Rightarrow P) \,*\, [\![\mathsf{obor}^{\alpha}_{q}\, P]\!]_{\delta} &\vDash [\![\mathsf{obor}^{\alpha}_{q}\, Q]\!]_{\delta} \quad \text{obor-ALTER}
\end{aligned}$$

For a simple example, using bor-ALTER we can prove

$$[\![\mathsf{bor}^{\alpha}(P * Q)]\!]_{\delta} \;=\; [\![\mathsf{bor}^{\alpha}(Q * P)]\!]_{\delta}$$

for any $\delta \in \mathit{Deriv}$, just like (4.19).

Also, the following basic conversion rules hold for any δ:

$$\begin{aligned}
\beta \sqsubseteq \alpha \,*\, [\![\mathsf{bor}^{\alpha}\, P]\!]_{\delta} &\vDash [\![\mathsf{bor}^{\alpha}\, P]\!]_{\delta} \quad \text{bor-LFT} \\
\alpha \sqsubseteq \beta \,*\, [\![\mathsf{lend}^{\alpha}\, P]\!]_{\delta} &\vDash [\![\mathsf{lend}^{\beta}\, P]\!]_{\delta} \quad \text{lend-LFT} \\
\beta \sqsubseteq \alpha \,*\, ([\alpha]_{q} \mathrel{-\!\!*} [\beta]_{r}) \,*\, [\![\mathsf{obor}^{\alpha}_{q}\, P]\!]_{\delta} &\vDash [\![\mathsf{obor}^{\beta}_{r}\, P]\!]_{\delta} \quad \text{obor-LFT} \\
\dagger\alpha &\vDash [\![\mathsf{bor}^{\alpha}\, P]\!]_{\delta} \quad \text{bor-FAKE}
\end{aligned}$$

These semantically alterable connectives satisfy the following proof rules for manipulating borrows, just like those presented in § 6.3.2. Here, we consider the interpretation $[\![\ ]\!] \triangleq [\![\ ]\!]_{\mathsf{der}}$ by the best derivability predicate der. These rules are easily derived from the original rules in § 6.3.2, especially using the *soundness* of der (Theorem 4.3).

$$\begin{aligned}
[\![P]\!] &\vDash \dot{\Rrightarrow}^{\mathrm{Wbor}_{M}\,[\![\ ]\!]}\, \big([\![\mathsf{bor}^{\alpha}\, P]\!] \,*\, [\![\mathsf{lend}^{\alpha}\, P]\!]\big) \quad \text{bor-lend-NEW} \\
\dagger\alpha \,*\, [\![\mathsf{lend}^{\alpha}\, P]\!] &\vDash M^{\mathrm{Wbor}_{M}\,[\![\ ]\!]}\, [\![P]\!] \quad \text{lend-RETRIEVE} \\
[\alpha]_{q} \,*\, [\![\mathsf{bor}^{\alpha}\, P]\!] &\vDash M^{\mathrm{Wbor}_{M}\,[\![\ ]\!]}\, \big([\![\mathsf{obor}^{\alpha}_{q}\, P]\!] \,*\, [\![P]\!]\big) \quad \text{bor-OPEN} \\
[\![\mathsf{obor}^{\alpha}_{q}\, P]\!] \,*\, [\![P]\!] &\vDash \dot{\Rrightarrow}^{\mathrm{Wbor}_{M}\,[\![\ ]\!]}\, \big([\alpha]_{q} \,*\, [\![\mathsf{bor}^{\alpha}\, P]\!]\big) \quad \text{obor-CLOSE}
\end{aligned}$$

$$\begin{aligned}
\beta \sqsubseteq \alpha \,*\, [\![\mathsf{obor}^{\alpha}_{q}\, P]\!] \,*\, \mathop{\text{\Large$*$}}_{i} [\![Q_i]\!] \,*\, \Big(\dagger\beta \,*\, \mathop{\text{\Large$*$}}_{i} [\![Q_i]\!] \mathrel{-\!\!*} M\, [\![P]\!]\Big) & \\
\vDash \dot{\Rrightarrow}^{\mathrm{Wbor}_{M}\,[\![\ ]\!]}\, \Big([\alpha]_{q} \,*\, \mathop{\text{\Large$*$}}_{i} [\![\mathsf{bor}^{\beta}\, Q_i]\!]\Big) & \quad \text{obor-SUBDIV}
\end{aligned}$$

$$\mathop{\ast}_{j} \big(\beta \sqsubseteq \alpha_j \ast [\![\mathsf{obor}^{\alpha_j}_{q_j} P_j]\!] \big) \ast \mathop{\ast}_{i} [\![Q_i]\!] \ast \Big(\dagger\beta \ast \mathop{\ast}_{i} [\![Q_i]\!] \mathrel{-\!\!\ast} M \big(\mathop{\ast}_{j} [\![P_j]\!] \big) \Big) \\ \vDash \dot{\Rrightarrow}^{\mathrm{Wbor}_M [\![\,]\!]} \Big(\mathop{\ast}_{j} [\alpha_j]_{q_j} \ast \mathop{\ast}_{i} [\![\mathsf{bor}^{\beta}\, Q_i]\!] \Big) \qquad \mathsf{obor}\text{-MERGE-SUBDIV}$$

$$[\alpha]_q \ast [\![\mathsf{bor}^{\alpha}\, P]\!] \vDash M^{\mathrm{Wbor}_M [\![\,]\!]} \big([\alpha]_q \ast [\![\mathsf{bor}^{\alpha \sqcap \beta}\, P]\!] \ast (\dagger\beta \mathrel{-\!\!\ast} [\![\mathsf{bor}^{\alpha}\, P]\!]) \big) \qquad \mathsf{bor}\text{-REBORROW}$$

$$[\![\mathsf{lend}^{\alpha}\, P]\!] \ast \Big([\![P]\!] \mathrel{-\!\!\ast} M \big(\mathop{\ast}_{i} [\![Q_i]\!] \big) \Big) \vDash \dot{\Rrightarrow}^{\mathrm{Wbor}_M [\![\,]\!]} \Big(\mathop{\ast}_{i} [\![\mathsf{lend}^{\alpha}\, Q_i]\!] \Big) \qquad \mathsf{lend}\text{-SPLIT}$$

6.5 Model

Finally, we briefly explain the model of the lifetime and borrow mechanisms. Although we inherit the high-level ideas (e.g., each borrower having the state closed, open, or reborrowing) from RustBelt's lifetime logic, our model is substantially different from theirs. In particular, thanks to being free of the problem of step-indexing, we have a much more direct model of borrows, while RustBelt needed to use an external 'Box' library. Also, our ghost state for borrows behaves independently of the ghost state for lifetimes, unlike RustBelt. Now we dive into the details.

6.5.1 Lifetime Mechanism

Lifetime The *lifetime* $\alpha, \beta \in \mathit{Lft}$ is modeled as a *finite multiset* of ghost names, just like RustBelt's lifetime logic:

$$\mathit{Lft} \quad \triangleq \quad \mathrm{Multiset}_{\mathrm{fin}}\ \mathit{GhostName}.$$

We use the finite multiset to support the static lifetime and lifetime intersection. The static lifetime is defined as the empty multiset $\top \triangleq \varnothing$ and the lifetime intersection is defined as the multiset sum (adding the multiplicities) $\alpha \sqcap \beta \triangleq \alpha \uplus \beta$.

We also introduce *pure lifetime inclusion* $\dot{\sqsubseteq} : \mathit{Lft} \times \mathit{Lft} \to \mathit{Prop}$, which is a pure relation defined as reverse multiset inclusion $\alpha \mathrel{\dot{\sqsubseteq}} \beta \triangleq \alpha \supseteq \beta$.

Resource Algebra The *resource algebra* LFT for lifetimes is defined as follows:

$$\textsc{Lft} \quad \triangleq \quad \textsc{Dfrac} +_{\lightning} \textsc{Unit}.$$

It is the sum RA of the discardable fraction RA DFRAC and the unit RA UNIT (§ 2.2.2), respectively modeling the live and dead state of an atomic lifetime $\gamma \in \mathit{GhostName}$. This is similar to RustBelt's lifetime logic, but we use the discardable fraction RA instead of the fraction RA to support the eternal lifetime token $\infty\alpha$ we newly introduced.

Propositions Now the *live* $[\alpha]_q$, *dead* $\dagger\alpha$ and *eternal* $\infty\alpha$ lifetime tokens are modeled as follows:

$$[\alpha]_q \quad \triangleq \quad \mathop{\ast}_{\gamma \in \alpha} \boxed{\mathsf{inl}\, q}^{\gamma}_{\textsc{Lft}} \qquad\qquad \dagger\alpha \quad \triangleq \quad \exists\, \gamma \in \alpha.\ \boxed{\mathsf{inr}\, ()}^{\gamma}_{\textsc{Lft}}$$

$$\infty\alpha \quad \triangleq \quad \mathop{\ast}_{\gamma \in \alpha} \boxed{\mathsf{inl}\, \star}^{\gamma}_{\textsc{Lft}}$$

For the live lifetime token $[\alpha]_q$, we use $\mathsf{inl}\, q$, a fraction resource $q \in \mathbb{Q}_{>0}$ in the left-hand summand of LFT. For the dead lifetime token $\dagger\alpha$, we use $\mathsf{inr}\, ()$, the resource in the right-hand summand. For the eternal lifetime token $\infty\alpha$, we use $\mathsf{inl}\, \star$, the discard witness in the left-hand summand. We use the resource update DFRAC-RESTORE (§ 2.2.2) to prove the rule LFT-ETERN-LIVE. The iterative separating conjunction $\ast_{\gamma \in \alpha}$ takes into account the multiplicity of each ghost name γ in the multiset α, which is important for the rule LFT-LIVE-$\sqcap$.

The *lifetime inclusion* $\alpha \sqsubseteq \beta \in iProp$ is modeled as follows:

$$\alpha \sqsubseteq \beta \quad \triangleq \quad \dagger\alpha \;\vee\; \left(\exists \gamma \text{ s.t. } \alpha \sqcap \gamma \mathrel{\dot{\sqsubseteq}} \beta.\; \infty\gamma\right)$$

It is either a dead lifetime token $\dagger\alpha$ (to support LFT-DEAD-$\sqsubseteq$) or an eternal lifetime token $\infty\gamma$ for some γ that satisfies the pure lifetime inclusion $\alpha \sqcap \gamma \mathrel{\dot{\sqsubseteq}} \beta \in Prop$ (to support LFT-ETERN-$\sqsubseteq$). This model of lifetime inclusion differs from that of RustBelt's lifetime logic, which modeled lifetime inclusion based on the rules LFT-$\sqsubseteq$-LIVE-ACC and LFT-$\sqsubseteq$-DEAD. We adopted a more direct model here, especially to make lifetime inclusion *timeless* (LFT-$\sqsubseteq$-TIMELESS).

6.5.2 Borrow Mechanism

Resource Algebra The *resource algebra for the borrow mechanism* $\textsc{Bor}_{nProp}$ is defined as follows:

$$\begin{aligned}
\textsc{Bor}_{nProp} \quad &\triangleq \quad \textsc{Auth}\left(DepoId \xrightarrow{\text{fin}} \textsc{Depo}_{nProp}\right) \\
\textsc{Depo}_{nProp} \quad &\triangleq \quad \textsc{Ag}\,(Lft \times \mathbb{N}) \times \textsc{Borr}_{nProp} \times \textsc{Lend}_{nProp} \\
\textsc{Borr}_{nProp} \quad &\triangleq \quad BorrId \xrightarrow{\text{fin}} \text{Ex}\left(nProp \times BorrMode\right) \\
BorrMode \ni b \quad &::= \quad \text{closed} \mid \text{open}_q \mid \text{reborrow}_\beta \\
\textsc{Lend}_{nProp} \quad &\triangleq \quad LendId \xrightarrow{\text{fin}} \text{Ex}\, nProp
\end{aligned}$$

The resource algebra $\textsc{Bor}_{nProp}$ is the authoritative RA (§ 2.2.2) over a finite map RA to *deposits* $\textsc{Depo}_{nProp}$, to which borrowers $\textsc{Borr}_{nProp}$ and lenders $\textsc{Lend}_{nProp}$ are linked.

Each deposit is identified with a deposit id $i \in DepoId$ and associated with a fixed lifetime $\alpha \in Lft$ and a fixed depth $d \in \mathbb{N}$ in the hierarchy of reborrows. When a deposit *reborrows* from another deposit, the former's depth should be strictly larger than the latter's.

Each *borrower* is identified with a borrower id $j \in BorrId$ and associated with the proposition $nProp$ and the mode $BorrMode$, which is closed, open with a live lifetime token $[\alpha]_q$ stored, or reborrowing under the lifetime β. The idea of using these three modes comes from RustBelt's lifetime logic.

Each *lender* is identified with a lender id $k \in LendId$ and associated just with the proposition $nProp$.

The structure of resources is quite different from what RustBelt's lifetime logic used. In RustBelt's lifetime logic, borrows are managed per *lifetime*. This makes lifetimes tightly coupled with borrows. Also, their proof around reborrows depends on a subtle property about lifetimes: from any non-empty set of lifetimes, a lifetime that is minimal with respect to pure lifetime inclusion $\dot{\sqsubseteq}$ can be taken. Our proof is more robust, just using the acyclicity ensured by the *depth* information, not assuming any special structures about lifetimes.

Tokens The *lender token* $\text{lend}^{\alpha} P$ is modeled as follows:

$$\text{lend}^{\alpha} P \quad \triangleq \quad \exists i, \alpha', d, k.\;\; \alpha' \sqsubseteq \alpha \;\ast\; \boxed{\circ\left[\, i := \left(\text{ag}\,(\alpha', d),\; \varnothing,\; [k := \text{ex}\, P]\right)\right]}^{\gamma_{\textsc{Bor}}}$$

It is existentially quantified over the deposit id i, the deposit's lifetime α' and depth d, and the lender id k. Lifetime inclusion $\alpha' \sqsubseteq \alpha$ ensures that $\dagger\alpha$ entails $\dagger\alpha'$ (LFT-⊑-DEAD) for the rule LEND-RETRIEVE.

The *open borrower token* $\mathrm{obor}^{\alpha}_{q}\, P$ is modeled as follows:

$$\mathrm{obor}^{\alpha}_{q}\, P \;\triangleq\; \exists i, \alpha', d, j, r.\;\; \alpha \sqsubseteq \alpha' \;*\; ([\alpha']_r \twoheadrightarrow [\alpha]_q) \;*\; [\alpha']_{r/2} \;* \\ \boxed{\circ \big[\, i := \big(\mathrm{ag}\,(\alpha', d),\ [j := \mathrm{ex}\,(P, \mathrm{open}_{r/2})],\ \varnothing\,\big)\,\big]}^{\gamma_{\mathrm{Bor}}}$$

It is existentially quantified over the deposit id i, the deposit's lifetime α' and depth d, the borrower id j, and the fraction $r \in \mathbb{Q}_{>0}$. The open borrower token stores only the half token $[\alpha']_{r/2}$ to the deposit, retaining the other half token $[\alpha']_{r/2}$.

The *borrower token* $\mathrm{bor}^{\alpha}\, P$ is modeled as follows:

$$\mathrm{bor}^{\alpha}\, P \;\triangleq\; \dagger\alpha \;\vee\; \exists i, \alpha', d, j.\;\; \alpha \sqsubseteq \alpha' \;* \\ \Big(\boxed{\circ \big[\, i := \big(\mathrm{ag}\,(\alpha', d),\ [j := \mathrm{ex}\,(P, \mathrm{closed})],\ \varnothing\,\big)\,\big]}^{\gamma_{\mathrm{Bor}}} \;\vee \\ \exists \beta.\; \dagger\beta \;*\; \boxed{\circ \big[\, i := \big(\mathrm{ag}\,(\alpha', d),\ [j := \mathrm{ex}\,(P, \mathrm{reborrow}_{\beta})],\ \varnothing\,\big)\,\big]}^{\gamma_{\mathrm{Bor}}}\Big)$$

The first disjunct $\dagger\alpha$ is just for the rule BOR-FAKE. The main part is existentially quantified over the deposit id i, the deposit's lifetime α' and depth d, and the borrower id j. There are two cases: the borrower is either closed closed or reborrowed $\mathrm{reborrow}_{\beta}$ under a *dead* lifetime β.

World Satisfaction The *borrower world satisfaction* $\mathrm{Wborr}^{(P,b)}_{\alpha,d}\,[\![\;]\!]$ for the deposit's lifetime $\alpha \in \mathit{Lft}$ and depth $d \in \mathbb{N}$, the proposition $P \in \mathit{nProp}$, the borrower mode $b \in \mathit{BorrMode}$, and the interpretation $[\![\;]\!]\colon \mathit{nProp} \to \mathit{iProp}$ is defined as follows:

$$\mathrm{Wborr}^{(P,b)}_{\alpha,d}\,[\![\;]\!] \;\triangleq\; \begin{cases} [\![P]\!] & b = \mathrm{closed} \\ [\alpha]_q & b = \mathrm{open}_q \\ \exists d' > d.\;\; \mathrm{lend}'^{\,\alpha\sqcap\beta}_{d'}\, P & b = \mathrm{reborrow}_{\beta} \end{cases}$$

For the closed state, it just stores the interpretation of the borrowed proposition $[\![P]\!]$. For the open state with the fraction q, it stores the live lifetime token $[\alpha]_q$. For the reborrow state under the lifetime β, it stores $\mathrm{lend}'^{\,\alpha\sqcap\beta}_{d'}\, P$ for some $d' > d$, where the modified lender token $\mathrm{lend}'^{\,\alpha}_{d}\, P$ is defined as follows:

$$\mathrm{lend}'^{\,\alpha}_{d}\, P \;\triangleq\; \exists i, k.\; \boxed{\circ \big[\, i := \big(\mathrm{ag}\,(\alpha, d),\ [k := \mathrm{ex}\; P],\ \varnothing\,\big)\,\big]}^{\gamma_{\mathrm{Bor}}}$$

When a reborrower $\mathrm{reborrow}_{\beta}$ opens the borrow again under the premise that $\dagger\beta$, it retrieves the content using the lender token $\mathrm{lend}'^{\,\alpha\sqcap\beta}_{d'}\, P$.

The set of the deposit states *Depo* is defined as follows:

$$\mathit{Depo} \;\triangleq\; (\mathit{Lft} \times \mathbb{N}) \times \big(\mathit{BorrId} \overset{\mathrm{fin}}{\rightharpoonup} \mathit{nProp} \times \mathit{BorrMode}\big) \times \big(\mathit{LendId} \overset{\mathrm{fin}}{\rightharpoonup} \mathit{nProp}\big)$$

The *deposit world satisfaction* $\mathrm{Wdepo}^{D}_{M}\,[\![\;]\!]$ for the deposit state $D \in \mathit{Depo}$, the update modality $M\colon \mathit{iProp} \to \mathit{iProp}$ and the interpretation $[\![\;]\!]\colon \mathit{nProp} \to \mathit{iProp}$ is defined as follows:

$$\mathrm{Wdepo}^{((\alpha,d),\mathit{Bs},\mathit{Ls})}_{M}\,[\![\;]\!] \;\triangleq\; \Big(\dagger\alpha \;*\; M\big(\mathop{\Large *}_{k \in \mathrm{dom}\,\mathit{Ls}} [\![\mathit{Ls}\,k]\!]\big)\Big) \;\vee \\ \Big(\mathop{\Large *}_{j \in \mathrm{dom}\,\mathit{Bs}} \mathrm{Wborr}^{\mathit{Bs}\,j}_{\alpha,d}\,[\![\;]\!] \;*\; \Big(\dagger\alpha \;*\; \mathop{\Large *}_{j \in \mathrm{dom}\,\mathit{Bs}} [\![(\mathit{Bs}\,j).1]\!] \twoheadrightarrow M\,\big(\mathop{\Large *}_{k \in \mathrm{dom}\,\mathit{Ls}} [\![\mathit{Ls}\,k]\!]\big)\Big)\Big)$$

The first disjunct is the case where the lifetime of the deposit has been dead and the lenders have been retrieved. The second disjunct is the case where the deposit is alive. The disjunct stores the world satisfaction $\mathrm{Wborr}_{\alpha,d}^{Bs\,j}\,[\![\ ]\!]$ for each borrower. Also, it owns the 'converter' from the borrowers' propositions $\mathop{\ast}_{j\in\mathrm{dom}\,Bs}[\![(Bs\,j).1]\!]$ to the lenders' propositions $\mathop{\ast}_{k\in\mathrm{dom}\,Ls}[\![Ls\,k]\!]$ with the update by M assuming $\dagger\alpha$. This converter is used when a lender is retrieved (LEND-RETRIEVE).

We define the authoritative token $\mathrm{depos}\,Ds \in \mathit{iProp}$ for a finite map $Ds\colon \mathit{DepoId} \xrightarrow{\text{fin}} \mathit{Depo}$ as follows:

$$\mathrm{depos}\,Ds \quad \triangleq \quad \boxed{\bullet\ \mathrm{map}\ \Big(\lambda\big((\alpha, d), Bs, Ls\big).\ \big(\mathrm{ag}\,(\alpha, d),\ \mathrm{map}\ (\mathrm{ex})\ Bs,\ \mathrm{map}\ (\mathrm{ex})\ Ls\,\big)\Big)\ Ds}^{\,\gamma_{\mathrm{Bor}}}$$

Finally, the *world satisfaction* $\mathrm{Wbor}_M\,[\![\ ]\!]$ for the borrow mechanism is defined as follows:

$$\mathrm{Wbor}_M\,[\![\ ]\!] \quad \triangleq \quad \exists Ds.\ \ \mathrm{depos}\,Ds \ast \mathop{\ast}_{i\in\mathrm{dom}\,Ds} \mathrm{Wdepo}_M^{Ds\,i}\,[\![\ ]\!]$$

Chapter 7

Later-Free Prophetic Borrows

The best way to predict the future is to invent it

Alan Kay, at a 1971 meeting of PARC

This chapter presents the later-free *prophetic borrow* mechanism of our framework, built on our later-free borrow mechanism presented in Chapter 6. Our prophetic borrow mechanism abstracts and refines RustHornBelt (Matsushita et al., 2022)'s reasoning approach to functional verification about Rust-style borrows with *prophecies* in the style of RustHorn (Matsushita et al., 2020, 2021).

This is an interesting and useful *metatheory* on its own, providing general later-free proof rules for reasoning in the style of RustHornBelt but without directly manipulating the mechanism for prophetic agreement. From another perspective, this provides a *case study* of the expressivity of our framework, building a richer mechanism on a more basic one.

This chapter is organized as follows. Section 7.1 reviews the author's prior work, RustHorn and RustBelt, as the background of Nola's prophetic borrows. Section 7.2 presents the proof rules of our prophetic borrows. Section 7.3 discusses semantic alteration by the derivability technique of Chapter 4. Section 7.4 explains the model of our prophetic borrows.

7.1 Background — The Author's Prior Work

In this section, we review the author's prior work, RustHorn and RustBelt, as the background of Nola's prophetic borrows.

7.1.1 RustHorn: Prophecies for Rust-Style Borrows

The work RustHorn was originally the author's senior thesis (Matsushita, 2019) and then published in the conference proceedings of ESOP 2020 (Matsushita et al., 2020) and in a journal ACM TOPLAS (Matsushita et al., 2021). The core contributions of RustHorn, including the idea, the formalization, and the implementation and evaluation, are the author's.

RustHorn proposed the novel idea of using *prophecies* to model Rust-style (mutable) borrows and reason functionally about them. Using RustHorn's idea, mutable borrows can be naturally and uniformly translated into *first-class values* without state information, significantly simplifying the verification. Remarkably, this approach supports advanced borrowing patterns like reborrows and nested borrows.

The work RustHorn itself demonstrated the effectiveness of this idea in the context of fully automated verification. RustHorn also presented a paper proof of the translation's correctness (soundness and completeness) over a simplified core calculus modeling Rust. Notably, RustHorn's idea gave rise to Creusot (Denis et al., 2022), a leading semi-automated Rust verifier at present.

Hereafter in this subsection, we explain the work RustHorn and its idea in more detail.

Challenge: Functional Reasoning about Rust-Style Borrows Rust's borrowing machinery is powerful and useful for managing ownership, especially because borrowers can freely throw away ownership fragments at any time *without direct communication with their lender*, as discussed in § 6.1.1.

But the borrowing machinery of Rust's type system provides only *rough information* about mutable state under a borrow, in that the only guarantee about the borrowed object returned to the lender after the lifetime ends is that the object satisfies the *ownership type* T. Although this is effective for guaranteeing memory and thread *safety*, we often want to go further to verify *functional correctness*, which takes into account the input and output *values* of computation. Regarding a borrow, the lender should be able to know the *exact value* of the borrowed object at the end of the lifetime, reflecting mutations by the borrower. But this is not obvious exactly due to the advantage of Rust's borrowing: there is no direct communication between the borrowers and the lender. How can we functionally reason about mutation by Rust-style borrows?

RustHorn's Solution: Prophecies The author's prior work, *RustHorn* (Matsushita, 2019; Matsushita et al., 2020, 2021) solved this challenge naturally and uniformly. The key device is a *prophecy variable* (or *prophecy* in short) (Abadi and Lamport, 1988; Vafeiadis, 2008; Jung et al., 2020b), which peeks into some information about the *future* ahead of time in program verification.

RustHorn's idea can be roughly explained as follows. When a borrow starts, we introduce a *prophecy variable* x that represents the *future* value of the borrowed object at the end of the lifetime, i.e., the *final* value of the borrowed object during the borrow. We let the borrower and the lender *share* this prophecy variable x. We model a borrower as the borrowed object's current value a and the prophecy x for its final value. When the borrower mutates the borrowed object, we update the object's value a accordingly. When a borrower throws away the ownership, we *resolve* the prophecy x into the object's value a of the borrowed object at that point. For an advanced case where a borrower is subdivided, we create new prophecies $y_1, \ldots, y_n$ for the new borrowers and *partially* resolve the old prophecy x into an appropriate value that depends on the new prophecies $y_1, \ldots, y_n$. When the lifetime ends, we can model the object returned to the lender as the value of the prophecy x, which has been resolved into the final value of the borrowed object.

In a typical setting, a prophecy variable is modeled by *non-determinism*, branching over all possibilities of the future value x. When the prophecy x is resolved into a value a, we *assume* the equality $x = a$, i.e., cut off all branches of non-determinism that do not satisfy the equality $x = a$. With this approach, the behavior of a Rust program with mutable borrows can be reduced to the behavior of a stateless functional program, making functional verification much easier and more efficient.

How RustHorn-Style Prophecies Are Used RustHorn demonstrated the effectiveness of this prophecy-based approach explained above in the context of CHC-based verification, i.e., fully automated functional verification by reduction to constrained Horn clauses (Bjørner et al., 2015).

Later, Creusot (Denis et al., 2022) embodied RustHorn's approach in the context of SMT-driven semi-automated verification, i.e., verification by manual annotations of loop invariants etc. and automated SMT solving. Creusot translates Rust programs with annotations into stateless functional programs and verifies them in the Why3 platform (Filliâtre and Paskevich, 2013). Remarkably, using Creusot, a performative SAT solver written in Rust was verified functionally correct (Skotåm, 2022).

Moreover, RusSOL (Fiala et al., 2023) used RustHorn-style functional specifications to automatically synthesize Rust programs that satisfy Rust's ownership principles and the given specifications.

A combination of RustHorn's approach with fractional ownership has also been explored (Nakayama et al., 2024).

Simple Example: Choice from Mutable Borrows For a simple example, consider the following Rust program that features a choice from two mutable borrows depending on dynamic information:[1] [2]

```
fn max_mut<α>(p : &α mut int, q : &α mut int) -> &α mut int {
  if *p >= *q { p } else { q }
}
fn test() {
  let mut m : int = rand_int();  let mut n : int = rand_int();
  let r : &mut int = max_mut(&mut m, &mut n);  *r += 7;
  assert!(7 <= abs(m - n));
}
```

Code 7.1: Simple Verification Problem on Choice from Mutable Borrows

The function `max_mut` takes two integer mutable references `p`, `q : &α mut int` and returns the one that stores the larger value. The tricky thing is that the address of the returned reference depends on dynamic information about values. The goal is to verify the assertion of the test function `test`, which goes as follows. First, we create integer variables `m`, `n` initialized randomly. Next, we mutably borrow the two and call the function `max_mut` to get a mutable reference `r` to the larger one. Then, we increment the value stored at `r` by 7. Finally, we assert that the difference between `m` and `n` is no less than 7. This assertion is always true because we have incremented the larger of `m` and `n` by 7.

We want to verify Code 7.1 without any explicit model of heap memory. RustHorn's prophecies let us do this. We model an integer mutable reference `p` as a pair $(m, x) \in \mathbb{Z}^2$ of the current value $m \in \mathbb{Z}$ and the prophetic final value $x \in \mathbb{Z}$ at the address. Then the function `max_mut` is modeled as the following input-output relation (postcondition) $\mathit{MaxMut} \colon \mathbb{Z}^2 \to \mathbb{Z}^2 \to \mathbb{Z}^2 \to \mathit{Prop}$:[3]

$$\begin{aligned} &\mathit{MaxMut}\ (m, x)\ (n, y)\ r \quad \triangleq \\ &\quad \big(m \geq n \land r = (m, x) \land y = n\big) \lor \big((m < n \land r = (n, y) \land x = m)\big) \end{aligned} \tag{7.1}$$

In the case `p` is returned, the other mutable reference `q : &α mut int` throws away the ownership and thus *resolves* its *prophecy* y into n. A similar thing goes for the case `q` is returned.

[1] For simplicity, we suppose an unbounded integer type `int` instead of a bounded integer type like `i32`.

[2] The function `rand_int` non-deterministically outputs a random integer.

[3] To aid understanding, we present the direct definition of input-output relations, unlike RustHorn's paper. The work RustHorn itself represented Rust programs as constrained Horn clauses (CHCs), giving constraints on predicate *variables* for the input-output relations. The least solution to RustHorn's CHCs amounts to the direct input-output relations we present here.

Now the verification condition of the test function `test` can be expressed as follows, using *MaxMut* defined by (7.1):

$$\forall m, n.\ \forall x, y.\ \forall o, z \text{ s.t. } \mathit{MaxMut}\ (m, x)\ (n, y)\ (o, z).\\ z = o + 7\ \rightarrow\ 7 \le |m - n| \tag{7.2}$$

Because the mutable reference `r` loses ownership after performing the mutation `*r += 7`, its *prophecy* z is *resolved* into the value $o + 7$, where o is the value stored at `r` before the mutation. We can easily verify that the verification condition (7.2) is satisfied by case analysis on the cases $m \ge n$ and $m < n$.

Advanced Example: Subdividing a Mutable Borrow over a List For an advanced example, let us consider the following Rust program that features a mutable borrow over a list and its subdivision:[4] [5]

```
enum List { Nil, Cons(int, Box<List>) }  use List::*;
fn take_some<α>(p : &α mut List) -> &α mut int {
  match p { Nil => loop {},
    Cons(q, r) => if rand_bool() { q } else { take_some(r) }
  }
}
fn sum(p : &List) -> int {
  match p { Nil => 0, Cons(n, q) => n + sum(q) }
}
fn test() {
  let mut l : List = rand_list();  let s : int = sum(&l);
  let p : &mut int = take_some(&mut l);  *p += 1;
  assert!(sum(&l) == s + 1);
}
```

Code 7.2: Advanced Verification Problem on Subdividing a Mutable Borrow over a List

The *recursive* data type `List` represents a singly linked list of integers. The function `take_some` takes a mutable reference `p : &α mut List` to a list and returns a mutable reference `&α mut int` to some integer element in the list (or loops infinitely). Importantly, this function *subdivides* a mutable borrow over the list into a mutable borrow over an element. Also, this function is recursive, making the verification more challenging. The helper function `sum` just computes the sum of the elements of the list `p : &List`. We want to verify the assertion of the test function `test`, which goes as follows. First, it takes a random list `l` and remembers the sum `s` of its elements. Next, it mutably borrows the list `l` and subdivides it into a mutable borrow `p` over some element using the function `take_some`. Then, it increments the value stored at `p` by 1. Finally, it asserts that the sum of the elements of the list `l` has been incremented by 1, i.e., has become `s + 1`.

Notably, using RustHorn's idea, we can naturally model and verify Code 7.2. The input-output relation of the function `take_some` can be expressed as the *least solution* for the predicate variable *TakeSome*: $(\mathit{List}\ \mathbb{Z})^2 \longrightarrow \mathbb{Z}^2 \longrightarrow \mathit{Prop}$ to the following constraints, which are technically CHCs (constrained Horn clauses):

$$\mathit{TakeSome}\ \ ([\,],\ xs)\ \ r \quad \Longleftarrow \quad \bot \tag{7.3}$$

$$\mathit{TakeSome}\ \ (n : ns,\ x : xs)\ \ r \quad \Longleftarrow \quad r = (n, x)\ \wedge\ xs = ns \tag{7.4}$$

[4] The part `use List::*;` just allows writing `Nil` and `Cons` instead of `List::Nil` and `List::Cons`.
[5] The functions `rand_bool` and `rand_list` output a random boolean and a random list, respectively.

$$\mathit{TakeSome}\ (n : ns,\ x : xs)\ r \quad \Longleftarrow \quad x = n \land \mathit{TakeSome}\ (ns, xs)\ r \tag{7.5}$$

The relation $\Longleftarrow$ simply means the logical implication from the right to the left. Variables like r are universally quantified. Like *MaxMut* above (7.1), for the predicate *TakeSome*, the input and output mutable references are modeled as the pair of the current value and the prophecy for the final value. The constraint (7.3) represents the case when `*p` is the empty list `Nil`. The constraints (7.4) and (7.5) represent the case when `*p` is non-empty. First, the prophecy of the input reference is *partially resolved* into a cons value $x : xs$. In the branch where the mutable reference to the head element is returned, as described by (7.4), the prophecy of the tail list xs is resolved into the current tail ns and the return value is the cons of the current head n and the prophecy of the head x. In the branch where `take_some` is recursively called on the tail of the list, as described by (7.5), the prophecy of the head is resolved into the current head n and the recursive call takes the pair of the current tail and the prophecy of the tail.

The verification condition of the test function `test` can be expressed as follows, using the *least solution* $\mathit{TakeSome}_\mu$ to the constraints (7.3), (7.4) and (7.5):

$$\begin{aligned} &\forall ns.\ \forall xs.\ \forall m, y \text{ s.t. } \mathit{TakeSome}_\mu\ (ns, xs)\ (m, y). \\ &\qquad\qquad y = m + 1 \ \to\ \mathsf{sum}\, xs = \mathsf{sum}\, ns + 1 \end{aligned} \tag{7.6}$$

Here, $\mathsf{sum} \colon \mathit{List}\,\mathbb{Z} \longrightarrow \mathbb{Z}$ is the function that computes the sum of the elements of an integer list. The verification condition (7.6) can be easily verified because the predicate

$$\lambda(ns, xs).\ \lambda(m, y).\ y = m + 1 \ \to\ \mathsf{sum}\, xs = \mathsf{sum}\, ns + 1 \tag{7.7}$$

is a solution for *TakeSome* to the constraints (7.3), (7.4) and (7.5), which by definition is implied by the least solution $\mathit{TakeSome}_\mu$. In other words, the condition (7.7) found from (7.6) is an *inductive invariant* for the recursive constraints (7.3), (7.4) and (7.5).

Remarkably, in RustHorn's experimental evaluation (Matsushita et al., 2020, § 4.3), the verification of the Rust code corresponding to Code 7.2 was completed *instantly* (taking less than a second) in a *fully automated* way, using HoIce (Champion et al., 2018) as the backend CHC solver.

Modeling an Advanced API: Mutable Iterator Using RustHorn's idea, we can also precisely model advanced Rust APIs with mutable borrows. For an interesting example, here we consider the *mutable iterator* `IterMut<α,T>` over a vector. Its core behavior boils down to the following two methods:

```
fn iter_mut<α,T>(v : &α mut Vec<T>) -> IterMut<α,T>
fn next<α,T>(it : &mut IterMut<α,T>) -> Option<&α mut T>
```

Code 7.3: Core Methods of Rust's Mutable Iterator

The method `iter_mut` converts a mutable reference to a vector `v : &α mut Vec<T>` into a mutable iterator `IterMut<α,T>`. A mutable iterator `IterMut<α,T>` has the ownership of some subsequence of a vector borrowed under the lifetime `α`. The key operation on the mutable iterator is `next`, which performs one step of iteration. When the subsequence owned by the iterator is not empty, the method removes the subsequence's head element from the subsequence and returns a mutable reference to it. When the subsequence is empty, the method returns `None` to tell the end of the iteration. Also note that the input `it` of the method `next` is an example of a *nested mutable reference*, since it is a mutable reference to a mutable iterator.

Using RustHorn's idea, we can naturally model the mutable iterator `IterMut<α,T>` and its methods `iter_mut` and `next`.[6] Suppose an object of the type `T` is modeled as a

[6] This functional representation of the API was first explicitly proposed and formalized by the work RustHornBelt (Matsushita, 2021; Matsushita et al., 2022), introduced later in § 7.1.2.

value in a set A. A vector `Vec<T>` is modeled as a list $[a_1, \ldots, a_n] \in \mathit{List}\, A$. We model a mutable iterator `IterMut<α,T>` as a list of pairs $\left[(a_1, x_1), \ldots, (a_n, x_n)\right] \in \mathit{List}\ A^2$ such that the i-th element (a_i, x_i) represents a mutable reference to the i-th element of the subsequence, modeled as the pair of the current value and prophecy.

The method `iter_mut<α,T>` can be modeled as the following input-output relation $\mathit{IterMut} \colon (\mathit{List}\, A)^2 \to \mathit{List}\ A^2 \to \mathit{Prop}$:

$$\mathit{IterMut}\ \ (\mathit{as}, \mathit{xs})\ \ \mathit{axs} \quad \triangleq \quad |\mathit{xs}| = |\mathit{as}| \ \land\ \mathit{axs} = \mathsf{zip}\, \mathit{as}\, \mathit{xs}.$$

We *partially resolve* the prophecy for the final vector value xs, fixing its length $|\mathit{xs}| = |\mathit{as}|$. The output list axs is the *zipped* list $\mathsf{zip}\ \mathit{as}\ \mathit{xs}$ of the current values as and prophecies xs (for example, $\mathsf{zip}\ [a_1, a_2, a_3]\ [x_1, x_2, x_3] = \left[(a_1, x_1), (a_2, x_2), (a_3, x_3)\right]$).

Once we model the mutable iterator in this way, the method `next<α,T>` can be modeled straightforwardly as the following input-output relation $\mathit{Next} \colon \left(\mathit{List}\ A^2\right)^2 \to \mathit{Option}\ A^2 \to \mathit{Prop}$:

$$\begin{aligned}\mathit{Next}\ \ (\mathit{ax} : \mathit{axs}, \mathit{axs}')\ \ \mathit{oax} \quad &\triangleq \quad \mathit{axs}' = \mathit{axs} \ \land\ \mathit{oax} = \mathsf{some}\, \mathit{ax}\\ \mathit{Next}\ \ ([\,], \mathit{axs}')\ \ \mathit{oax} \quad &\triangleq \quad \mathit{axs}' = [\,] \ \land\ \mathit{oax} = \mathsf{none}.\end{aligned}$$

7.1.2 RustHornBelt: Semantic Foundation for Prophetic Borrows

The work RustHornBelt was originally the author's master's thesis (Matsushita, 2021) and then published in the conference proceedings of ACM PLDI 2022 (Matsushita et al., 2022) winning the distinguished paper award. Key technical ideas of this work, including the machinery of parametric prophecies, and a large part of the Coq mechanization are the author's. The work was born through the author's internship in the RustBelt/Iris team led by Derek Dreyer at the Max Planck Institute for Software Systems (MPI-SWS).

RustHornBelt extended RustBelt (Jung et al., 2018a) to provide a semantic foundation for RustHorn-style prophetic verification. Its central scientific contribution is a new technique, nicknamed *parametric prophecies*, to reason about prophecies flexibly in separation logic.

In this subsection, we explain the work RustHornBelt in more detail. We also discuss the problems of its approach, which we tackle by Nola's later-free prophetic borrows.

Challenge: Verify RustHorn-Style Prophecies Generally As discussed above in § 7.1.1, RustHorn's prophetic approach to functional verification can naturally reason about various patterns of mutable borrows, including reborrows and nested borrows. However, proving the soundness of RustHorn's approach in general is challenging. Rust-style borrows and prophecies are even separately challenging to reason about, and RustHorn's approach combines them together in a highly non-trivial way.

The work RustHorn (Matsushita et al., 2021) presented a paper proof of the correctness of its prophetic translation. However, their proof only covered a simplified core calculus modeling Rust and depended on a subtle syntactic bisimulation argument. It was not clear how to extend their proof in general, especially in the presence of various Rust APIs.

RustHornBelt: RustBelt Extended for Prophetic Borrows The work RustHornBelt (Matsushita, 2021; Matsushita et al., 2022) extended RustBelt (Jung et al., 2018a) (reviewed in § 6.1.2) to provide a semantic foundation for functional verification of Rust programs in the style of RustHorn, beyond the memory and thread safety check of Rust's ownership type system. More specifically, RustHornBelt verified the soundness of RustHorn-style functional specifications over a large subset of Rust with various Rust APIs, including `Vec`, `SmallVec`, `IterMut`, `Cell`, `spawn/join`, and `Mutex`. For some of these APIs, the work RustHornBelt also newly proposed RustHorn-style specifications themselves.

The key aspect is that RustHornBelt semantically modeled various Rust APIs in the separation logic Iris, inheriting RustBelt's approach. This enables verifying the soundness of the ownership type interface and the RustHorn-style functional specification of each API separately in a modular way. Also, the definitions and proofs of RustHornBelt are fully mechanized in the Coq proof assistant.

As the central scientific contribution, RustHornBelt invented a new technique, nicknamed *parametric prophecies*, to reason about prophecies flexibly in separation logic. This machinery is explained more in detail later in § 7.2.1. By combining RustBelt's lifetime logic and the machinery of parametric prophecies, RustHornBelt successfully modeled and reasoned about prophetic borrows, i.e., Rust-style mutable borrows interacting with RustHorn-style prophecies.

Problem #1: Later Modality Still, RustHornBelt suffered from the *later modality* $\triangleright$, which comes from RustBelt's lifetime logic.

The later modality blocks RustHornBelt from proving that RustHorn-style reduction preserves *infinite behaviors* of programs, which is essential to justify RustHorn-style reduction in verifying *liveness properties*.

Also, even for *safety verification*, RustHornBelt encountered a problem of stripping off a statically unbounded number of later modalities. RustHornBelt found a workaround for this issue, using a technique nicknamed flexible step-indexing. But that substantially complicated the proof and the semantic model of Rust's types (Matsushita et al., 2022, § 3.5) (we explain this later in § 9.3).

Problem #2: Lack of Abstraction Also, RustHornBelt's proofs about mutable borrows *directly manipulated* an involved model of mutable borrows in an *ad hoc* way. As a result, their proofs are often complicated and hard to reuse. This is especially the case for their proofs about *reborrowing*. We need a good abstraction of RustHornBelt's reasoning about mutable borrows.

7.2 Proof Rules of Nola's Prophetic Borrows

Now we present the proof rules of Nola's later-free prophetic borrows. We first introduce the basic part, parametric prophecies (§ 7.2.1). Then we explain Nola's later-free prophetic borrows (§ 7.2.2). Finally, we present some examples of using Nola's prophetic borrows (§ 7.2.3).

7.2.1 Parametric Prophecies

We inherit from RustHornBelt the machinery of *parametric prophecies* to reason about prophecies flexibly in separation logic. Here we present the key definitions and proof rules. Its model is explained later in § 7.4.1.

Prophecy Variables First, we consider the set of *prophecy variables* (or *prophecies* in short) PrVar_A parameterized over any set $A \in \mathbb{U}$ of some universe $\mathbb{U}$. We use the metavariables x, y, z for prophecy variables.

Prophecy Assignment A *prophecy assignment* $\pi \in \mathit{PrAsn}$ assigns to each prophecy variable $x \in \mathit{PrVar}_A$ of any set $A \in \mathbb{U}$ a concrete *value* $\pi\, x \in A$. A prophecy assignment represents one possible future about the prophecy variables.

Clairvoyant Monad The *clairvoyant monad* $\mathit{Clair}\, A$ for a set A is simply the *reader monad* over prophecy assignments: $\mathit{Clair}\, A \triangleq \mathit{PrAsn} \to A$. We call a value $\hat{a} \in \mathit{Clair}\, A$ of the clairvoyant monad (i.e., a value parameterized over the prophecy assignment) a *clairvoyant value*. We mark clairvoyant values with a circumflex $\hat{\ }$.

Prophecy Observation The key Iris proposition for parametric prophecies is the *prophecy observation* $\langle \hat{\phi} \rangle \in \mathit{iProp}$, which *persistently* asserts that *all futures* $\pi \in \mathit{PrAsn}$ *that are currently possible satisfy the condition* $\hat{\phi}\, \pi$, specified by a *clairvoyant pure assertion* $\hat{\phi} \in \mathit{Clair}\, \mathit{Prop} = \mathit{PrAsn} \to \mathit{Prop}$.

$$\langle \hat{\phi} \rangle \text{ is persistent} \qquad \textsc{proph-obs-persist}$$

The prophecy observation is monotone over the clairvoyant pure proposition. Also, we can combine two prophecy observations to get a prophecy observation for the conjunction assertion.

$$\frac{\forall \pi.\ \hat{\phi}\, \pi \to \hat{\psi}\, \pi}{\langle \hat{\phi} \rangle \vDash \langle \hat{\psi} \rangle} \qquad \textsc{proph-obs-mono}$$

$$\langle \hat{\phi} \rangle * \langle \hat{\psi} \rangle = \langle \lambda \pi.\ \hat{\phi}\, \pi \wedge \hat{\psi}\, \pi \rangle \qquad \textsc{proph-obs-}*$$

The following is thekey rule for the *adequacy* of the prophecy observation $\langle \hat{\phi} \rangle$, associating its clairvoyant assertion $\hat{\phi}$ with the 'reality':[7]

$$\langle \hat{\phi} \rangle \vDash \exists \pi_0.\ \hat{\phi}\, \pi_0 \qquad \textsc{proph-obs-sat}$$

It states that the prophecy observation $\langle \hat{\phi} \rangle$ ensures that there exists at least one possible future $\pi_0 \in \mathit{PrAsn}$ that satisfies the clairvoyant pure assertion $\hat{\phi}\, \pi_0$. As a special case, if the clairvoyant assertion of the prophecy observation is constant over the prophecy assignment $\lambda\_.\, \phi$, then we directly get the assertion ϕ by PROPH-OBS-SAT:[8]

$$\langle \lambda\_.\, \phi \rangle \vDash \phi \qquad \textsc{proph-obs-const}$$

For example, if we have two prophetic observations $\langle \lambda \pi.\ \pi\, x = a \rangle$ and $\langle \lambda \pi.\ \pi\, x = a' \rangle$ for a prophecy variable x, we can combine the two to get the observation $\langle \lambda\_.\ a = a' \rangle$ (by PROPH-OBS-$*$ and PROPH-OBS-MONO) and get the equality $a = a'$ by PROPH-OBS-CONST.

7.2.2 Prophetic Borrows

Now we introduce our mechanism of prophetic borrows. Our prophetic borrow mechanism is built on our borrow mechanism presented in § 6.3.2, inheriting the overall structure.

[7] Our rule PROPH-OBS-SAT is improved from the original rule of RustHornBelt (Matsushita et al., 2022, § 3.2), where the entailment was weakened by a fancy update $\Rrightarrow_{\mathcal{N}_{\text{proph}}}$. We updated the resource algebra for prophecies PROPH to include the invariant for this rule into the validity predicate ✓, as we see later in § 7.4.1.

[8] Actually, we can also derive PROPH-OBS-SAT from PROPH-OBS-CONST, because the prophecy observation $\langle \hat{\phi} \rangle$ entails $\langle \lambda\_.\, \exists \pi_0.\ \hat{\phi}\, \pi_0 \rangle$ by PROPH-OBS-MONO. In this sense, the two rules for adequacy of parametric prophecies PROPH-OBS-SAT and PROPH-OBS-CONST are equivalent.

Again, the resource algebra $\textsc{Pbor}_{nProp}$ for the prophetic borrow mechanism is parameterized over the *syntactic data type for propositions nProp*. Also, Nola's prophetic borrow mechanism provides a *world satisfaction* $\mathsf{Wpbor}_M[\![\ ]\!]$ parameterized over the *semantic interpretation* $[\![\ ]\!] \colon nProp \to iProp$ as well as the *update modality* M used for *borrow subdivision*.

Basic Rules The prophetic borrow mechanism features the *prophetic* version of the *borrower*, *lender*, and *open borrower* tokens: $\mathsf{bor}^{\alpha}_{a,x}\ \Phi$, $\mathsf{lend}^{\alpha}_{\hat{a}}\ \Phi$, and $\mathsf{obor}^{\alpha}_{q,x}\ \Phi$. Their functionalities are analogous to the corresonding tokens $\mathsf{bor}^{\alpha}\ P$, $\mathsf{lend}^{\alpha}\ P$, $\mathsf{obor}^{\alpha}\ P$ of our borrow mechanism (§ 6.3.2), but they are enriched with the prophecies.

The prophetic borrower token $\mathsf{bor}^{\alpha}_{a,x}\ \Phi$ newly takes a *current value* $a \in A$ and a *prophecy* $x \in PrVar_A$, for an implicitly specified domain $A \in \mathbb{U}$. Also, the prophetic borrower token takes a *syntactic predicate* $\Phi \colon A \to nProp$ instead of just a proposition $P \in nProp$. Similarly, the prophetic open borrower token $\mathsf{obor}^{\alpha}_{q,x}\ \Phi$ takes a prophecy $x \in PrVar_A$ and a predicate $\Phi \colon A \to nProp$ for $A \in \mathbb{U}$. The prophetic lender $\mathsf{lend}^{\alpha}_{\hat{a}}\ \Phi$ takes a predicate $\Phi \colon A \to nProp$ and a *clairvoyant value* $\hat{a}$ for $A \in \mathbb{U}$. The clairvoyant value of the lender is typically set to the value of a prophecy $\lambda\pi.\ \pi\, x$.

The tokens are all timeless:

$$\mathsf{bor}^{\alpha}_{a,x}\ \Phi \text{ is timeless} \quad \textsc{pbor-timeless} \qquad \mathsf{lend}^{\alpha}_{\hat{a}}\ \Phi \text{ is timeless} \quad \textsc{plend-timeless}$$

$$\mathsf{obor}^{\alpha}_{q,x}\ \Phi \text{ is timeless} \quad \textsc{pobor-timeless}$$

By storing $[\![\Phi\ a]\!]$, we can create a prophetic borrower $\mathsf{bor}^{\alpha}_{a,x}\ \Phi$ and a prophetic lender $\mathsf{lend}^{\alpha}_{\lambda\pi.\ \pi\, x}\ \Phi$ for some freshly taken prophecy variable x:

$$[\![\Phi\ a]\!] \ \vDash \dot{\Rrightarrow}^{\mathsf{Wpbor}_M[\![\ ]\!]} \big(\exists x.\ \ \mathsf{bor}^{\alpha}_{a,x}\ \Phi \ * \ \mathsf{lend}^{\alpha}_{\lambda\pi.\ \pi\, x}\ \Phi\big) \quad \textsc{pbor-plend-new}$$

After the lifetime α has died, the prophetic lender $\mathsf{lend}^{\alpha}_{\hat{a}}\ \Phi$ *can retrieve the borrowed content*, like LEND-RETRIEVE:

$$\dagger\alpha \ * \ \mathsf{lend}^{\alpha}_{\hat{a}}\ \Phi \ \ \vDash M^{\mathsf{Wpbor}_M[\![\ ]\!]} \big(\exists\, a' \text{ s.t. } \langle \lambda\pi.\ \hat{a}\,\pi = a' \rangle.\ [\![\Phi\ a']\!]\big) \quad \textsc{plend-retrieve}$$

Notably, the value a' of the content $[\![\Phi\ a']\!]$ is ensured to be equal to the lender's *clairvoyant value* $\hat{a}\,\pi$ by the *prophecy observation* $\langle \lambda\pi.\ \hat{a}\,\pi = a' \rangle$.

A prophetic borrower $\mathsf{bor}^{\alpha}_{a,x}\ \Phi$ can *open the borrow* by storing a live lifetime token $[\alpha]_q$, like BOR-OPEN:

$$[\alpha]_q \ * \ \mathsf{bor}^{\alpha}_{a,x}\ \Phi \ \ \vDash M^{\mathsf{Wpbor}_M[\![\ ]\!]} \big(\mathsf{obor}^{\alpha}_{q,x}\ \Phi \ * \ [\![\Phi\ a]\!]\big) \quad \textsc{pbor-open}$$

We get a prophetic open borrower token $\mathsf{obor}^{\alpha}_{q,x}\ \Phi$ and the borrowed content $[\![\Phi\ a]\!]$. Importantly, the borrowed content $[\![\Phi\ a]\!]$ has the current value a and is *not* weakened by the later modality $\triangleright$.

A prophetic open borrower can close a borrow by storing the content back, like OBOR-CLOSE:

$$\mathsf{obor}^{\alpha}_{q,x}\ \Phi \ * \ [\![\Phi\ a']\!] \ \vDash \dot{\Rrightarrow}^{\mathsf{Wpbor}_M[\![\ ]\!]} \big([\alpha]_q \ * \ \mathsf{bor}^{\alpha}_{a',x}\ \Phi\big) \quad \textsc{pobor-close}$$

Notably, the open borrower token can specify *any* value $a' \in A$ for the content $[\![\Phi\ a']\!]$, which is not necessarily equal to the original value before opening the borrow by PBOR-OPEN. The obtained borrower token $\mathsf{bor}^{\alpha}_{a',x}\ \Phi$ has this new value a'.

When a prophetic borrower throws away its ownership, it can *resolve the prophecy* x into the current value a, which is a new rule for prophetic borrowing:

$$[\alpha]_q \ * \ \mathsf{bor}^{\alpha}_{a,x}\ \Phi \ \ \vDash M^{\mathsf{Wpbor}_M[\![\ ]\!]} \big([\alpha]_q \ * \ \langle \lambda\pi.\ \pi\, x = a \rangle\big) \quad \textsc{pbor-resolve}$$

By the prophecy resolution, we narrow the 'possible futures' (or prophecy assignments) π to only those that assign the value a to the prophecy variable x. This new information about the possible futures is asserted by the *prophecy observation* $\langle \lambda\pi.\ \pi\, x = a \rangle$.

Converting Tokens We have the following rules for modifying the lifetime of tokens, just like BOR-LFT, LEND-LFT and OBOR-LFT:

$$\beta \sqsubseteq \alpha \ * \ \mathsf{bor}^{\alpha}_{a,x}\ \Phi \ \vDash \ \mathsf{bor}^{\beta}_{a,x}\ \Phi \qquad \text{PBOR-LFT}$$

$$\alpha \sqsubseteq \beta \ * \ \mathsf{lend}^{\alpha}_{\hat{a}}\ \Phi \ \vDash \ \mathsf{lend}^{\beta}_{\hat{a}}\ \Phi \qquad \text{PLEND-LFT}$$

$$\beta \sqsubseteq \alpha \ * \ ([\alpha]_q \twoheadrightarrow [\beta]_r) \ * \ \mathsf{obor}^{\alpha}_{q,x}\ \Phi \ \vDash \ \mathsf{obor}^{\beta}_{r,x}\ \Phi \qquad \text{POBOR-LFT}$$

Also, we can 'fake' a prophetic borrower token $\mathsf{bor}^{\alpha}_{a,x}\ \Phi$ when the lifetime α is dead, just like BOR-FAKE:

$$\dagger\alpha \ \vDash \ \mathsf{bor}^{\alpha}_{a,x}\ \Phi \qquad \text{PBOR-FAKE}$$

Borrow Subdivision and Merger We have the following rule for subdividing a prophetic borrower *without* changing the prophecy x, just like OBOR-SUBDIV:

$$\beta \sqsubseteq \alpha \ * \ \mathsf{obor}^{\alpha}_{q,x}\ \Phi \ * \ [\![\Phi\ a]\!] \ * \ \big(\forall a'.\ \dagger\beta \ * \ [\![\Psi\ a']\!] \twoheadrightarrow M\ [\![\Phi\ a']\!]\big)$$
$$\vDash \dot{\Rrightarrow}^{\mathrm{Wpbor}_M [\![\,]\!]}\ [\alpha]_q \ * \ \mathsf{bor}^{\beta}_{a,x}\ \Psi$$
$$\text{POBOR-NSUBDIV}$$

This takes a 'converter' $\forall a'.\ \dagger\beta \ * \ [\![\Psi\ a']\!] \twoheadrightarrow M\ [\![\Phi\ a']\!]$, which turns the new content $[\![\Psi\ a']\!]$ into the old content $[\![\Phi\ a']\!]$ for any final value a' with the update modality M, under the assumption that the lifetime β is dead.

We have the following advanced rule for subdividing a prophetic borrower into smaller ones with *new prophecies*, a substantially enriched version of OBOR-SUBDIV:

$$\beta \sqsubseteq \alpha \ * \ \mathsf{obor}^{\alpha}_{q,x}\ \Phi \ * \ \mathop{\Huge *}_i [\![\Psi_i\ b_i]\!] \ * \ \Big(\forall \bar{b}'.\ \dagger\beta \ * \ \mathop{\Huge *}_i [\![\Psi_i\ b'_i]\!] \twoheadrightarrow M\ [\![\Phi\ (f(\bar{b}'))]\!]\Big)$$
$$\vDash \dot{\Rrightarrow}^{\mathrm{Wpbor}_M [\![\,]\!]}\ \Big([\alpha]_q \ * \ \exists\,\bar{y}\ \text{s.t.}\ \big\langle \lambda\pi.\ \pi\,x = f(\overline{\pi\,y})\big\rangle.\ \mathop{\Huge *}_i \mathsf{bor}^{\beta}_{y_i,b_i}\ \Psi_i\Big)$$
$$\text{POBOR-SUBDIV}$$

The prophecies $\bar{y}$ for the new borrowers are freshly taken. The rule takes a *conversion function* $f \colon \prod_i B_i \longrightarrow A$, which maps the final values of the new borrowers's contents $\bar{b}'$ into the final value of the old borrower's content $f(\bar{b}')$. The converter

$$\forall \bar{b}'.\ \dagger\beta \ * \ \mathop{\Huge *}_i [\![\Psi_i\ b'_i]\!] \twoheadrightarrow M\ [\![\Phi\ (f(\bar{b}'))]\!]$$

is simply universally quantified over the final values $\bar{b}'$ of the new borrowers' contents (just like in POBOR-NSUBDIV) and uses the value $f(\bar{b}')$ for the old borrower's content. By this borrow subdivision, the old prophecy x is *partially resolved* into the value $f(\overline{\pi\,y})$, which depends on the new prophecies' values $\pi\,y_i$. This is asserted by the *prophecy observation* $\big\langle \lambda\pi.\ \pi\,x = f(\overline{\pi\,y})\big\rangle$. Note that we can derive the simple resolution rule PBOR-RESOLVE we presented from this rule POBOR-SUBDIV and the borrow opening rule PBOR-OPEN.

We also have the following rule for merging and subdividing prophetic borrowers, obtained by extending POBOR-SUBDIV, just like OBOR-MERGE-SUBDIV is obtained from OBOR-SUBDIV:

$$\mathop{\Huge *}_j \big(\beta \sqsubseteq \alpha_j \ * \ \mathsf{obor}^{\alpha_j}_{q_j,x_j}\ \Phi_j\big) \ * \ \mathop{\Huge *}_i [\![\Psi_i\ b_i]\!] \ *$$
$$\Big(\forall \bar{b}'.\ \dagger\beta \ * \ \mathop{\Huge *}_i [\![\Psi_i\ b'_i]\!] \twoheadrightarrow M\ \big(\mathop{\Huge *}_j [\![\Phi_j(f_j(\bar{b}'))]\!]\big)\Big)$$
$$\vDash \dot{\Rrightarrow}^{\mathrm{Wpbor}_M [\![\,]\!]}\ \Big([\alpha]_q \ * \ \exists\,\bar{y}\ \text{s.t.}\ \big\langle \lambda\pi.\ \forall j.\ \pi\,x_j = f_j(\overline{\pi\,y})\big\rangle.\ \mathop{\Huge *}_i \mathsf{bor}^{\beta}_{y_i,b_i}\ \Psi_i\Big)$$

POBOR-MERGE-SUBDIV

This rule simply allows taking multiple open borrowers $\mathsf{obor}^{\alpha_j}_{q_j,x_j}\ \Phi_j$ instead of only one $\mathsf{obor}^{\alpha}_{q,x}\ \Phi$ as in POBOR-SUBDIV. We also take conversion functions f_j and get a prophecy observation $\langle \lambda\pi.\ \forall j.\ \pi\, x_j = f_j(\overline{\pi\, y}) \rangle$.

Reborrowing Reborrowing is much trickier for prophetic borrows than for ordinary borrows. The rule BOR-REBORROW for an ordinary borrow reborrowed $\mathsf{bor}^{\alpha}\ P$ to create a reborrower $\mathsf{bor}^{\alpha\sqcap\beta}\ P$ and a promise to return the original borrower $\dagger\beta \twoheadrightarrow \mathsf{bor}^{\alpha}\ P$. But for prophetic borrowing, we should take a new prophecy for the reborrower, and the effect of the reborrower's mutation should be *communicated* to the original borrower via that prophecy.

To handle this difficulty, we newly designed a proof rule for prophetic reborrowing, which directly targets the situation where a borrower $\mathsf{bor}^{\beta}_{b,y}\ \Psi$ to be reborrowed is taken from some outer borrower $\mathsf{obor}^{\alpha}_{q,x}\ \Phi$:

$$\begin{aligned}&\mathsf{obor}^{\alpha}_{q,x}\ \Phi \ *\ [\beta]_r \ *\ \mathsf{bor}^{\beta}_{b,y}\ \Psi \ *\ \big(\forall b'.\ \dagger\alpha \ *\ \mathsf{bor}^{\beta}_{b',y}\ \Psi \twoheadrightarrow M\,[\![\Phi\,(f\,b')]\!]\big)\\ &\vDash M^{\mathsf{Wpbor}_M[\![\,]\!]}\Big([\alpha]_q \ *\ [\beta]_r \ *\ \exists\, y'\ \text{s.t.}\ \langle \lambda\pi.\ \pi\, x = f(\pi\, y') \rangle.\ \mathsf{bor}^{\alpha\sqcap\beta}_{b,y'}\ \Psi\Big)\end{aligned}$$

POBOR-PBOR-REBORROW

We reborrow the inner borrower $\mathsf{bor}^{\beta}_{b,y}\ \Psi$ to create a new borrower $\mathsf{bor}^{\alpha\sqcap\beta}_{b,y'}\ \Psi$ with a freshly taken prophecy y'. We take a *conversion function* $f\colon B \longrightarrow A$ that takes the final version of the 'current' value of the inner borrower $\mathsf{bor}^{\beta}_{b',y}\ \Psi$ and returns the final value of the outer borrower's content $[\![\Phi\,(f\,b')]\!]$. Accordingly, the rule takes the converter

$$\forall b'.\ \dagger\alpha \ *\ \mathsf{bor}^{\beta}_{b',y}\ \Psi \twoheadrightarrow M\,[\![\Phi\,(f\,b')]\!].$$

Because the value b' here is determined by the mutation of the reborrower $\mathsf{bor}^{\alpha\sqcap\beta}_{b,y'}\ \Psi$, the value of the outer borrower's prophecy x is *partially resolved* to the value $f(\pi\, y')$ calculated from the new prophecy's value $\pi\, y'$, resulting in the prophecy observation $\langle \lambda\pi.\ \pi\, x = f(\pi\, y') \rangle$.

Lender Splitting We can also *split* a prophetic lender token $\mathsf{lend}^{\alpha}_{\hat a}\ \Phi$ into multiple lender tokens $\ast_i\ \mathsf{lend}^{\alpha}_{\hat b_i}\ \Psi_i$ by the following rule, like LEND-SPLIT:

$$\begin{aligned}&\Big(\forall\, a'\ \text{s.t.}\ \langle \lambda\pi.\ \hat a\,\pi = a' \rangle.\ [\![\Phi\ a']\!] \twoheadrightarrow M\,\big(\mathop{\ast}_i\big(\exists\, b'\ \text{s.t.}\ \langle \lambda\pi.\ \hat b_i\,\pi = b' \rangle.\ [\![\Psi_i\, b']\!]\big)\big)\Big) \ *\\ &\qquad\qquad \mathsf{lend}^{\alpha}_{\hat a}\ \Phi \ \vDash\ \dot{\Rrightarrow}^{\mathsf{Wbor}_M[\![\,]\!]}\Big(\mathop{\ast}_i \mathsf{lend}^{\alpha}_{\hat b_i}\ \Psi_i\Big)\end{aligned}$$

PLEND-SPLIT

The converter for this rule handles prophetic observations.

World Satisfaction Allocation We also have the rule for allocating the world satisfaction $\mathsf{Wpbor}_M[\![\,]\!]$, just like WBOR-ALLOC:

$$\vDash \dot{\Rrightarrow}\ \big(\exists\gamma_{\mathrm{Bor}}.\ \forall[\![\,]\!], M.\ \mathsf{Wpbor}_M[\![\,]\!]\big) \quad \text{WPBOR-ALLOC}$$

It takes a fresh *ghost name* γ_{Bor} for our *borrow machinery* § 6.3.2, because our prophetic borrow machinery is built on top of our borrow machinery, as we see in § 7.4.3.

7.2.3 Examples

Now we present some examples of using our prophetic borrow mechanism.

To begin with, we instantiate our prophecy borrow mechanism with *nProp* of (3.16) (§ 3.3.3) extended with a prophetic borrower connective $\mathsf{bor}^{\alpha}_{a,x}\ \Phi$ interpreted as the prophetic borrower token, just like § 6.3.3:

$$[\![\mathsf{bor}^{\alpha}_{a,x}\ \Phi]\!] \quad \triangleq \quad \mathsf{bor}^{\alpha}_{a,x}\ \Phi$$

Mutable Reference to a Pair For a simple example, a mutable reference to a pair ℓ: `&α mut (T, U)` can be expressed as follows:

$$\mathsf{bor}^{\alpha}_{(a,b),x}\,\big(\lambda(a',b').\ \Phi\,\ell\,a' \ \star\ \Psi\,(\ell+k)\,b'\,\big)$$

Here, we modeled the types `T`, `U` as predicates $\Phi\colon Loc \to A \to nProp$ and $\Psi\colon Loc \to B \to nProp$, and assumed that the size of `T` is k. We assume that the mutable reference has the current value $(a,b) \in A \times B$ and the prophecy $x \in PrVar_{A\times B}$. For example, when `T` is an integer type `int`, then we can set $A = \mathbb{Z}$, $\Phi\,\ell\,n \triangleq \ell \mapsto n$, and $k = 1$.

We can split the mutable reference ℓ: `&α mut (T, U)` into the mutable references to ℓ: `&α mut T` and $(\ell+k)$: `&α mut U` as follows:

$$[\alpha]_q \ast \mathsf{bor}^{\alpha}_{(a,b),x}\,\big(\lambda(a',b').\ \Phi\,\ell\,a' \ \star\ \Psi\,(\ell+k)\,b'\,\big) \ \vDash M^{\mathsf{Wbor}_M[\![\,]\!]}$$
$$\big([\alpha]_q \ast \exists\, y,z \text{ s.t. } \big\langle \lambda\pi.\ \pi\,x = (\pi\,y, \pi\,z)\big\rangle.\ \mathsf{bor}^{\alpha}_{a,y}\ \Phi\,\ell \ast \mathsf{bor}^{\alpha}_{b,z}\ \Psi\,(\ell+k)\,\big)$$

We can prove this using POBOR-SUBDIV. Importantly, we get the prophecy observation $\big\langle \lambda\pi.\ \pi\,x = (\pi\,y, \pi\,z)\big\rangle$ that partially resolves the original prophecy x into the pair of the new prophecies's values $(\pi\,y, \pi\,z)$.

Conversely, we can also merge ℓ: `&α mut T` and $(\ell+k)$: `&α mut U` into ℓ: `&α mut (T, U)` as follows:

$$[\alpha]_q \ast \mathsf{bor}^{\alpha}_{a,y}\ \Phi\,\ell \ast \mathsf{bor}^{\alpha}_{b,z}\ \Psi\,(\ell+k) \ \vDash M^{\mathsf{Wbor}_M[\![\,]\!]}$$
$$\big([\alpha]_q \ast \exists\, x \text{ s.t. } \big\langle \lambda\pi.\ (\pi\,y, \pi\,z) = \pi\,x\big\rangle.\ \mathsf{bor}^{\alpha}_{(a,b),x}\,\big(\lambda(a',b').\ \Phi\,\ell\,a' \ \star\ \Psi\,(\ell+k)\,b'\,\big)\big)$$

We can prove this using POBOR-MERGE-SUBDIV. By the rule, we get two prophecy observations $\big\langle \lambda\pi.\ \pi\,y = (\pi\,x).1\big\rangle$ and $\big\langle \lambda\pi.\ \pi\,z = (\pi\,x).2\big\rangle$, whose combination gives $\big\langle \lambda\pi.\ (\pi\,y, \pi\,z) = \pi\,x\big\rangle$.

Nested Mutable Reference For a more advanced example, a *nested mutable reference* ℓ: `&α mut &β mut T` can be expressed as follows, modeling the type `T` as a predicate $\Phi\colon Loc \to A \to nProp$:

$$\mathsf{bor}^{\alpha}_{(a,y),x}\,\big(\lambda(a',y').\ \exists\,\ell'.\ \ell \mapsto \ell' \ \star\ \mathsf{bor}^{\beta}_{a',y'}\,(\Phi\,\ell')\,\big).$$

The current value of the outer borrow is $(a,y) \in A \times PrVar_A$, the pair of the current value a and prophecy y of the inner borrow.

Like in § 6.3.3, let us consider the *dereference* of the nested mutable reference $!\ell$: `&(α ⊓ β) mut T`. We can verify the following total Hoare triple for this dereference:

$$\Big[\ [\alpha\sqcap\beta]_q \ast \mathsf{bor}^{\alpha}_{(a,y),x}\,\big(\lambda(a',y').\ \exists\,\ell'.\ \ell \mapsto \ell' \ \star\ \mathsf{bor}^{\beta}_{a',y'}\,(\Phi\,\ell')\,\big)\Big]\ \ !\ell$$
$$\Big[\ \lambda v.\ [\alpha\sqcap\beta]_q \ast \exists\,\ell' \text{ s.t. } v = \ell',\ y' \text{ s.t. } \big\langle \lambda\pi.\ \pi\,x = (\pi\,y', y)\big\rangle.\ \mathsf{bor}^{\alpha\sqcap\beta}_{a,y'}(\Phi\,\ell')\ \Big]^{\mathsf{Wbor}_{\Rrightarrow}[\![\,]\!]} \tag{7.8}$$

This significantly extends the total Hoare triple (6.1) for ordinary borrows. The reborrowed reference $\mathsf{bor}^{\alpha\sqcap\beta}_{a,y'}(\Phi\,\ell')$ inherits the current value a and uses a newly taken prophecy y'. For the prophecy x of the outer borrow, the second component is resolved to the prophecy y of the inner borrow, and the first component is partially resolved to the value $\pi\,y'$ of the newly taken prophecy y', resulting in the prophecy observation $\langle \lambda\pi.\ \pi\,x = (\pi\,y', y)\rangle$. Also note that the resulting inner borrow is *free of the later modality* $\triangleright$, enabling accessing further inside the borrow. Our later-free prophetic borrow mechanism thus enables *prophetic* liveness verification of nested borrows.

The *prophetic* verification of this dereference goes as follows.

Proof of (7.8). For convenience, we name the content predicate of the outer borrower token $\mathtt{refpbor}\,(a', y') \triangleq \exists\,\ell'.\ \ell \mapsto \ell' * \mathsf{bor}^{\beta}_{a',y'}(\Phi\,\ell')$.

By PBOR-OPEN, we open the outer borrower $\mathsf{bor}^{\alpha}_{(a,y),x}\ \mathtt{refpbor}$, getting an *open borrower token* $\mathsf{obor}^{\alpha}_{q,x}\ \mathtt{refpbor}$ and the content

$$[\![\mathtt{refpbor}\,(a, y)]\!] \quad = \quad \exists\ell'.\ \ell \mapsto \ell' \ * \ \mathsf{bor}^{\beta}_{a,y'}(\Phi\,\ell').$$

We destruct the existential quantifier to get the location $\ell' \in \mathit{Loc}$. We perform the dereference $!\ell$ using the points-to token $\ell \mapsto \ell'$.

Then we apply the rule POBOR-PBOR-REBORROW for *prophetic reborrowing*, storing the open borrower token $\mathsf{obor}^{\alpha}_{q,x}\ \mathtt{refpbor}$ for the outer reference, the borrower token $\mathsf{bor}^{\beta}_{a',y'}(\Phi\,\ell')$ for the inner reference, and the following converter:

$$\forall a'.\ \dagger\alpha \ * \ \mathsf{bor}^{\beta}_{a',y'}(\Phi\,\ell') \ -\!\!*\ M\,[\![\mathtt{refpbor}\,(a', y')]\!].$$

This converter can easily be constructed from the points-to token $\ell \mapsto \ell'$. Finally, we get the reborrower $\mathsf{bor}^{\alpha\sqcap\beta}_{a,y'}(\Phi\,\ell')$ and the prophecy observation $\langle \lambda\pi.\ \pi\,x = (\pi\,y', y)\rangle$ for a newly taken prophecy y'. ☐

Remark 7.1 (Nested Prophecies). Here we consider a prophecy $x \in \mathit{PrVar}_{A\times\mathit{PrVar}_A}$ for a value that *further contains a prophecy* PrVar_A. Such *nested prophecies* are key to the simplicity of our approach. RustHornBelt did not support such nested prophecies and instead employed unnested prophecies like $\mathit{PrVar}_{A\times A}$ for nested mutable references. Due to this, RustHornBelt suffered from the problem of managing dependent prophecies. We avoid that problem by using nested prophecies.

7.3 Semantic Alteration by Derivability

We can support *semantic alteration* of the predicate Φ of the prophetic borrower, lender and open borrower connectives by the general *derivability* construction presented in Chapter 4, just like for ordinary borrows (§ 6.4).

Like in § 6.4, we add to the judgment data type *Judg* the following constructor:

$$\mathit{Judg} \ni J \ ::= \ \cdots \ \mid \ \Phi \Rightarrow \Psi \quad (\Phi, \Psi \colon A \to \mathit{nProp};\ A \in \mathbb{U})$$

The judgment $\Phi \Rightarrow \Psi$ means that $\Phi\,a$ can be converted into $\Psi\,a$ for any value $a \in A$.

Next, we construct the *interpretation* $[\![\ ]\!]_\delta \colon \mathit{nProp} \to \mathit{iProp}$ *parameterized* with the derivability predicate $\delta \colon \mathit{Judg} \to \mathit{iProp}$ (§ 4.2). The syntactic prophetic borrower, lender and open borrower assertions $\mathsf{bor}^{\alpha}_{a,x}\,\Phi$, $\mathsf{lend}^{\alpha}_{\hat{a}}\,\Phi$, $\mathsf{obor}^{\alpha}_{q,x}\,\Phi$ are interpreted as follows:

$$[\![\mathsf{bor}^{\alpha}_{a,x}\,\Phi]\!]_\delta \quad \triangleq \quad \exists\Psi.\ \delta\,(\Phi \Rightarrow \Psi) \ * \ \delta\,(\Psi \Rightarrow \Phi) \ * \ \mathsf{bor}^{\alpha}_{a,x}\,\Psi$$

$$[\![\mathsf{lend}^{\alpha}_{\hat{a}}\,\Phi]\!]_\delta \quad \triangleq \quad \exists\Psi.\ \delta\,(\Phi \Rightarrow \Psi) \ * \ \mathsf{lend}^{\alpha}_{\hat{a}}\,\Psi$$

$$\llbracket \mathtt{obor}^{\alpha}_{q,x}\ \Phi \rrbracket_{\delta} \quad \triangleq \quad \exists \Psi.\ \delta\,(\Phi \Rightarrow \Psi) \ * \ \mathtt{obor}^{\alpha}_{q,x}\ \Psi$$

Now we can define the semantics $\llbracket\ \rrbracket^{+}_{\delta}\colon \mathit{Judg} \to \mathit{iProp}$ of judgments *Judg parameterized* with the derivability predicate $\delta\colon \mathit{Judg} \to \mathit{iProp}$. For the judgment $\Phi \Rightarrow \Psi$, we define the semantics as follows:

$$\llbracket \Phi \Rightarrow \Psi \rrbracket^{+}_{\delta} \quad \triangleq \quad \forall a.\ \llbracket \Phi\ a \rrbracket_{\delta} \mathrel{-\!\!*} \dot{\Rrightarrow} \llbracket \Psi\ a \rrbracket_{\delta}.$$

Now we automatically obtain the *best derivability predicate* der $\in \mathit{Judg} \to \mathit{iProp}$ and the set of *good derivability predicates* $\mathit{Deriv} \subseteq \mathit{Judg} \to \mathit{iProp}$ by Definition 4.1 (§ 4.3).

Now the prophetic borrower, lender and open borrower connectives satisfy the following rules for semantic alteration by derivability, which hold for any good derivability predicate $\delta \in \mathit{Deriv}$, just like $\mathtt{bor}$-ALTER, $\mathtt{lend}$-ALTER and $\mathtt{obor}$-ALTER:

$$\delta\,(\Phi \Rightarrow \Psi) \ * \ \delta\,(\Psi \Rightarrow \Phi) \ * \ \llbracket \mathtt{bor}^{\alpha}_{a,x}\ \Phi \rrbracket_{\delta} \ \vDash \ \llbracket \mathtt{bor}^{\alpha}_{a,x}\ \Psi \rrbracket_{\delta} \qquad \mathtt{pbor}\text{-}\textsc{alter}$$

$$\delta\,(\Phi \Rightarrow \Psi) \ * \ \llbracket \mathtt{lend}^{\alpha}_{\hat{a}}\ \Phi \rrbracket_{\delta} \ \vDash \ \llbracket \mathtt{lend}^{\alpha}_{\hat{a}}\ \Psi \rrbracket_{\delta} \qquad \mathtt{plend}\text{-}\textsc{alter}$$

$$\delta\,(\Psi \Rightarrow \Phi) \ * \ \llbracket \mathtt{obor}^{\alpha}_{q,x}\ \Phi \rrbracket_{\delta} \ \vDash \ \llbracket \mathtt{obor}^{\alpha}_{q,x}\ \Psi \rrbracket_{\delta} \qquad \mathtt{pobor}\text{-}\textsc{alter}$$

For a simple example, using $\mathtt{pbor}$-ALTER we can prove

$$\llbracket \mathtt{bor}^{\alpha}_{a,x}\ (\lambda a'.\ \Phi\ a' * \Psi\ a') \rrbracket_{\delta} \ = \ \llbracket \mathtt{bor}^{\alpha}_{a,x}\ (\lambda a'.\ \Psi\ a' * \Phi\ a') \rrbracket_{\delta}$$

for any $\delta \in \mathit{Deriv}$, just like (4.19).

Also, the following basic conversion rules hold for any δ:

$$\beta \sqsubseteq \alpha \ * \ \llbracket \mathtt{bor}^{\alpha}_{a,x}\ \Phi \rrbracket_{\delta} \ \vDash \ \llbracket \mathtt{bor}^{\beta}_{a,x}\ \Phi \rrbracket_{\delta} \qquad \mathtt{pbor}\text{-}\textsc{lft}$$

$$\alpha \sqsubseteq \beta \ * \ \llbracket \mathtt{lend}^{\alpha}_{\hat{a}}\ \Phi \rrbracket_{\delta} \ \vDash \ \llbracket \mathtt{lend}^{\beta}_{\hat{a}}\ \Phi \rrbracket_{\delta} \qquad \mathtt{plend}\text{-}\textsc{lft}$$

$$\beta \sqsubseteq \alpha \ * \ ([\alpha]_q \mathrel{-\!\!*} [\beta]_r) \ * \ \llbracket \mathtt{obor}^{\alpha}_{q,x}\ \Phi \rrbracket_{\delta} \ \vDash \ \llbracket \mathtt{obor}^{\beta}_{r,x}\ \Phi \rrbracket_{\delta} \qquad \mathtt{pobor}\text{-}\textsc{lft}$$

$$\dagger\alpha \ \vDash \ \llbracket \mathtt{bor}^{\alpha}_{a,x}\ \Phi \rrbracket_{\delta} \qquad \mathtt{pbor}\text{-}\textsc{fake}$$

These semantically alterable connectives satisfy the following proof rules for manipulating prophetic borrows, just like those presented in § 7.2.2. Here, we consider the interpretation $\llbracket\ \rrbracket \triangleq \llbracket\ \rrbracket_{\mathrm{der}}$ by the best derivability predicate der. These rules are easily derived from the original rules in § 7.2.2, especially using the *soundness* of der (Theorem 4.3).

$$\llbracket \Phi\ a \rrbracket \ \vDash \dot{\Rrightarrow}^{\mathrm{Wpbor}_M \llbracket\ \rrbracket} \big(\llbracket \mathtt{bor}^{\alpha}_{a,x}\ \Phi \rrbracket \ * \ \llbracket \mathtt{lend}^{\alpha}_{\hat{a}}\ \Phi \rrbracket \big) \qquad \mathtt{pbor}\text{-}\mathtt{plend}\text{-}\textsc{new}$$

$$\dagger\alpha \ * \ \llbracket \mathtt{lend}^{\alpha}_{\hat{a}}\ \Phi \rrbracket \ \vDash M^{\mathrm{Wpbor}_M \llbracket\ \rrbracket} \big(\exists\, a \text{ s.t. } \langle \lambda\pi.\ \hat{a}\,\pi = a \rangle.\ \llbracket \Phi\ a \rrbracket \big) \qquad \mathtt{plend}\text{-}\textsc{retrieve}$$

$$[\alpha]_q \ * \ \llbracket \mathtt{bor}^{\alpha}_{a,x}\ \Phi \rrbracket \ \vDash M^{\mathrm{Wpbor}_M \llbracket\ \rrbracket} \big(\llbracket \mathtt{obor}^{\alpha}_{q,x}\ \Phi \rrbracket \ * \ \llbracket \Phi\ a \rrbracket \big) \qquad \mathtt{pbor}\text{-}\textsc{open}$$

$$\llbracket \mathtt{obor}^{\alpha}_{q,x}\ \Phi \rrbracket \ * \ \llbracket \Phi\ a' \rrbracket \ \vDash \dot{\Rrightarrow}^{\mathrm{Wpbor}_M \llbracket\ \rrbracket} \big([\alpha]_q \ * \ \llbracket \mathtt{bor}^{\alpha}_{a',x}\ \Phi \rrbracket \big) \qquad \mathtt{pobor}\text{-}\textsc{close}$$

$$[\alpha]_q \ * \ \llbracket \mathtt{bor}^{\alpha}_{a,x}\ \Phi \rrbracket * \ \vDash M^{\mathrm{Wpbor}_M \llbracket\ \rrbracket} \big([\alpha]_q \ * \ \langle \lambda\pi.\ \pi\,x = a \rangle \big) \qquad \mathtt{pbor}\text{-}\textsc{resolve}$$

$$\begin{gathered} \beta \sqsubseteq \alpha \ * \ \llbracket \mathtt{obor}^{\alpha}_{q,x}\ \Phi \rrbracket \ * \ \llbracket \Phi\ a \rrbracket \ * \ \big(\forall a'.\ \dagger\beta \ * \ \llbracket \Psi\ a' \rrbracket \mathrel{-\!\!*} M\, \llbracket \Phi\ a' \rrbracket \big) \\ \vDash \dot{\Rrightarrow}^{\mathrm{Wpbor}_M \llbracket\ \rrbracket} \ [\alpha]_q \ * \ \llbracket \mathtt{bor}^{\beta}_{a,x}\ \Psi \rrbracket \\ \mathtt{pobor}\text{-}\textsc{nsubdiv} \end{gathered}$$

$$\begin{gathered} \beta \sqsubseteq \alpha \ * \ \llbracket \mathtt{obor}^{\alpha}_{q,x}\ \Phi \rrbracket \ * \ \mathop{\Large *}_i \llbracket \Psi_i\ b_i \rrbracket \ * \ \Big(\forall \bar{b'}.\ \dagger\beta \ * \ \mathop{\Large *}_i \llbracket \Psi_i\ b'_i \rrbracket \mathrel{-\!\!*} M\, \llbracket \Phi\ (f(\bar{b'})) \rrbracket \Big) \\ \vDash \dot{\Rrightarrow}^{\mathrm{Wpbor}_M \llbracket\ \rrbracket} \Big([\beta]_q \ * \ \exists\, \bar{y} \text{ s.t. } \langle \lambda\pi.\ \pi\,x = f(\overline{\pi\,y}) \rangle.\ \mathop{\Large *}_i \llbracket \mathtt{bor}^{\beta}_{y_i,b_i}\ \Psi_i \rrbracket \Big) \\ \mathtt{pobor}\text{-}\textsc{subdiv} \end{gathered}$$

$$
\begin{aligned}
&\mathop{\Large\ast}\limits_{j} \big(\beta \sqsubseteq \alpha_j \;\ast\; \llbracket \mathtt{obor}^{\alpha_j}_{q_j, x_j}\ \Phi_j \rrbracket \big) \;\ast\; \mathop{\Large\ast}\limits_{i} \llbracket \Psi_i\, b_i \rrbracket \;\ast\\
&\quad \Big(\forall \bar{b}'.\ \dagger\beta \;\ast\; \mathop{\Large\ast}\limits_{i} \llbracket \Psi_i\, b'_i \rrbracket \mathrel{-\!\!*} M \big(\mathop{\Large\ast}\limits_{j} \llbracket \Phi_j\, (f_j(\bar{b}')) \rrbracket \big) \Big)\\
&\qquad\qquad \vDash \dot{\Rrightarrow}^{\mathrm{Wpbor}_M \llbracket\,\rrbracket} \Big([\beta]_q \;\ast\; \exists\, \bar{y} \text{ s.t. } \big\langle \lambda\pi.\ \forall j.\ \pi\, x_j = f_j(\overline{\pi\, y}) \big\rangle.\ \mathop{\Large\ast}\limits_{i} \llbracket \mathtt{bor}^{\beta}_{y_i, b_i}\ \Psi_i \rrbracket \Big)
\end{aligned}
$$

pobor-MERGE-SUBDIV

$$
\begin{aligned}
&\llbracket \mathtt{obor}^{\alpha}_{q, x}\ \Phi \rrbracket \;\ast\; [\beta]_r \;\ast\; \llbracket \mathtt{bor}^{\beta}_{b, y}\ \Psi \rrbracket \;\ast\; \big(\forall b'.\ \dagger\alpha \;\ast\; \llbracket \mathtt{bor}^{\beta}_{b', y}\ \Psi \rrbracket \mathrel{-\!\!*} M\, \llbracket \Phi\, (f\, b') \rrbracket \big)\\
&\qquad \vDash M^{\mathrm{Wpbor}_M \llbracket\,\rrbracket} \Big([\alpha]_q \;\ast\; [\beta]_r \;\ast\; \exists\, y' \text{ s.t. } \big\langle \lambda\pi.\ \pi\, x = f(\pi\, y') \big\rangle.\ \llbracket \mathtt{bor}^{\alpha\sqcap\beta}_{b, y'}\ \Psi \rrbracket \Big)
\end{aligned}
$$

pobor-pbor-REBORROW

$$
\begin{aligned}
&\Big(\forall\, a' \text{ s.t. } \big\langle \lambda\pi.\ \hat{a}\, \pi = a' \big\rangle.\ \llbracket \Phi\, a' \rrbracket \mathrel{-\!\!*} M \big(\mathop{\Large\ast}\limits_{i} \big(\exists\, b' \text{ s.t. } \big\langle \lambda\pi.\ \hat{b}_i\, \pi = b' \big\rangle.\ \llbracket \Psi_i\, b' \rrbracket \big) \big) \Big) \;\ast\\
&\qquad\qquad\qquad \llbracket \mathtt{lend}^{\alpha}_{\hat{a}}\ \Phi \rrbracket \;\vDash \dot{\Rrightarrow}^{\mathrm{Wbor}_M \llbracket\,\rrbracket} \Big(\mathop{\Large\ast}\limits_{i} \llbracket \mathtt{lend}^{\alpha}_{\hat{b}_i}\ \Psi_i \rrbracket \Big)
\end{aligned}
$$

plend-SPLIT

7.4 Model

Now we present the model of our later-free prophetic borrow mechanism. The overall structure inherits RustHornBelt's model. The prophetic borrow mechanism is built on the mechanism of parametric prophecies (§ 7.4.1) as well as the prophetic agreement mechanism (§ 7.4.2), which is hidden from users of prophetic borrows. Finally, we instantiate our later-free borrow mechanism (Chapter 6) with a custom data type for syntactic propositions to achieve the later-free prophetic borrow mechanism (§ 7.4.3).

7.4.1 Parametric Prophecies

First, we introduce our model of parametric prophecies. This is based on RustHornBelt's model, but we have made improvements to simplify the model and make proof rules like PROPH-OBS-SAT work with the basic update $\dot{\Rrightarrow}$ instead of the fancy update $\Rrightarrow_{\mathcal{N}_{\mathrm{proph}}}$.

Prophecy Variables and Clairvoyant Values The set of prophecy variables PrVar_A is defined as an infinite set for non-empty A and as the empty set for $A = \varnothing$. The sets PrVar_A are disjoint for different As. We define $\mathit{PrVar} \triangleq \cup_{A \in \mathbb{U}} \mathit{PrVar}_A$.

We define the relation $\hat{a} \perp X$ between a clairvoyant value $\hat{a} \in \mathit{Clair}\, A$ and a set of prophecies $X \subseteq \mathit{PrVar}$ as follows:

$$
\hat{a} \perp X \quad \triangleq \quad \forall \pi, \pi'.\ (\forall x \notin X.\ \pi\, x = \pi'\, x) \to \hat{a}\, \pi = \hat{a}\, \pi'.
$$

This means that the clairvoyant value $\hat{a}$ does not depend on the value of any prophecies in the set X.

Prophecy Log A *prophecy log* $L \in \mathit{ProphLog}$ is a list of items of form $x := \hat{a}$, assigning a clairvoyant value $\hat{a} \in \mathit{Clair}\, A$ to a prophecy $x \in \mathit{PrVar}_A$ for $A \in \mathbb{U}$. A prophecy log represents a list of past prophecy resolutions.

The *domain* $\operatorname{dom} L \subseteq \mathit{PrVar}$ of a prophecy log $L \in \mathit{ProphLog}$ is defined as follows:

$$
\operatorname{dom} [\,] \;\triangleq\; \varnothing \qquad\qquad \operatorname{dom} \big((x := \hat{a}) : L \big) \;\triangleq\; \{x\} \cup \operatorname{dom} L
$$

It means the set of prophecies that appear in the prophecy log.

The *validity* $\checkmark L$ of a prophecy log $L \in \mathit{ProphLog}$ is defined as follows:

$$\checkmark [\,] \;\triangleq\; \top \qquad \checkmark \big((x := \hat{a}) : L\big) \;\triangleq\; x \notin \operatorname{dom} L \;\wedge\; \hat{a} \perp \big(\{x\} \cup \operatorname{dom} L\big) \;\wedge\; \checkmark L$$

The validity means that every prophecy appears at most once and that each assigned clairvoyant value $\hat{a}$ does not depend on the value of the prophecies $\{x\} \cup \operatorname{dom} L$ that have appeared so far.

The relation 'π satisfies L' $\pi \prec L$ for a prophecy assignment $\pi \in \mathit{PrAsn}$ and a prophecy log $L \in \mathit{ProphLog}$ is defined as follows:

$$\pi \prec L \quad\triangleq\quad \forall (x := \hat{a}) \in L.\ \pi\, x = \hat{a}.$$

Resource Algebra The resource algebra $\textsc{Proph}$ for the parametric prophecies is defined as follows:

$$\textsc{Proph}' \quad\triangleq\quad \mathit{PrVar} \overset{\text{fin}}{\rightharpoonup} \Big(\textsc{Unit} +_{\not\,} \textsc{Ag}\Big(\bigcup_{A \in \mathbb{U}} \mathit{Clair}\, A\Big)\Big)$$

$$\llcorner \textsc{Proph} \lrcorner \;\triangleq\; \llcorner \textsc{Proph}' \lrcorner \qquad o \cdot_{\textsc{Proph}} o' \;\triangleq\; o \cdot_{\textsc{Proph}'} o' \qquad |o|_{\textsc{Proph}} \;\triangleq\; |o|_{\textsc{Proph}'}$$

$$\checkmark_{\textsc{Proph}}\, o \;\triangleq\; \checkmark_{\textsc{Proph}'}\, o \;\wedge\; \exists L \text{ s.t. } \checkmark L.\ o \rightsquigarrow L$$

$$\begin{aligned} o \rightsquigarrow L \quad\triangleq\quad & \big(\forall x \text{ s.t. } o\,x = \mathsf{inl}\,().\ x \notin \operatorname{dom} L\big) \;\wedge \\ & \big(\forall x, \hat{a} \text{ s.t. } o\,x = \mathsf{inr}\,(\mathsf{ag}\,\hat{a}).\ (x := \hat{a}) \in L\big) \end{aligned}$$

The RA $\textsc{Proph}$ is based on the base RA $\textsc{Proph}'$, inheriting the carrier set, product and core operation.

The base RA $\textsc{Proph}'$ is the finite map RA $\overset{\text{fin}}{\rightharpoonup}$ from prophecy variables PrVar to the sum RA $+_{\not\,}$ (§ 2.2.2). The sum RA switches between the unit RA $\textsc{Unit}$, representing the unresolved state, and the agreement RA $\textsc{Ag}\,(\bigcup_{a \in A} \mathit{Clair}\, A)$, representing the resolved state with the clairvoyant value $\hat{a} \in \mathit{Clair}\, A$ of any domain $a \in A$.

The validity $\checkmark_{\textsc{Proph}}\, o$ requires aside from the original validity $\checkmark_{\textsc{Proph}'}\, o$ that there exists a valid prophecy log L satisfying $o \rightsquigarrow L$. The relation $o \rightsquigarrow L$ means that the exclusive resource $\mathsf{inl}\,()$ ensures that the prophecy has not been resolved in L and an agreement resource $\mathsf{inr}\,(\mathsf{ag}\,\hat{a})$ ensures that the prophecy has been resolved to the clairvoyant value $\hat{a}$ in L. The condition $\exists L \text{ s.t. } \checkmark L.\ o \rightsquigarrow L$ in the resource validity $\checkmark_{\textsc{Proph}}\, o$ works as an invariant for the rule PROPH-OBS-SAT.

In the original model of RustHornBelt, the RA for prophecies $\textsc{Proph}$ was modeled as an authoritative RA $\textsc{Auth}\,\textsc{Proph}''$, where the base RA $\textsc{Proph}''$ corresponds to our $\textsc{Proph}'$. The original model expressed the invariant for the rule PROPH-OBS-SAT as an Iris proposition outside the RA, which was put into an Iris invariant $\boxed{-}^{\mathcal{N}_{\text{proph}}}$. As a result, proof rules like PROPH-OBS-SAT worked only with the fancy update $\Rrightarrow_{\mathcal{N}_{\text{proph}}}$, not with the basic update $\dot{\Rrightarrow}$.

Propositions The *prophecy token* $[x] \in \mathit{iProp}$ for $x \in \mathit{PrVar}_A$ is defined as follows:[9]

$$[x] \quad\triangleq\quad \boxed{[x := \mathsf{inl}\,()]}^{\gamma_{\textsc{Proph}}}$$

Here, $\gamma_{\textsc{Proph}}$ is the ghost name for parametric prophecies.

[9] For simplicity, we do not make the prophecy token fractional $[x]_q$, because fractional prophecy tokens are not used for modeling the prophetic borrow mechanism. To use fractional prophecy tokens, we fix $\textsc{Unit}$ into $\textsc{Frac}$ in the definition of $\textsc{Proph}'$ (and slightly modify $o \rightsquigarrow L$ accordingly).

We have the following rule for *allocating* a prophecy token $[x]_q$ taking a fresh prophecy variable $x \in \mathit{PrVar}_A$:

$$\frac{A \neq \varnothing}{\vDash \dot{\Rrightarrow} \left(\exists\, x \in \mathit{PrVar}_A.\, [x] \right)} \quad \text{PROPH-ALLOC}$$

The *prophecy observation* $\langle \hat{\phi} \rangle$ is modeled as follows:

$$\langle \hat{\phi} \rangle \quad \triangleq \quad \exists L \text{ s.t. } (\forall \pi \prec L.\ \hat{\phi}\,\pi). \quad \mathop{\LARGE\ast}_{(x:=\hat{a}) \,\in\, L} \boxed{x := \mathsf{inr}\,(\mathsf{ag}\,\hat{a})}^{\gamma_{\mathrm{PROPH}}}$$

It owns the agreement resource $\boxed{[x := \mathsf{inr}\,(\mathsf{ag}\,\hat{a})]}^{\gamma_{\mathrm{PROPH}}}$ for each past resolution $x := \hat{a}$ recorded in the prophecy log L and also asserts that any prophecy assignment π satisfying L satisfies the clairvoyant assertion $\hat{\phi}$. The prophecy observation satisfies the rules presented in § 7.2.1.

We have the following rule for resolving a prophecy x to a value $f(\overline{\pi\,y})$ that depend on unresolved prophecies $\bar{y}$:[10]

$$[x] \ * \ \mathop{\LARGE\ast} \overline{[y]} \ \vDash \dot{\Rrightarrow} \left(\mathop{\LARGE\ast} \overline{[y]} \ * \ \left\langle \lambda\pi.\ \pi\,x = f(\overline{\pi\,y}) \right\rangle \right) \quad \text{PROPH-RESOLVE}$$

We consume the prophecy token $[x]$ and get the prophecy observation $\langle \lambda\pi.\ \pi\,x = f(\overline{\pi\,y}) \rangle$, using the prophecy tokens of the dependencies $\mathop{\ast} \overline{[y]}$ as a catalyst. This rule can be proved using the rule OWN-$\rightsquigarrow$ (§ 2.2.1). To prove the resource update $\rightsquigarrow$, we add a new item $x := \lambda\pi.\, f(\overline{\pi\,y})$ to the head of the prophecy log L of the validity predicate $\checkmark_{\mathrm{PROPH}}$.

Note that the side condition that the dependencies $\bar{y}$ have not been resolved is essential to the rule PROPH-RESOLVE. Omitting it causes a paradox immediately. For example, suppose we own two prophecy tokens $[x]$, $[y]$ for integer prophecies $x, y \in \mathit{PrVar}_{\mathbb{Z}}$. If we had a variant of PROPH-RESOLVE that does not require the dependency prophecy tokens, then we could consume $[x]$ to get the prophecy observation $\langle \lambda\pi.\ \pi\,x = \pi\,y \rangle$ and consume $[y]$ to get the prophecy observation $\langle \lambda\pi.\ \pi\,y = \pi\,x + 1 \rangle$. Combining the two by PROPH-OBS-$*$ and modifying the result by PROPH-OBS-MONO, we get $\langle \lambda\_.\ \bot \rangle$, which causes a contradiction by the adequacy of the prophecy observation PROPH-OBS-CONST.

7.4.2 Prophetic Agreement

We introduce the mechanism of *prophetic agreement*, which is hidden from users of prophetic borrows. The idea of prophetic agreement comes from RustHornBelt, but our model is simpler.

Resource Algebra The resource algebra PrAg for the prophetic agreement is defined as follows:

$$\textsc{PrAg} \quad \triangleq \quad \textsc{Frac} \times \textsc{Ag}\left(\bigcup_{A \in \mathbb{U}} A \right)$$

It is the product RA of the fractional RA Frac and the agreement RA Ag over a value $\sum_{A \in \mathbb{U}} A$ of some domain $A \in \mathbb{U}$ (§ 2.2.2).

[10] If we allow fractional prophecy tokens, then we can generalize $[y_i]$ into any fraction $[y_i]_q$ in this rule PROPH-RESOLVE.

Value Observer We introduce an Iris proposition called the *value observer* $\mathrm{vo}^\gamma\, a \in \mathit{iProp}$ for a ghost name $\gamma \in \mathit{GhostName}$ and a value $a \in A$ of some domain $A \in \mathbb{U}$, which is defined as follows:

$$\mathrm{vo}^\gamma\, a \quad \triangleq \quad \left[(1/2,\ \mathrm{ag}\, a)\right]^{\gamma}_{\textsc{PrAg}}$$

It simply has the half ownership of the value a for the ghost name γ.

The value observer satisfies the following rules:

$$\vDash \dot{\Rrightarrow} \left(\exists \gamma.\ \ \mathrm{vo}^\gamma\, a \ * \ \mathrm{vo}^\gamma\, a\right) \quad \textsc{vo}^2\textsc{-alloc}$$

$$\mathrm{vo}^\gamma\, a \ * \ \mathrm{vo}^\gamma\, a' \ \vDash \ a = a' \quad \textsc{vo}^2\textsc{-agree}$$

$$\mathrm{vo}^\gamma\, a \ * \ \mathrm{vo}^\gamma\, a \ \vDash \dot{\Rrightarrow} \left(\mathrm{vo}^\gamma\, a' \ * \ \mathrm{vo}^\gamma\, a'\right) \quad \textsc{vo}^2\textsc{-update}$$

$$\mathrm{vo}^\gamma\, a \ * \ \mathrm{vo}^\gamma\, a \ * \ \mathrm{vo}^\gamma\, a \ \vDash \ \bot \quad \textsc{vo}^3\text{-}\bot$$

We can always allocate two value observers of a fresh ghost name γ ($\textsc{vo}^2\textsc{-alloc}$). Two value observers of the same ghost name agree on the value ($\textsc{vo}^2\textsc{-agree}$). We can update the value of the two value observers of the same ghost name ($\textsc{vo}^2\textsc{-update}$). There should not exist three value observers of the same ghost name ($\textsc{vo}^3\text{-}\bot$).

Prophecy Controller Based on the value observer, we introduce an Iris proposition called the *prophecy controller* $\mathrm{pc}^\gamma_x\, a$ of a ghost name $\gamma \in \mathit{GhostName}$, a value $a \in A$, and a *prophecy* $x \in \mathit{PrVar}_A$ for some domain $A \in \mathbb{U}$, which is defined as follows:

$$\mathrm{pc}^\gamma_x\, a \quad \triangleq \quad \left([x] \ * \ \mathrm{vo}^\gamma\, a\right) \ \lor \ \Big(\left\langle \lambda\pi.\ \pi\, x = a\right\rangle \ * \ \exists a'.\ \mathrm{vo}^\gamma\, a' \ * \ \mathrm{vo}^\gamma\, a'\Big) \qquad (7.9)$$

The first disjunct represents the case where the prophecy x has not been resolved yet. It exclusively asserts that the prophecy x is unresolved (by the prophecy token $[x]$) and asserts with half ownership that the value assigned to the ghost name γ is a (by the value observer $\mathrm{vo}^\gamma\, a$). The second disjunct represents the case where the prophecy x has been resolved. It asserts that the value of the prophecy $\pi\, x$ should be a (by the prophecy observation $\langle \lambda\pi.\ \pi\, x = a\rangle$) and also owns two value observers for the ghost name $\mathrm{vo}^\gamma\, a' \ * \ \mathrm{vo}^\gamma\, a'$.

This model (7.9) of the prophecy controller is much simpler than RustHornBelt's model, which used an involved proposition called the 'prophecy equalizer' to handle dependent prophecies. We can use this simple model because we consider *nested prophecies* in our prophetic borrow mechanism (Remark 7.1, § 7.2.3).

Proof Rules We can create a prophecy controller $\mathrm{pc}^\gamma_x\, a$ from a value observer $\mathrm{vo}^\gamma\, a$ and a prophecy token $[x]$:

$$\mathrm{vo}^\gamma\, a \ * \ [x] \ \vDash \ \mathrm{pc}^\gamma_x\, a \quad \textsc{vo-proph-pc}$$

So combining this with $\textsc{proph-alloc}$ and $\textsc{vo}^2\textsc{-alloc}$, we can create a value observer and a prophecy controller:[11]

$$\vDash \dot{\Rrightarrow}\left(\exists x, \gamma.\ \ \mathrm{vo}^\gamma\, a \ * \ \mathrm{pc}^\gamma_x\, a\right) \quad \textsc{vo-pc-alloc}$$

Also, a prophecy controller decomposes into a value observer and a prophecy token under a value observer, because the second disjunct of (7.9) is rejected by the rule $\textsc{vo}^3\text{-}\bot$:

$$\mathrm{vo}^\gamma\, a \ * \ \mathrm{pc}^\gamma_x\, a \ \vDash \ \mathrm{vo}^\gamma\, a \ * \ \mathrm{vo}^\gamma\, a \ * \ [x] \quad \textsc{vo-pc-proph}$$

[11] We do not need the side condition $A \neq \varnothing$ for the implicit domain $A \in \mathbb{U}$, because the value $a \in A$ ensures the non-emptiness of A.

The value observer and the prophecy controller of the same ghost name agree on the value, by VO²-AGREE:

$$\mathsf{vo}^{\gamma}\, a \;*\; \mathsf{pc}^{\gamma}_{x}\, a' \;\vDash\; a = a' \qquad \textsc{vo-pc-agree}$$

We can update the value of the value observer and the prophecy controller of the same ghost name, by VO²-UPDATE:

$$\mathsf{vo}^{\gamma}\, a \;*\; \mathsf{pc}^{\gamma}_{x}\, a \;\vDash\dot{\Rrightarrow}\; \big(\mathsf{vo}^{\gamma}\, a' \;*\; \mathsf{pc}^{\gamma}_{x}\, a'\big) \qquad \textsc{vo-pc-update}$$

When we have a value observer $\mathsf{vo}^{\gamma}\, a$ and a prophecy controller $\mathsf{pc}^{\gamma}_{x}\, a$, we can resolve the prophecy x to the value a, by getting rid of the value observer $\mathsf{vo}^{\gamma}\, a$:

$$\mathsf{vo}^{\gamma}\, a \;*\; \mathsf{pc}^{\gamma}_{x}\, a \;\vDash\dot{\Rrightarrow}\; \big(\big\langle \lambda\pi.\ \pi\, x = a \big\rangle \;*\; \mathsf{pc}^{\gamma}_{x}\, a\big) \qquad \textsc{vo-pc-resolve}$$

To prove this, we switch from the first disjunct to the second disjunct in the prophecy controller (7.9).

Enriching VO-PC-RESOLVE, we can also *partially resolve* the prophecy x of a prophecy controller by the following rule:

$$\begin{gathered}\mathsf{vo}^{\gamma}\, a \;*\; \mathsf{pc}^{\gamma}_{x}\, a \;*\; \mathop{\Huge *}\overline{[y]} \;\vDash\dot{\Rrightarrow} \\ \Big(\mathop{\Huge *}\overline{[y]} \;*\; \big\langle \lambda\pi.\ \pi\, x = f(\overline{\pi\, y})\big\rangle \;*\; \big(\forall\, \bar{b}' \text{ s.t. } \big\langle \lambda\pi.\ \forall i.\ \pi\, y_i = b'_i\big\rangle.\ \mathsf{pc}^{\gamma}_{x}\, (f(\bar{b}'))\big)\Big) \\ \textsc{vo-pc-preresolve}\end{gathered}$$

We partially resolve the prophecy x into a value $f(\overline{\pi\, y})$ that depends on unresolved prophecies $\bar{y}$, which can be proved by PROPH-RESOLVE. We get the 'promise' to get the prophecy controller $\mathsf{pc}^{\gamma}_{x}\, (f(\bar{b}'))$ for any values $\bar{b}'$ that the dependent prophecies are resolved to. This promise can be constructed from the two value observers $\mathsf{vo}^{\gamma}\, a \,*\, \mathsf{vo}^{\gamma}\, a$ and the prophecy observation $\big\langle \lambda\pi.\ \pi\, x = f(\overline{\pi\, y})\big\rangle$. This rule is used for proving *prophetic borrow subdivision* (POBOR-SUBDIV, POBOR-MERGE-SUBDIV) of our prophetic borrow mechanism (§ 7.2.2).

Also, by consuming a prophecy observer $\mathsf{pc}^{\gamma}_{x}\, a$, we can get a prophecy observation $\big\langle \lambda\pi.\ \pi\, x = a\big\rangle$:

$$\mathsf{pc}^{\gamma}_{x}\, a \;\vDash\dot{\Rrightarrow}\; \big\langle \lambda\pi.\ \pi\, x = a\big\rangle \qquad \textsc{pc-resolve}$$

If the prophecy controller is the first disjunct of (7.9), we resolve the prophecy by consuming its prophecy token. Otherwise, we just have the prophecy observation inside the second disjunct.

7.4.3 Prophetic Borrows

Now we instantiate our later-free borrows Chapter 6 to achieve later-free prophetic borrows.

Custom Syntactic Propositions First, the *custom data type of syntactic propositions* $\mathit{nProp}^{\mathrm{pbor}}$ passed to the borrow mechanism is constructed as follows, based on the data type *nProp* passed to the prophetic borrow mechanism by the user:

$$\begin{aligned}\mathit{nProp}^{\mathrm{pbor}} \ni P^{*} \;::=\;& \mathtt{xbor}^{\gamma}_{x}\, \Phi \quad (\gamma \in \mathit{GhostName};\ x \in \mathit{PrVar}_A;\ \Phi : A \to \mathit{nProp};\ A \in \mathbb{U}) \\ \mid\;& \mathtt{xlend}_{\hat{a}}\, \Phi \quad (\hat{a} \in \mathit{Clair}\, A;\ \Phi : A \to \mathit{nProp};\ A \in \mathbb{U}) \\ \mid\;& \mathtt{xrebor}^{\gamma_o,\gamma_i,\gamma_+}_{x}\, f \quad (\gamma_o, \gamma_i, \gamma_+ \in \mathit{GhostName};\ x \in \mathit{PrVar}_A;\ f : B \to A;\ A, B \in \mathbb{U})\end{aligned}$$

The constructor $\mathtt{xbor}^{\gamma}_{x}\, \Phi$ is used for modeling *prophetic borrowers*. The constructor $\mathtt{xlend}_{\hat{a}}\, \Phi$ is used for modeling *prophetic lenders*. The constructor $\mathtt{xrebor}^{\gamma,\gamma',\gamma_+}_{x}\, f$ is used for an intermediate borrow used for proving the *prophetic reborrowing* rule POBOR-PBOR-REBORROW. The function $f\colon B \longrightarrow A$ is a parameter of this rule. Note that the user of prophetic borrows does not need to know this data type $\mathit{nProp}^{\mathrm{pbor}}$. In particular, the user does not need to directly manipulate the involved intermediate borrow by $\mathtt{xrebor}_x^{\gamma,\gamma',\gamma_+}\, f$.

Now the *resource algebra* $\textsc{Pbor}_{\mathit{nProp}}$ for our prophetic borrows is defined as follows, instantiating the resource algebra $\textsc{Bor}_{-}$ for the later-free borrows with this custom data type $\mathit{nProp}^{\mathrm{pbor}}$:

$$\textsc{Pbor}_{\mathit{nProp}} \quad \triangleq \quad \textsc{Bor}_{\mathit{nProp}^{\mathrm{pbor}}}$$

Intepretation of Custom Syntactic Propositions Given the interpretation $[\![\ ]\!]\colon \mathit{nProp} \longrightarrow \mathit{iProp}$ of the base syntactic propositions nProp from the user, we can construct the *interpretation* for our custom syntactic propositions $[\![\ ]\!]^{\mathrm{pbor}}\colon \mathit{nProp}^{\mathrm{pbor}} \longrightarrow \mathit{iProp}$ as follows:

$$[\![\mathtt{xbor}_x^{\gamma}\ \Phi]\!]^{\mathrm{pbor}} \quad \triangleq \quad \exists a.\ \ \mathsf{pc}_x^{\gamma}\, a \,*\, [\![\Phi\ a]\!] \tag{7.10}$$

$$[\![\mathtt{xlend}_{\hat{a}}\ \Phi]\!]^{\mathrm{pbor}} \quad \triangleq \quad \exists\, a' \text{ s.t. } \langle \lambda\pi.\ \hat{a}\,\pi = a' \rangle.\ [\![\Phi\ a']\!] \tag{7.11}$$

$$[\![\mathtt{xrebor}_x^{\gamma_o,\gamma_i,\gamma_+}\ f]\!]^{\mathrm{pbor}} \quad \triangleq \quad \exists b.\ \ \mathsf{pc}_x^{\gamma_o}\,(f\ b) \,*\, \mathsf{vo}^{\gamma_i}\ b \,*\, \mathsf{vo}^{\gamma_+}\ b \tag{7.12}$$

The *world satisfaction* for our prophetic borrows $\mathsf{Wpbor}_M\,[\![\ ]\!]$ for the intepretation $[\![\ ]\!]\colon \mathit{nProp} \longrightarrow \mathit{iProp}$ is defined as follows, instantiating the world satisfaction for borrows $\mathsf{Wbor}_M\ -$ with this custom interpretation $[\![\ ]\!]^{\mathrm{pbor}}$:

$$\mathsf{Wpbor}_M\,[\![\ ]\!] \quad \triangleq \quad \mathsf{Wbor}_M\,[\![\ ]\!]^{\mathrm{pbor}}$$

Prophetic Lender The *prophetic lender token* $\mathsf{lend}_{\hat{a}}^{\alpha}\ \Phi$ is modeled as follows, using the original lender token lend^{α}:

$$\mathsf{lend}_{\hat{a}}^{\alpha}\ \Phi \quad \triangleq \quad \mathsf{lend}^{\alpha}\,(\mathtt{xlend}_{\hat{a}}\ \Phi).$$

By the interpretation (7.11), this lends $\exists\, a'$ s.t. $\langle \lambda\pi.\ \hat{a}\,\pi = a' \rangle.\ [\![\Phi\ a']\!]$, the content $[\![\Phi\ a']\!]$ of the final value a' that is linked with the clairvoyant value $\hat{a}'$ by the prophecy observation $\langle \lambda\pi.\ \hat{a}\,\pi = a' \rangle$.

The rules PLEND-LFT, PLEND-RETRIEVE, PLEND-SPLIT for prophetic borrowers are immediately derived from the rules LEND-LFT, LEND-RETRIEVE, LEND-SPLIT for original lenders.

Prophetic Borrower The *prophetic borrower token* $\mathsf{bor}_{a,x}^{\alpha}\ \Phi$ is modeled as follows, using the original borrower token bor^{α}:

$$\mathsf{bor}_{a,x}^{\alpha}\ \Phi \quad \triangleq \quad \dagger\alpha \ \lor\ \exists\gamma.\ \ \mathsf{vo}^{\gamma}\ a \,*\, \mathsf{bor}^{\alpha}\,(\mathtt{xbor}_x^{\gamma}\ \Phi).$$

The first disjunct $\dagger\alpha$ is just for the rule PBOR-FAKE. The main part owns the *value observer* $\mathsf{vo}^{\gamma}\ a$ outside the borrow and by (7.10) borrows

$$\exists a'.\ \ \mathsf{pc}_x^{\gamma}\, a' \,*\, [\![\Phi\ a']\!],$$

the content $[\![\Phi\ a']\!]$ and the *prophecy controller* $\mathsf{pc}_x^{\gamma}\ a'$ for the content's value a' and the prophecy x.

When the prophetic borrower opens the borrow (PBOR-OPEN), it knows that the value a' of the content is equal to the value observer's value a by the rule VO-PC-AGREE.

Prophetic Open Borrower The *prophetic open borrower token* $\mathsf{obor}^{\alpha}_{q,x}\ \Phi$ is modeled as follows, using the original open borrower token obor^{α}:

$$\mathsf{obor}^{\alpha}_{q,x}\ \Phi \quad \triangleq \quad \exists\gamma.\ \mathsf{obor}^{\alpha}_{q}\left(\mathsf{xbor}^{\gamma}_{x}\ \Phi\right)\ *\ \exists a.\ \mathsf{vo}^{\gamma}\ a\ *\ \mathsf{pc}^{\gamma}_{x}\ a.$$

It owns the open borrower token $\mathsf{obor}^{\alpha}_{q}\left(\mathsf{xbor}^{\gamma}_{x}\ \Phi\right)$ with the value observer and the prophecy controller.

When it closes the borrow with an updated value (POBOR-CLOSE), it updates the value of the value observer and the prophecy controller by VO-PC-UPDATE.

For prophetic borrow subdivision (POBOR-SUBDIV, POBOR-MERGE-SUBDIV), it *partially resolves the prophecy x* by VO-PC-PRERESOLVE. The converter for ordinary borrow subdivision (OBOR-SUBDIV, OBOR-MERGE-SUBDIV) can be constructed using the promise for the prophecy controller $\forall\, a'$ s.t. $\langle \lambda\pi.\ f(\overline{\pi\, y}) = a' \rangle.\ \mathsf{pc}^{\gamma}_{x}\ a'$ provided by the rule VO-PC-PRERESOLVE.

Proof of Prophetic Reborrowing Now we are ready to prove the rule for prophetic reborrowing POBOR-PBOR-REBORROW. The proof is tricky, using an intermediate borrow over $\mathsf{xrebor}^{\gamma_o,\gamma_i,\gamma_+}_{x}\ f$. A key advantage of our prophetic borrows is that the tricky proof of prophetic reborrowing is *encapsulated* as a *reusable proof rule* POBOR-PBOR-REBORROW for the user.

Proof of POBOR-PBOR-REBORROW. First, we decompose the inner prophetic borrower token $\mathsf{bor}^{\beta}_{b,y}\ \Psi$ into the ordinary borrower token $\mathsf{bor}^{\beta}_{y}\left(\mathsf{xbor}^{\gamma_i}_{y}\ \Psi\right)$ and the value observer $\mathsf{vo}^{\gamma_i}\ b$ for the inner ghost name γ_i.

Then we *reborrow* the inner ordinary borrower token (BOR-REBORROW) under the outer lifetime α to get the reborrower

$$\mathsf{bor}^{\alpha\sqcap\beta}_{y}\left(\mathsf{xbor}^{\gamma_i}_{y}\ \Psi\right) \tag{7.13}$$

and the promise to get the inner ordinary borrower

$$\dagger\alpha\ \twoheadrightarrow*\ \mathsf{bor}^{\beta}_{y}\left(\mathsf{xbor}^{\gamma_i}_{y}\ \Psi\right) \tag{7.14}$$

Next, we decompose the outer prophetic open borrower token $\mathsf{obor}^{\alpha}_{q,x}\ \Phi$ into the ordinary open borrower token $\mathsf{obor}^{\alpha}_{q}\left(\mathsf{xbor}^{\gamma_o}_{x}\ \Psi\right)$, the value observer $\mathsf{vo}^{\gamma_o}\ a$ and the prophecy controller $\mathsf{pc}^{\gamma_o}_{x}\ a$ for the outer ghost name γ_o. By VO-PC-UPDATE, we update the value of the outer value observer and the outer prophecy controller to $f\ b$.

We also allocate two auxiliary value observers $\mathsf{vo}^{\gamma_+}\ b, \mathsf{vo}^{\gamma_+}\ b$ for a fresh ghost name γ_+ by VO^2-ALLOC.

Then we *subdivide* the outer ordinary open borrower token $\mathsf{obor}^{\alpha}_{q}\left(\mathsf{xbor}^{\gamma_o}_{x}\ \Psi\right)$ by OBOR-SUBDIV to create the following intermediate borrow:

$$\mathsf{bor}^{\alpha\sqcap\beta}\left(\mathsf{xrebor}^{\gamma_o,\gamma_i,\gamma_+}_{x}\ f\right) \tag{7.15}$$

We can get the resource $[\![\mathsf{xrebor}^{\gamma_o,\gamma_i,\gamma_+}_{x}\ f]\!]$ (7.12) out of the outer prophecy controller $\mathsf{pc}^{\gamma_o}_{x}(f\ b)$, the inner value observer $\mathsf{vo}^{\gamma_i}\ b$, and one of the auxiliary value observers $\mathsf{vo}^{\gamma_+}\ b$. We can construct the converter for this subdivision by combining the promise (7.14) supplied by the ordinary reborrow and the converter

$$\forall b'.\ \dagger\alpha\ *\ \mathsf{bor}^{\beta}_{b',y}\ \Psi\ \twoheadrightarrow*\ M\,[\![\Phi\,(f\ b')]\!]$$

supplied to this rule POBOR-PBOR-REBORROW. We also get a live lifetime token $[\alpha]_q$ for the outer lifetime.

After that, we open the obtained intermediate borrower (7.15). We get the open borrower token $\mathsf{obor}_s^{\alpha\sqcap\beta}\,(\mathsf{xrebor}_x^{\gamma_o,\gamma_i,\gamma_+}\,f)$ for some fraction $s \in \mathbb{Q}_{>0}$. Also, we recover the outer prophecy controller $\mathsf{pc}_x^{\gamma_o}\,(f\,b)$, the inner value observer $\mathsf{vo}^{\gamma_i}\,b$ and the auxiliary value observer $\mathsf{vo}^{\gamma_+}\,b$. Here, we used the agreement VO^2-AGREE of auxiliary value observers to see that the value existentially quantified in the content $[\![\mathsf{xrebor}_x^{\gamma_o,\gamma_i,\gamma_+}\,f]\!]$ is equal to b.[12]

We also open the reborrower (7.13) of the inner borrow to get the inner prophecy controller $\mathsf{pc}^{\gamma_i}\,b$, the content $[\![\Psi\,b]\!]$, and an open borrower token $\mathsf{bor}_y^{\alpha\sqcap\beta}\,(\mathsf{xbor}_y^{\gamma_i}\,\Psi)$ for some fraction $s' \in \mathbb{Q}_{>0}$.

Moreover, we allocate a new couple of a value observer $\mathsf{vo}^{\gamma_i'}\,b$ and a prophecy controller $\mathsf{pc}_{y'}^{\gamma_i'}\,b$ for a fresh prophecy y' and a fresh ghost name γ_i' by VO-PC-ALLOC. Then by VO-PC-PRERESOLVE, we *partially resolve the prophecy* x of the outer prophecy controller $\mathsf{pc}_x^{\gamma_o}\,(f\,b)$ to get the desired prophecy observation $\langle \lambda\pi.\ \pi\,x = f\,(\pi\,y')\rangle$ and the promise to get the outer prophecy controller:

$$\forall\, b' \text{ s.t. } \langle \lambda\pi.\ \pi\,y' = b' \rangle.\ \ \mathsf{pc}_x^{\gamma_o}\,(f\,b') \tag{7.16}$$

Finally, we *merge and subdivide* the two open borrower tokens by OBOR-MERGE-SUBDIV to get the following ordinary borrower token for the output prophetic borrower token:

$$\mathsf{bor}^{\alpha\sqcap\beta}\,(\mathsf{xbor}_{y'}^{\gamma_i'}\,\Psi)$$

We construct $[\![\mathsf{xbor}_{y'}^{\gamma_i'}\,\Psi]\!]$ (7.10) by the content $[\![\Psi\,b]\!]$ and the prophecy controller $\mathsf{pc}_{y'}^{\gamma_i'}\,b$.

We construct the converter for this merger-subdivision out of the promise to get the outer prophecy controller (7.16), the inner value observer $\mathsf{vo}^{\gamma_i}\,b$ and prophecy controller $\mathsf{pc}_y^{\gamma_i}\,b$, and the auxiliary value observers $\mathsf{vo}^{\gamma_+}\,b, \mathsf{vo}^{\gamma_+}\,b$. To feed the promise (7.16) with the prophecy observation $\langle \lambda\pi.\ \pi\,y' = b' \rangle$, we consume the prophecy controller $\mathsf{pc}_{y'}^{\gamma_i'}\,b'$ from the output borrower's final content by the rule PC-RESOLVE. Also, we update the value of the inner value observer $\mathsf{vo}^{\gamma_i}\,b$ and prophecy controller $\mathsf{pc}_y^{\gamma_i}\,b$ to the final value b' of the output reborrower by VO-PC-UPDATE. For this update, it is essential that *both* the inner value observer and prophecy controller are stored inside this converter; the *merger* of the reborrower (7.13) and the intermediate borrow (7.15) has made it possible to store both inside the converter.

Combining the resulting new borrower token $\mathsf{bor}^{\alpha\sqcap\beta}\,(\mathsf{xbor}_{y'}^{\gamma_i'}\,\Psi)$ and the value observer $\mathsf{vo}^{\gamma_i'}\,b$, we get the desired output prophetic borrower token $\mathsf{bor}_{b,y'}^{\alpha\sqcap\beta}\,\Psi$. □

[12] If the conversion function f is *injective*, then we can omit these auxiliary value observers, because the agreement between the outer value observer and prophecy controller gives the equality $f\,b = f\,b'$ (where b' is the value existentially quantified in $[\![\mathsf{xrebor}_x^{\gamma_o,\gamma_i,\gamma_+}\,f]\!]$), which implies $b = b'$ if f is injective. RustHornBelt's original reborrow proofs did not use the auxiliary value observers because the conversion functions f they considered were injective. Although the injectivity is a reasonable assumption, we chose not to assume it for the rule POBOR-PBOR-REBORROW aiming at more generality.

Chapter 8

Our Coq Mechanization

On mechanical slavery, on the slavery of the machine,
the future of the world depends.

Oscar Wilde, *The Soul of Man Under Socialism*

This chapter reports on our mechanization of the Nola framework in the Coq Proof Assistant. Section 8.1 gives an overview of our Coq mechanization of Nola. Section 8.2 illustrates how to use our Coq mechanization in more detail. Section 8.3 revisits the case study of linked list mutation in § 3.3, elaborating how verification goes in our Coq mechanization of Nola.

8.1 Overview

Availability The source code of our mechanization of the Nola framework is publicly available at the GitHub repository `https://github.com/hopv/nola`. In particular, the version for this dissertation is provided in the branch `phd-thesis`.

Architecture Our Coq development is built on top of the Coq development of the *Iris separation logic framework* (Iris Team, 2023b), which provides various useful definitions, lemmas, class instances and proof tactics for interactive proof (Krebbers et al., 2017b, 2018). Also, our development depends on the std++ library (std++ Team, 2023), a general-purpose Coq library on which Iris also depends.

Our Core Achievements We have mechanized the later-free mechanisms for two central types of propositional sharing, *invariants* (§ 3.2) and *borrows* (Chapter 6). Our mechanisms are parameterized over the choice of the data type for propositions *nProp* and the semantic interpretation $[\![\ ]\!] \colon \mathit{nProp} \to \mathit{iProp}$.

We have also mechanized a general library for the derivability technique presented in Chapter 4, which can be used for semantically altering the content propositions of the logical connectives for propositional sharing.

Extended Weakest Precondition Predicates For program verification in the style of Nola, we have developed a new general Iris library for the *extended Hoare triples* $\{P\}\, e\, \{\Psi\}^{W}_{\mathcal{E}}$, $[P]\, e\, [\Psi]^{W}_{\mathcal{E}}$ and, more fundamentally, the *extended weakest precondition predicates* $\mathrm{pwp}\ e\, \{\Psi\}^{W}_{\mathcal{E}}$, $\mathrm{twp}\ e\, [\Psi]^{W}_{\mathcal{E}}$, which enjoys a *custom world satisfaction* $W \in$ *iProp* (§ 3.2.1).

Because Iris's existing library for the weakest precondition predicates is huge and well-developed, we have reused that library as much as possible, not re-implementing the whole library with the extended fancy update modality ${}_{\mathcal{E}}{\Rrightarrow}^{W}_{\mathcal{E}'}$.

To attain this goal, we have repurposed the *state interpretation*, an Iris predicate over the global mutable state of the target low-level language, such as the heap memory state H. Iris's weakest precondition predicate library is parameterized over the choice of the target language and its semantic characterization given by the class `irisGS_gen`, which includes the state interpretation `state_interp`.

Hence, we have defined our extended weakest precondition predicates as Iris's original weakest precondition predicates instantiated with the semantic language characterization `irisGS_gen` whose state interpretation `state_interp` is set to $\lambda\sigma.\ W * \Phi\,\sigma$, the separating conjunction of the custom world satisfaction W and the originally intended state interpretation Φ.[1]

This design choice enables reusing the existing lemmas and class instances from Iris's library for the weakest precondition predicates. We also newly developed lemmas and class instances regarding the custom world satisfaction W, such as eliminating the extended fancy update (like HOAREW-$\mathrel{\mid\!\Rrightarrow}$W) and expanding the custom world satisfaction (like HOAREW-EXPAND).

For Nola-style verification, one should prepare lemmas, instances and tactics for the extended weakest precondition predicates on a specific target language (e.g., Iris's HeapLang). This can be done by slightly enriching the existing lemmas, instances and tactics for the original weakest precondition predicates with the custom world satisfaction parameter W.

Examples and Paradoxes Using these general libraries, we have mechanized the verification examples discussed in this dissertation, including the iteration over shared mutable lists (§ 3.3.3), strong normalization of the stratified type system (Chapter 5), and the mechanism of prophetic borrows (Chapter 7). To encode variable binding for second-order quantifiers and recursive propositions/types, we have adopted *typed de Bruijn indexing*, with non-trivial efforts regarding variable substitution. We have also used the derivability technique of Chapter 4 for semantic alteration.

We have also mechanized the paradox of the later-eliminating total weakest precondition ((1.17), § 1.4) and the new simple paradox of later-free invariants (Theorem 3.9, § 3.4.1).

Axioms Remarkably, our Coq development is free of axioms except for the functional extensionality axiom.

We do not depend on any of Axiom K, the uniqueness of identity proofs (UIP) axiom, and the proof irrelevance axiom, which are known to be inconsistent with *homotopy* or *cubical type theories*. Although we have used dependent types in a non-trivial way, especially for typed de Bruijn indexing, we managed to avoid these axioms.

8.2 How to Use Our Coq Mechanization

Here we illustrate how to use our Coq mechanization of Nola.

[1] To be precise, the weakest precondition predicates $\mathrm{pwp}\ e\ \{\Psi\}^{*W}_{\mathcal{E}}$, $\mathrm{twp}\ e\ [\Psi]^{*W}_{\mathcal{E}}$ thus defined by 'hacking' the state interpretation (we mark them with $*$ here) do not change the fancy update modality put on the postcondition for the disjunct of the value case ((3.5), (3.6) in § 3.2.1). As a result, the absorption rules like $\mathrel{\mid\!\Rrightarrow}$W-HOAREW and HOAREW-$\mathrel{\mid\!\Rrightarrow}$W hold only for *non-value expressions*. This is a minor issue, as we usually do not consider the weakest preconditions for values. Also, the genuine extended weakest precondition predicates $\mathrm{pwp}\ e\ \{\Psi\}^{W}_{\mathcal{E}}$, $\mathrm{twp}\ e\ [\Psi]^{W}_{\mathcal{E}}$ can be obtained by putting the extended fancy update modality $\mathrel{\mid\!\Rrightarrow}^{W}_{\mathcal{E}}$ on the postcondition of our 'hacking' version: $\mathrm{pwp}\ e\ \{\lambda v.\ \mathrel{\mid\!\Rrightarrow}^{W}_{\mathcal{E}}\ \Psi\,v\}^{*W}_{\mathcal{E}}$, $\mathrm{twp}\ e\ [\lambda v.\ \mathrel{\mid\!\Rrightarrow}^{W}_{\mathcal{E}}\ \Psi\,v]^{*W}_{\mathcal{E}}$.

Installation In the GitHub repository https://github.com/hopv/nola, we publish our Coq source code for Nola as a package coq-nola of opam, the standard package manager for OCaml. Nola's package depends on the Coq proof assistant (written by OCaml) and the Iris framework (written by Coq). You can install the dependencies for Nola and build it using opam. For that, you first register to your opam the repositories for Coq and Iris:

```
opam repo add coq-released https://coq.inria.fr/opam/released
opam repo add iris-dev https://gitlab.mpi-sws.org/iris/opam.git
```

Then you can install to your opam the Nola package, automatically installing the dependencies:

```
opam install PATH-TO-NOLA
```

Source Code Nola's source code consists of three parts. The path nola.iris contains Nola's core libraries built on the Iris framework. The path nola.util contains general-purpose utilities, extending the functionalities of the std++ library. The path nola.examples contains the verification examples for Nola, including those discussed in this dissertation. You can use our Coq mechanization of Nola for your verification projects by importing relevant modules in these paths.

Quick Guide: How to Use Nola's Invariant Now we present a quick guide on how to use Nola's invariant.

First you import the inv module in nola.iris, e.g. by adding the following line:

```
From nola.iris Require Import inv.
```

Then you should prepare some data type nProp of syntactic separation logic propositions (typeset as *nProp* in the dissertation).

Then add to the context the ghost state type class ninvGS nProp Σ for Nola's invariant.[2] A simple way is to add the following line inside the section where you use Nola's invariant:

```
Context `{!ninvGS nProp Σ}.
```

Now you get the *invariant token* inv_tok N P : iProp Σ (typeset as $\mathrm{inv}^{\mathcal{N}}\, P$ in the dissertation):

```
inv_tok `{!ninvGS nProp Σ} : namespace → nProp → iProp Σ
```

You also get the *world satisfaction* inv_wsat intp : iProp Σ (typeset as Winv in the dissertation) for the invariant, parameterized over the semantic interpretation intp : nProp → iProp Σ:

```
inv_wsat `{!ninvGS nProp Σ} : (nProp → iProp Σ) → iProp Σ
```

Now you should construct the semantic interpretation intp : nProp → iProp Σ of your syntactic propositions nProp (typeset as $[\![\,]\!] : nProp \to iProp$ in the dissertation), which typically depends on inv_tok to support nested invariants.

Then you can use proof rules on Nola's invariant machinery for the fancy update extended with the world satisfaction inv_wsat intp. For example, we provide the following rules corresponding to INV-ALLOC and INV-ACC-CH:

```
Lemma inv_tok_alloc {intp} P N :
  intp P =[inv_wsat intp]=∗ inv_tok N P.
```

[2] We add the prefix n for Nola because the name invGS is already used in Iris for its original invariant.

```
Lemma inv_tok_acc {intp N E P} :
  ↑N ⊆ E → inv_tok N P =[inv_wsat intp]{E,E\↑N}=∗
    intp P ∗ (intp P =[inv_wsat intp]{E\↑N,E}=∗ True).
```

To verify programs, you need proof rules and tactics that work with the extended Hoare triples on the target language. In `nola.examples.heap_lang`, we provide such things for Iris's HeapLang, a simple ML-like language with heap memory operations.

8.3 Case Study: Linked List Mutation

Here we revisit the case study of linked list mutation in § 3.3, elaborating how verification goes in our Coq mechanization of Nola.

The Coq source code for this part is in the file `nola/examples/minilogic.v`.

Construct the Syntax *nProp* To begin with, let us construct the data type `nProp` for the syntactic representation of separation logic propositions. We can express *nProp* of (3.16) in Coq as the following inductive data type:

```
Inductive nProp : Type :=
| all {A : Type} (Φ : A → nProp) | ex {A : Type} (Φ : A → nProp)
| and (P Q : nProp) | or (P Q : nProp) | imp (P Q : nProp)
| pure (φ : Prop)
| sep (P Q : nProp) | wand (P Q : nProp) | pers (P : nProp)
| bupd (P : nProp) | later (P : nProp)
| pointsto (q : frac) (l : loc) (v : val)
| inv (N : namespace) (P : nProp)
| ilist (N : namespace) (Φ : loc → nProp) (l : loc).
```

Code 8.1: (3.16)'s *nProp* in Coq

Note again that we use higher-order abstract syntax for (first-order) quantifiers `all` and `ex`. Technically, in Coq, the type of types `Type` is implicitly parameterized over the *universe level* `u`. Coq's type system assigns a universal level to each `Type` for the universe consistency. By that, the domain type `A : Type` of the quantifiers lives in a universe level strictly smaller than that of `nProp`, which makes the type `nProp` well-formed.

Construct the Semantics $[\![\ ]\!]$ Now we construct the semantic interpretation `intp : nProp → iProp Σ` of the data type `nProp`.

First, we need to add some constraints on the global camera `Σ` for Iris propositions `iProp Σ`. We require `ninvGS nProp Σ` to use Nola's invariant. Also, we require `heapGS_gen HasNoLc Σ` for the heap memory state of Iris's HeapLang, especially to use the points-to token `l ↦{q} v : iProp Σ`.[3] A simple way is to directly add these constraints to the context:

```
Context `{!ninvGS nProp Σ, !heapGS_gen HasNoLc Σ}.
```

Now we can construct the semantic interpretation `intp : nProp → iProp Σ` by structural induction over the inductive data type `nProp`:

[3] The flag `HasNoLc` says that we do *not* use the later credit machinery (Spies et al., 2022) for Iris's original invariant mechanism and fancy update modality. The default of current Iris is to use the later credit with the flag `HasLc` (`heapGS Σ` is an alias of `heapGS_gen HasLc Σ`, `invGS Σ` is an alias of `invGS_gen HasLc Σ`, etc.). However, the later credit is unsuitable for verifying *liveness properties* (see also § 9.1), and so we use the flag `HasNoLc` for this case study targeting total correctness.

```
Fixpoint intp (P : nProp) : iProp Σ := match P with
| all Φ => ∀ x, intp (Φ x) | ex Φ => ∃ x, intp (Φ x)
| and P Q => intp P ∧ intp Q | or P Q => intp P ∨ intp Q
| imp P Q => intp P → intp Q | pure φ => ⌜φ⌝
| sep P Q => intp P ∗ intp Q | wand P Q => intp P -∗ intp Q
| pers P => □ intp P | bupd P => |==> intp P | later P => ▷ intp P
| pointsto q l v => l ↦{#q} v
| inv N P => inv_tok N P
| ilist N Φ l => inv_tok N (Φ l) ∗ inv_tok N
    (ex (λ l' : loc, sep (pointsto 1 (l +ₗ 1) (#l')) (ilist N Φ
   l')))
end.
```

Code 8.2: Interpretation intp for Code 8.1's nProp

The most part is straightforward. Here we comment on some points.

For the quantifiers all and ex in higher-order abstract syntax, the recursion is well-formed, passing Coq's termination checker, because Φ x is structurally smaller than all Φ and ex Φ for any x.

The invariant connective inv N P is just interpreted as Nola's invariant inv_tok N P (just as (3.17)). The infinite list connective ilist N Φ l is interpreted as the conjunction of the invariants for the head and the tail, without any recursive call to intp (just as (3.18)). Note again that the body P of Nola's invariant inv_tok N P is a syntactic proposition nProp, not a semantic one iProp Σ.

Verification Goal: Termination of Linked List Mutation The primary target function of our verification is the following function iter corresponding to (3.8), coded in Iris's HeapLang with heavy syntax sugar:[4]

```
Definition iter : val := rec: "self" "f" "c" "l" :=
  if: !"c" = #0 then #() else
    "f" "l";; "c" <- !"c" - #1;; "self" "f" "c" (!("l" +ₗ #1)).
```

Code 8.3: (3.8)'s iter in Coq

The function is shallow-embedded into Coq, where the variables are bound by Coq's strings like "l". The hash # is used just for annotating literals. Roughly speaking, the function iter applies the function f to the first !"c" elements of the list starting at l, decrementing the counter c.

Now our primary verification goal, corresponding to (3.19), can be described as follows:

```
Lemma twp_iter {N Φ c l} {f : val} {n : nat} :
  (∀ l0 : loc,
    [[{ inv_tok N (Φ l0) }]][inv_wsat intp]
      f #l0 @ ↑N
    [[{ RET #(); True }]]) -∗
  [[{ c ↦ #n ∗ intp (ilist N Φ l) }]][inv_wsat intp]
    iter f #c #l @ ↑N
  [[{ RET #(); c ↦ #0 }]].
```

Code 8.4: Verification goal of (3.19) in Coq

[4] Exploiting Coq's rich power, Iris's HeapLang introduces custom notations such as the recursive function rec: ... := ... and the if expression if: ... then ... else, using the colon : for disambiguation from Coq's native syntax.

Here we use the notation `[[{ P }]][ W ] e @ E [[{ RET #(); Q }]]`[5] for the *extended* total Hoare triple with a custom world satisfaction `W : iProp Σ` (corresponding to $[P]\ e\ [\Psi]^{W}_{E}$), a feature newly introduced by our Coq mechanization.

How Verification Goes in Coq Now we explain how we can verify the goal Code 8.4 using our Coq mechanization of the Nola framework. At a high level, the verification goes straightforwardly by induction over the natural number `n : nat`, as explained in § 3.3.3. In Coq, this is completed with the help of Iris Proof Mode (Krebbers et al., 2017b), which provides various proof tactics for interactive theorem proving in Iris. Here, we give a rough idea of that.

First, we start the proof by **`Proof`** and introduce the hypothesis `Hf` on the function `f` by `iIntros`:

```
Proof.
  iIntros "#Hf".
```

This magically turns the goal into the following (reformatted for readability):

```
"Hf" : ∀ l0 : loc,
         [[{ inv_tok N (Φ l0) }]][inv_wsat intp]
           f #l0 @ ↑N
         [[{ RET #(); True }]]
---------------------------------------□
[[{ c ↦ #n ∗ intp (ilist N Φ l) }]][inv_wsat intp]
  iter f #c #l @ ↑N
[[{ RET #(); c ↦ #0 }]]
```

Iris Proof Mode prints the hypotheses and goal of Iris separation logic in a readable way. Above `---□` are persistent hypotheses, and at the bottom is the goal.

Then we introduce the hypotheses of the goal Hoare triple. We can use the following tactic (we omit the details).

```
  iIntros (Ψ) "!> [c↦ #[ihd itl]] HΨ".
```

Then the goal turns into the following:

```
"Hf" : ∀ l0 : loc,
         [[{ inv_tok N (Φ l0) }]][inv_wsat intp]
           f #l0 @ ↑N
         [[{ RET #(); True }]]
"ihd" : inv_tok N (Φ l)
"itl" : inv_tok N (ex (λ l' : loc,
          sep (pointsto 1 (l +ₗ 1) #l') (ilist N Φ l')))
---------------------------------------□
"c↦" : c ↦ #n
"HΨ" : c ↦ #0 -∗ Ψ #()
---------------------------------------∗
WP[inv_wsat intp] iter f #c #l @ ↑N [{ v, Ψ v }]
```

Here, `Ψ` is the postcondition parameter and `HΨ` is the spatial hypothesis on `Ψ`. The points-to token on the counter `c ↦ #n` is named `c↦`. The invariants on the head and the tail are named `ihd` and `itl`. The goal is an extended total weakest precondition `WP[W] e @ E [{v, Ψ v}]` (corresponding to twp $e\ [\Psi]^{W}_{E}$), a feature newly introduced by our Coq mechanization.

Now we perform the induction over `n : nat` by the following tactic:

5 The part `RET #()` requires that the expression `e : expr` should return the unit value `#()`.

```
iInduction n as [|m] "IH" forall (l) "ihd itl".
```

We get two subgoals, for the base case `n = 0` and the induction step `n = S m`. We generalize the goal over the location value `l` and the hypotheses `ihd` and `itl`.

The base case is trivial and can be solved by the following tactics:

```
{ wp_rec. wp_pures. wp_load. wp_pures. by iApply "HΨ". }
```

For the induction step, `c↦` turns into `c ↦ #(S m)` and we have the following inductive hypothesis `IH`:

```
"IH" : ∀ l0 : loc,
          c ↦ #m -∗ (c ↦ #0 -∗ Ψ #()) -∗
          □ inv_tok N (Φ l0) -∗
          □ inv_tok N (ex (λ l' : loc,
              sep (pointsto 1 (l0 +ₗ 1) #l') (ilist N Φ l'))) -∗
          WP[inv_wsat intp] iter f #c #l0 @ ↑N [{ v, Ψ v }]
```

We first perform the case branching `if: !"c" = #0` by the following tactics:

```
wp_rec. wp_pures. wp_load. wp_pures.
```

The goal turns into the following:

```
WP[inv_wsat intp]
  f #l;; #c <- !#c - #1;; iter f #c (!(#l +ₗ #1)) @ ↑N
[{ v, Ψ v }]
```

We can discharge the function call `f #l;;` by the following tactics, using the hypothesis `Hf` on the function `f`:

```
wp_apply "Hf"; [done|]. iIntros "_". wp_pures.
```

Then we can discharge the counter decrement `#c <- !#c - #1;;` by the following tactics:

```
wp_load. wp_op. have -> : (S m - 1)%Z = m by lia. wp_store.
```

This also updates `c↦` to `c ↦ #m`. In general, spatial hypotheses can be updated according to the state mutation in Iris Proof Mode.

Now we have reached the core challenge: the dereference of the list into the tail. We first perform the following tactics.

```
wp_op. wp_bind (! _)%E.
```

This turns the goal into the following nested weakest precondition predicate:

```
WP[inv_wsat intp] !#(l +ₗ 1) @ ↑N
[{ v, WP[inv_wsat intp] iter f #c v @ ↑N [{ v, Ψ v }] }]
```

Now we *open* the invariant for the tail:

```
iMod (inv_tok_acc with "itl") as
  "/=[(%l' & ↦l' & #ithd & #ittl) cl]"; [done|].
```

This turns the contexts and goal into the following:

```
...
"itlhd" : inv_tok N (Φ l')
"itltl" : inv_tok N (ex (λ l'' : loc,
             sep (pointsto 1 (l' +ₗ 1) #l'') (ilist N Φ l'')))
-------------------------------------□
...
```

```
"↦l''" : (l +ₗ 1) ↦ #l'
"cl" : (∃ l' : loc, (l +ₗ 1) ↦ #l' ∗ inv_tok N (Φ l') ∗
         inv_tok N (ex (λ l'' : loc,
           sep (pointsto 1 (l' +ₗ 1) #l'') (ilist N Φ l''))))
       =[inv_wsat intp]{↑N\↑N,↑N}=∗ True
--------------------------------------∗
WP[inv_wsat intp] !#(l +ₗ 1) @ ↑N\↑N
[{ v, |=[inv_wsat intp]{↑N\↑N,↑N}=>
  WP[inv_wsat intp] iter f #c v @ ↑N [{ v, Ψ v }] }]
```

We have deposited the namespace N to open the invariant. Now we can perform the dereference using ↦l'.

To recover the namespace N, we *close* the invariant using cl and the hypotheses ↦l', itlhd and itltl. These can be done by the following tactics:

```
  wp_load. iModIntro. iMod ("cl" with "[↦l']") as "_".
  { iExists _. iFrame "↦l''". by iSplit. } iModIntro.
```

Now the goal is simply as follows:

```
WP[inv_wsat intp] iter f #c #l' @ ↑N [{ v, Ψ v }]
```

This can be proved just by applying the inductive hypothesis IH.

```
  by iApply ("IH" with "c↦ HΨ").
Qed.
```

Finally, we have reached the end of the proof, **Qed**!

Chapter 9

Related Work

If I have seen further it is by standing on þe sholders of Giants.

Isaac Newton, Letter to Robert Hooke

9.1 Invariants with Later-Free Rules

There are some existing logics (Swamy et al., 2020; Svendsen and Birkedal, 2014; Svendsen et al., 2013; Spies et al., 2022) that provide later-free proof rules for shared invariants. None of them has been applied to liveness verification, unlike Nola. But one may wonder if they could possibly be extended to liveness verification. As discussed below, all of them either *use step-indexed program logic* (which cannot support liveness verification) or *restrict nesting of invariants*, whereas our Nola framework supports the invariant mechanism that works in *non-step-indexed program logic* and supports *genuine nesting*.

SteelCore (Swamy et al., 2020) (or its interface Steel (Fromherz et al., 2021)) is an F*-based verification framework employing (nested) invariants with later-free proof rules. However, the program logic of SteelCore is actually *step-indexed*. Indeed, as discussed by Swamy et al. (2020, § 4.4), they are using "monotonic state" modeled by Ahman et al. (2017), which "has a 'later' modality in disguise". Due to this, it is unclear whether their approach extends to liveness verification. Moreover, there is no known formal proof of the soundness of SteelCore's logic.

iCAP (Svendsen and Birkedal, 2014) and HOCAP (Svendsen et al., 2013) are other logics with invariants with later-free proof rules. The soundness of their later-free rules does not seem to rely on step-indexing, so these logics may possibly be applicable to liveness verification.[1] Still, they *do not support genuine nesting of invariants*. iCAP does not allow any kind of nesting of the invariant connective. HOCAP allows nesting in a certain form, but it prohibits nesting invariants of overlapping *region types* t. This condition is essential to the soundness of their logic, because such nesting introduces "self-referential region assertions", which generally "do not admit modular stability proofs" (Svendsen et al., 2013, § 2.2). Unlike theirs, our Nola framework allows genuine nesting of invariants (e.g., infinite singly linked lists `ilist` in § 3.3.3).

The later credit $£\, n$ (Spies et al., 2022), recently introduced to Iris, is known to allow for the "prepaid invariant" $\boxed{P}^{\mathcal{N}}_{\text{pre}}$ that supports a seemingly later-free access rule InvPreOpen for a partial Hoare triple (Spies et al., 2022, § 6.1). Still, the program logic is *step-indexed* even under the later credit machinery. The later credit depends on a

[1] The program logic of HOCAP is actually *step-indexed*. But according to their paper (Svendsen et al., 2013, § 4), this indexing is for supporting nested Hoare triples (i.e., Hoare triples $\{P\}\ e\ \{\Psi\}$ such that the precondition P and postcondition Ψ can contain Hoare triple connectives). It may be possible to construct a variant of HOCAP that does not support nested Hoare triples but employs non-step-indexed program logic that can reason about liveness properties.

new, later-eliminating fancy update $\Rrightarrow_{\mathsf{le}}$ that contains laters $\triangleright$ inside it, just *hiding step-indexing*. Indeed, Spies et al. (2022, § 5.2) admit that this new fancy update $\Rrightarrow_{\mathsf{le}}$ does not satisfy "interaction rules with Iris's 'plainly modality' $\blacksquare P$". These 'plainly'-related rules (more specifically, $\Rrightarrow$-PURE-KEEP in § 3.1) are vital to proving the termination adequacy theorem for Iris's total weakest precondition $\mathrm{twp}\ e\ [\Psi]$ (like Theorem 3.2 in § 3.1). Therefore, the total weakest precondition works only with the original fancy update $\Rrightarrow$ and not with $\Rrightarrow_{\mathsf{le}}$.

Also, it is unclear whether the above approaches could lead to a later-free version of the borrows of RustBelt's lifetime logic (Jung et al., 2018a), which is far more complicated than invariants.

9.2 Termination and Liveness Verification

Our Nola framework provides later-free mechanisms, such as invariants and borrows, that work in *non-step-indexed* program logic, which can verify *liveness properties*, including termination and total correctness.

Transfinite Indexing There actually exists one approach to verifying liveness properties in non-step-indexed program logic. The approach is *transfinite indexing* (Spies et al., 2021b), where the indices $\mathbb{I}$ are *ordinal numbers*, in constract to the standard *finite indexing*, where the indices $\mathbb{I}$ are natural numbers $0, 1, 2, \ldots \in \mathbb{N}$. Recently, Transfinite Iris (Spies et al., 2021a), a variant of Iris that uses transfinite indexing, has been developed.

However, Transfinite Iris suffers from several limitations. First, in the step-indexed total Hoare triple of Transfinite Iris, as a trade-off of the power to eliminate the later modality $\triangleright$ for each program step, one should explicitly *bound* the number of program steps with an ordinal number α using a time credit $\$\alpha$ (Atkey, 2010; Mével et al., 2019). When we eliminate the later modality from the precondition, we should decrease the ordinal of the time credit by a proof rule like this:

$$\frac{\left[\$\beta * P\right] e \left[\Psi\right]^* \qquad \beta < \alpha}{\left[\$\alpha * \triangleright P\right] e \left[\Psi\right]}$$

We cannot remove this time credit and have the proof rule STEP-TWP (§ 1.4), because Löb induction LÖB holds also in transfinite indexing and thus the rule STEP-TWP leads to a contradiction as discussed in § 1.4. Second, liveness properties supported by Transfinite Iris of Spies et al. (2021a) are limited to fairly simple ones: termination and termination-preserving refinements for sequential, *non-concurrent* programs. In particular, it is an open problem how to construct in Transfinite Iris program logics for more complex liveness properties, such as Simuliris (Gäher et al., 2022) and Fairness Logic (Lee et al., 2023). Third, the mechanization of Transfinite Iris heavily depends on Coq's *universe polymorphism*, an experimental feature, to reason about ordinals. Finally, in Transfinite Iris, *the later modality* $\triangleright$ *loses the commutativity laws* with the separating conjunction $*$ and the existential quantifier $\exists$, which hold under *finite indexing* as we saw in § 3.1 ($\triangleright$-$*$, $\triangleright$-$\exists$):

$$\triangleright(P * Q) \;=\; (\triangleright P) * (\triangleright Q) \qquad\qquad \frac{A \neq \varnothing}{\triangleright(\exists a \in A.\, P_a) \;=\; \exists a \in A.\, \triangleright P_a}$$

Many Iris developments, including RustBelt's *lifetime logic* (Jung et al., 2018a), critically rely on these laws and thus stop working in Transfinite Iris. In particular, it is an open problem how to construct the borrow mechanism in Transfinite Iris.

Nevertheless, transfinite indexing satisfies the following *existential property*, unlike finite indexing: for any set A whose cardinality is not 'too large' and any predicate $\Phi\colon A \rightarrow iProp$, if $\vDash \exists a \in A.\, \Phi\, a$ holds, then there exists some $a_0 \in A$ satisfying $\vDash \Phi\, a_0$. The existential property is a primary selling point of Transfinite Iris (Spies et al., 2021a), which can be used to prove the adequacy theorem of some program logics for liveness verification built in Transfinite Iris. The existential property can also help other tasks like handling angelic non-determinism (Guéneau et al., 2023). Our Nola framework can also be ported to Transfinite Iris, if one ever wants to use the existential property.

Bounded Termination-Preserving Refinements Tassarotti et al. (2017) built on Iris step-indexed program logic that is able to verify termination-preserving refinements, which may seem like a liveness property. However, what the logic can prove is actually a *safety* property due to its strong restriction: the source program can use only *bounded* non-determinism and stuttering. The program cannot, for example, take a non-deterministic natural number. Also, their approach generally does not work for non-relational termination verification.

Extending the idea of Tassarotti et al. (2017), Timany et al. (2024) built Trillium, step-indexed program logic that can show intensional termination-preserving refinements between traces of the program and the specification model. Like Tassarotti et al. (2017)'s logic, what Trillium directly shows is a *safety* property, imposing a strong restriction between the program and the model. Also, to prove a liveness property of the program, besides proving the termination-preserving refinement in Trillium, one needs to prove a liveness property of the model separately, which is not always trivial.

Rich Liveness Properties Various separation logics that can verify rich liveness properties have been studied.

LiLi (Liang and Feng, 2016, 2018) is an early concurrent program logic for verifying liveness properties such as deadlock freedom. TaDA Live (D'Osualdo et al., 2021), based on da Rocha Pinto et al. (2016)'s logic, provides a rich interface for liveness verification of fine-grained concurrent programs. Simuliris (Gäher et al., 2022) is an Iris-based logic that can prove fair termination preservation of various concurrent program optimizations, including advanced Rust-inspired examples from Stacked Borrows (Jung et al., 2020a). Fairness Logic (Lee et al., 2023) is a logic based on Simuliris for reasoning about various kinds of fairness properties.

However, propositional sharing in these logics has been little studied. In particular, although Simuliris and Fairness Logic are Iris-based, they cannot use Iris's later-requiring invariants because their program logics cannot be step-indexed.

Using our Nola framework, one can use propositional sharing with these rich program logics for liveness verification. Exploring such direction is left for future work.

9.3 Tackling Laters

As discussed in § 1.4, the later modality $\triangleright$ is ill-behaved in that it is not idempotent and does not commute with the fancy update modality $\Rrightarrow_{\mathcal{E}}$. This is problematic even in *safety* verification.

For example, in order to traverse a nested data structure modeled with k-fold nesting of later-requiring propositional sharing, one should eliminate k laters $\triangleright^k$, or even worse, k laters interleaved with the fancy update $(\triangleright \Rrightarrow)^k$. Rules like STEP-PHOARE, stripping one later for one program step, are not enough for this purpose.

Various workarounds have been proposed to tackle such problems with the later modality.

RustBelt's Delayed Sharing One early example is from RustBelt (Jung et al., 2018a), a semantic foundation for Rust's type system. Conversion from a mutable reference `&α mut T` to a shared reference `&α T` naively requires traversing over the whole structure of a Rust object typed `T`. And when the object has the depth d, d laters $\triangleright^d$ should be eliminated. But that is not doable because an object can have an unbounded depth (e.g., a singly linked list).

They tackled this by a workaround called *delayed sharing* (Jung, 2020, Chapter 12): they give up traversal in one go and instead perform conversion to a shared reference on demand, i.e., only when each subobject gets accessed. Although it works, this substantially complicates the model of shared references.

Flexible Step-Indexing by RustHornBelt Another example is from RustHornBelt (Matsushita et al., 2022), a successor of RustBelt for RustHorn-style functional Rust program verification, introduced in § 7.1.2 To reason about mutable references `&α mut T`, they needed to traverse the whole object `T` (more specifically, take tokens out of the entire object), where delaying like delayed sharing does not work.

They tackled this by tweaking Iris's weakest precondition to strip off increasingly many laters for each program step, e.g., k laters $\triangleright^k$ at the k-th step. They lower-bound the number of past program steps by a time receipt $⧗\, n$ (Mével et al., 2019) and keep track of object depths. This technique, nicknamed flexible step-indexing, generally helps reason about nested data structures, being applied to other recent projects (Hinrichsen et al., 2022; Gondelman et al., 2023). But using it requires cumbersome bookkeeping of program step counts and object depths.

Later Credits One recent progress is the later credit $£\, n$ (Spies et al., 2022), which owns the right to eliminate n laters under the later elimination update $\Rrightarrow_{\mathsf{le}}$:

$$£\, n \ * \ \triangleright^n P \ \vDash \Rrightarrow_{\mathsf{le}} \ P.$$

The later elimination update $\Rrightarrow_{\mathsf{le}}$ is modeled as a fancy update $\Rrightarrow$ with hidden laters inside corresponding to the consumed later credits.

Still, even with later credits, it seems hard to traverse nested data structures *unboundedly many times* (e.g., RustHornBelt's case), which require an unbounded number of later credits.

With Nola Using our Nola framework, one can use later-free mechanisms for propositional sharing, eliminating the need for these techniques to tackle the later modality. Also, we can still apply these techniques even if we use the later modality to support advanced features as discussed in § 3.4.2.

Chapter 10

Conclusion

ゆく河の流れは絶えずしてしかももとの水にあらず
The river always flows and its water never stays the same

Kamo no Chōmei, *Hōjōki*

This dissertation proposed a novel general framework, Nola, which provides separation logic with later-free mechanisms for propositional sharing, such as invariants and borrows, which can be used for verifying programs with shared mutable state in non-step-indexed program logic.

We expect that our framework greatly opens the possibilities for verifying programs with shared mutable state, as discussed in § 1.5.2. The invariant and borrow can possibly be safely integrated into separation-logic-based automated verification platforms such as Viper (Müller et al., 2016), being free from the later modality and naturally supporting liveness verification. RustHornBelt (Matsushita et al., 2022), a semantic foundation for functional Rust verifiers such as RustHorn (Matsushita et al., 2020) and Creusot (Denis et al., 2022), can possibly be rebuilt on Nola to support liveness verification. Also, Simuliris (Gäher et al., 2022) can possibly be used to verify optimizations under the guarantee of ownership types modeled with Nola's invariant and borrow. We leave further investigation of these directions to future work.

From another perspective, we expect that our approach can be used even outside the context of separation logic. Our key idea, the isolation of the proposition syntax from its interpretation, may well find applications in other contexts. Also, our technique for semantic alteration (Chapter 4), especially the construction of the derivability predicate, can work in any logic, not only separation logic. We leave the scientific exploration of these possibilities as an interesting topic for future work.

Bibliography

Martín Abadi and Leslie Lamport. 1988. The Existence of Refinement Mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988.* IEEE Computer Society, 165–175. https://doi.org/10.1109/LICS.1988.5115

Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. 2017. Recalling a Witness: Foundations and Applications of Monotonic State. *CoRR* abs/1707.02466 (2017). arXiv:1707.02466 http://arxiv.org/abs/1707.02466

Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. 2002. A Stratified Semantics of General References Embeddable in Higher-Order Logic. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings.* IEEE Computer Society, 75. https://doi.org/10.1109/LICS.2002.1029818

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

Pierre America and Jan J. M. M. Rutten. 1989. Solving Reflexive Domain Equations in a Category of Complete Metric Spaces. *J. Comput. Syst. Sci.* 39, 3 (1989), 343–375. https://doi.org/10.1016/0022-0000(89)90027-5

Andrew W. Appel and David A. McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Trans. Program. Lang. Syst.* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023. Reference Capabilities for Flexible Memory Management. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1363–1393. https://doi.org/10.1145/3622846

Robert Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6012)*, Andrew D. Gordon (Ed.). Springer, 85–103. https://doi.org/10.1007/978-3-642-11957-6_6

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.* 2, POPL (2018), 5:1–5:29. https://doi.org/10.1145/3158093

Lars Birkedal and Aleš Bizjak. 2023. *Lecture Notes on Iris: Higher-Order Concurrent Separation Logic*. https://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First Steps in Synthetic Guarded Domain Theory: Step-indexing in the Topos of Trees. *Log. Methods Comput. Sci.* 8, 4 (2012). https://doi.org/10.2168/LMCS-8(4:1)2012

Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. 2010. The Category-Theoretic Solution of Recursive Metric-Space Equations. *Theor. Comput. Sci.* 411, 47 (2010), 4102–4122. https://doi.org/10.1016/j.tcs.2010.07.010

Ales Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: Managing Obligations in Higher-Order Concurrent Separation Logic. *Proc. ACM Program. Lang.* 3, POPL (2019), 65:1–65:30. https://doi.org/10.1145/3290378

Nikolaj S. Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday (Lecture Notes in Computer Science, Vol. 9300)*, Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte (Eds.). Springer, 24–51. https://doi.org/10.1007/978-3-319-23534-9_2

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission Accounting in Separation Logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 259–270. https://doi.org/10.1145/1040305.1040327

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2694)*, Radhia Cousot (Ed.). Springer, 55–72. https://doi.org/10.1007/3-540-44898-5_4

Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent Separation Logic. *ACM SIGLOG News* 3, 3 (2016), 47–65. https://doi.org/10.1145/2984450.2984457

Stephen D. Brookes. 2004. A Semantics for Concurrent Separation Logic. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3170)*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer, 16–34. https://doi.org/10.1007/978-3-540-28644-8_2

Alexandre Buisse, Lars Birkedal, and Kristian Støvring. 2011. Step-Indexed Kripke Model of Separation Logic for Storable Locks. In *Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics, MFPS 2011, Pittsburgh, PA, USA, May 25-28, 2011 (Electronic Notes in Theoretical Computer Science, Vol. 276)*, Michael W. Mislove and Joël Ouaknine (Eds.). Elsevier, 121–143. https://doi.org/10.1016/j.entcs.2011.09.018

Adrien Champion, Naoki Kobayashi, and Ryosuke Sato. 2018. HoIce: An ICE-Based Non-linear Horn Clause Solver. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11275)*, Sukyoung Ryu (Ed.). Springer, 146–156. https://doi.org/10.1007/978-3-030-02768-1_8

Chromium Projects. 2023. *Memory safety - Chromium Security*. https://www.chromium.org/Home/chromium-security/memory-safety/

Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). 2013. *Aliasing in Object-Oriented Programming: Types, Analysis and Verification*. Lecture Notes in Computer Science, Vol. 7850. Springer. https://doi.org/10.1007/978-3-642-36946-9

David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1998, Vancouver, British Columbia, Canada, October 18-22, 1998*, Bjørn N. Freeman-Benson and Craig Chambers (Eds.). ACM, 48–64. https://doi.org/10.1145/286936.286947

Coq Team. 2023. *The Coq Proof Assistant*. https://coq.inria.fr/

Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. 2016. Modular Termination Verification for Non-blocking Concurrency. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9632)*, Peter Thiemann (Ed.). Springer, 176–201. https://doi.org/10.1007/978-3-662-49498-1_8

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt Meets Relaxed Memory. *Proc. ACM Program. Lang.* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13478)*, Adrián Riesco and Min Zhang (Eds.). Springer, 90–105. https://doi.org/10.1007/978-3-031-17244-1_6

Edsger W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9 (1965), 569. https://doi.org/10.1145/365559.365617

Mike Dodds, Suresh Jagannathan, Matthew J. Parkinson, Kasper Svendsen, and Lars Birkedal. 2016. Verifying Custom Synchronization Constructs Using Higher-Order Separation Logic. *ACM Trans. Program. Lang. Syst.* 38, 2 (2016), 4:1–4:72. https://doi.org/10.1145/2818638

Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. 2021. TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs. *ACM Trans. Program. Lang. Syst.* 43, 4 (2021), 16:1–16:134. https://doi.org/10.1145/3477082

European Association for Theoretical Computer Science. 2016. *2016 Gödel Prize*. https://eatcs.org/index.php/component/content/article/1-news/2280-2016-godel-prize

Jonás Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, and Ilya Sergey. 2023. Leveraging Rust Types for Program Synthesis. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1414–1437. https://doi.org/10.1145/3591278

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32. http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf

Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. 2006. Linear Regions Are All You Need. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3924)*, Peter Sestoft (Ed.). Springer, 7–21. https://doi.org/10.1007/11693024_2

Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. 2021. Steel: Proof-Oriented Programming in a Dependently Typed Concurrent Separation Logic. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473590

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 442–451. https://doi.org/10.1145/3209108.3209174

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: A Separation Logic Framework for Verifying Concurrent Program Optimizations. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498689

David Gay and Alexander Aiken. 1998. Memory Management with Explicit Regions. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, Jack W. Davidson, Keith D. Cooper, and A. Michael Berman (Eds.). ACM, 313–323. https://doi.org/10.1145/277650.277748

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. 2023. Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols. *Proc. ACM Program. Lang.* 7, ICFP (2023), 847–877. https://doi.org/10.1145/3607859

Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. 2021. Mechanized Logical Relations for Termination-Insensitive Noninterference. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434291

Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and*

Implementation (PLDI), Berlin, Germany, June 17-19, 2002, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 282–293. https://doi.org/10.1145/512529.512563

Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 247 (oct 2023), 29 pages. https://doi.org/10.1145/3622823

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: Session-Type Based Reasoning in Separation Logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 6:1–6:30. https://doi.org/10.1145/3371074

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2022. Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic. *Log. Methods Comput. Sci.* 18, 2 (2022). https://doi.org/10.46298/lmcs-18(2:16)2022

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4960)*, Sophia Drossopoulou (Ed.). Springer, 353–367. https://doi.org/10.1007/978-3-540-78739-6_27

John Hogg, Doug Lea, Alan Cameron Wills, Dennis de Champeaux, and Richard C. Holt. 1992. The Geneva convention on the treatment of object aliasing. *OOPS Messenger* 3, 2 (1992), 11–16. https://doi.org/10.1145/130943.130947

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The Power of Parameterization in Coinductive Proof. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 193–206. https://doi.org/10.1145/2429069.2429093

Iris Team. 2023a. *The Iris 4.1 Reference.* https://plv.mpi-sws.org/iris/appendix-4.1.pdf

Iris Team. 2023b. *Iris Coq Development.* https://gitlab.mpi-sws.org/iris/iris

Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an Assertion Language for Mutable Data Structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, Chris Hankin and Dave Schmidt (Eds.). ACM, 14–26. https://doi.org/10.1145/360204.375719

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

Koen Jacobs, Dominique Devriese, and Amin Timany. 2022. Purity of An ST Monad: Full Abstraction by Semantically Typed Back-Translation. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–27. https://doi.org/10.1145/3527326

Jacques-Henri Jourdan. 2018. *Insufficient synchronization in* `Arc::get_mut` *— Rust Issue #51780*. https://github.com/rust-lang/rust/issues/51780

Ralf Jung. 2017. `MutexGuard<Cell<i32>>` *must not be* `Sync` *— Rust Issue #41622*. https://github.com/rust-lang/rust/issues/41622

Ralf Jung. 2020. *Understanding and Evolving the Rust Programming Language*. Ph. D. Dissertation. Saarland University, Saarbrücken, Germany. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020a. Stacked borrows: an aliasing model for Rust. *Proc. ACM Program. Lang.* 4, POPL (2020), 41:1–41:32. https://doi.org/10.1145/3371109

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 256–269. https://doi.org/10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020b. The Future is Ours: Prophecy Variables in Separation Logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes* *in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. https://doi.org/10.1145/3009837.3009855

Dongjae Lee, Minki Cho, Jinwoo Kim, Soonwon Moon, Youngju Song, and Chung-Kil Hur. 2023. Fair Operational Semantics. *Proc. ACM Program. Lang.* 7, PLDI (2023), 811–834. https://doi.org/10.1145/3591253

Hongjin Liang and Xinyu Feng. 2016. A Program Logic for Concurrent Objects under Fair Scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 385–399. https://doi.org/10.1145/2837614.2837635

Hongjin Liang and Xinyu Feng. 2018. Progress of Concurrent Objects with Partial Methods. *Proc. ACM Program. Lang.* 2, POPL (2018), 20:1–20:31. https://doi.org/10.1145/3158108

Niko Matsakis. 2017. *2094-nll — The Rust RFC Book*. https://rust-lang.github.io/rfcs/2094-nll.html

Niko Matsakis. 2022. *Non-lexical lifetimes (NLL) fully stable*. https://blog.rust-lang.org/2022/08/05/nll-by-default.html

Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, Michael B. Feldman and S. Tucker Taft (Eds.). ACM, 103–104. https://doi.org/10.1145/2663171.2663188

Yusuke Matsushita. 2019. *CHC-based Program Verification Exploiting Ownership Types.* Senior Thesis. University of Tokyo.

Yusuke Matsushita. 2021. *Extensible Functional-Correctness Verification of Rust Programs by the Technique of Prophecy.* Master's thesis. University of Tokyo.

Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs with Unsafe Code. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 841–856. https://doi.org/10.1145/3519939.3523704

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2020. RustHorn: CHC-Based Verification for Rust Programs. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 484–514. https://doi.org/10.1007/978-3-030-44914-8_18

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *ACM Trans. Program. Lang. Syst.* 43, 4 (2021), 15:1–15:54. https://doi.org/10.1145/3462205

J. McCarthy and P.J. Hayes. 1981. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Readings in Artificial Intelligence*, Bonnie Lynn Webber and Nils J. Nilsson (Eds.). Morgan Kaufmann, 431–450. https://doi.org/10.1016/B978-0-934613-03-3.50033-7

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

Microsoft Security Response Center. 2019. *We need a safer systems programming language.* https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/

Lawrence S. Moss. 2001. Parametric corecursion. *Theor. Comput. Sci.* 260, 1-2 (2001), 139–163. https://doi.org/10.1016/S0304-3975(00)00126-2

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Hiroshi Nakano. 2000. A Modality for Recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. IEEE Computer Society, 255–266. https://doi.org/10.1109/LICS.2000.855774

Takashi Nakayama, Yusuke Matsushita, Ken Sakayori, Ryosuke Sato, and Naoki Kobayashi. 2024. Borrowable Fractional Ownership Types for Verification. In *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 14500)*, Rayna Dimitrova, Ori Lahav, and Sebastian Wolff (Eds.). Springer, 224–246. https://doi.org/10.1007/978-3-031-50521-8_11

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3170)*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer, 49–67. https://doi.org/10.1007/978-3-540-28644-8_4

Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95. https://doi.org/10.1145/3211968

Peter W. O'Hearn and David J. Pym. 1999. The Logic of Bunched Implications. *Bull. Symb. Log.* 5, 2 (1999), 215–244. https://doi.org/10.2307/421090

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2142)*, Laurent Fribourg (Ed.). Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

Project Zero. 2019. *0day "In the Wild"*. https://googleprojectzero.blogspot.com/p/0day.html

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Rust Team. 2023. *Rust Programming Language*. http://www.rust-lang.org/

Sarek Høverstad Skotåm. 2022. *CreuSAT: Using Rust and Creusot to create the world's fastest deductively verified SAT solver*. Master's thesis. University of Oslo. https://www.duo.uio.no/handle/10852/96757

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021a. Transfinite Iris: Resolving an Existential Dilemma of Step-Indexed Separation Logic. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 80–95. https://doi.org/10.1145/3453483.3454031

Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later Credits: Resourceful Reasoning for the Later Modality. *Proc. ACM Program. Lang.* 6, ICFP (2022), 283–311. https://doi.org/10.1145/3547631

Simon Spies, Neel Krishnaswami, and Derek Dreyer. 2021b. Transfinite Step-indexing for Termination. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434294

std++ Team. 2023. *Coq-std++*. https://gitlab.mpi-sws.org/iris/stdpp

Jeffrey Vander Stoep. 2022. *Memory Safe Languages in Android 13 - Google Online Security Blog*. https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Modular Reasoning about Separation of Concurrent Data Structures. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 169–188. https://doi.org/10.1007/978-3-642-37036-6_11

Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs. *Proc. ACM Program. Lang.* 4, ICFP (2020), 121:1–121:30. https://doi.org/10.1145/3409003

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2024. Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement. *Proc. ACM Program. Lang.* 8, POPL (2024), 241–272. https://doi.org/10.1145/3632851

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2023. A Logical Approach to Type Soundness. (2023).

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of runST. *Proc. ACM Program. Lang.* 2, POPL (2018), 64:1–64:28. https://doi.org/10.1145/3158152

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based Memory Management. *Inf. Comput.* 132, 2 (1997), 109–176. https://doi.org/10.1006/INCO.1996.2613

Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification.* Ph. D. Dissertation. University of Cambridge, UK. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.612221

Philip Wadler. 1990. Linear Types can Change the World!. In *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, Manfred Broy and Cliff B. Jones (Eds.). North-Holland, 561.

Glynn Winskel. 1989. A Note on Model Checking the Modal nu-Calculus. In *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings (Lecture Notes in Computer Science, Vol. 372)*, Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca (Eds.). Springer, 761–772. https://doi.org/10.1007/BFB0035797

Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. GhostCell: Separating Permissions from Data in Rust. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473597