Nola: Later-Free Ghost State for Verifying Termination in Iris

YUSUKE MATSUSHITA, Kyoto University, Japan
TAKESHI TSUKADA, Chiba University, Japan

Separation logic (SL) has recently evolved at an exciting pace, opening the way to more complex goals, notably soundness proof of Rust's ownership type system and functional verification of Rust programs. In this paper, we address verification of *termination* in the presence of advanced features, especially Rust's ownership types. Perhaps surprisingly, this goal could not be achieved by a simple application of existing studies, which dealt only with *safety* properties. For high-level reasoning about advanced shared mutable state as used in Rust, they used *higher-order ghost state* (i.e., logical state that depends on SL assertions), but in a way that depends on the *later modality*, a fundamental obstacle in verifying termination.

To solve this situation, we propose a novel general framework, nicknamed *Nola*, which achieves later-free higher-order ghost state. Even in the presence of advanced features such as invariants and borrows, Nola enables verifying termination in a natural way, allowing arbitrary induction in the meta-logic. The key idea is to *parameterize* higher-order ghost state, generalizing and subsuming the existing approach. Nola is fully mechanized in Rocq as a library of Iris. Moreover, to demonstrate the power of Nola, we develop *RustHalt*, the first semantic and mechanized foundation for total correctness verification of Rust programs.

CCS Concepts: • **Theory of computation** → **Separation logic**; **Logic and verification**.

Additional Key Words and Phrases: Iris, later modality, step-indexing, higher-order ghost state, shared mutable state, Rust, termination

ACM Reference Format:
Yusuke Matsushita and Takeshi Tsukada. 2025. Nola: Later-Free Ghost State for Verifying Termination in Iris. In *Proceedings of the 46th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '25), June 16–20, 2025, Seoul, South Korea.* ACM, New York, NY, USA, 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Separation logic (SL) [O'Hearn and Pym 1999; O'Hearn et al. 2001; Reynolds 2002; O'Hearn 2004; Brookes 2004] has been very actively studied as the de facto standard, highly scalable logic for reasoning about mutable state [Brookes and O'Hearn 2016; O'Hearn 2019]. Its core idea is to equip propositions with ownership of some mutable state (e.g., the points-to token $\mathtt{r} \mapsto \mathtt{v}$ exclusively owns the memory cell at $\mathtt{r}$). Recently, separation logic has evolved at an exciting pace.

One of the milestones of this line of research is the verification of programs in Rust [Matsakis and Klock 2014]. Its verification is of practical importance, given the growing attention Rust has received in recent years. Furthermore, Rust's complex memory management mechanisms also serve as a challenging benchmark for evaluating the expressiveness of program logic. Jung et al. [2018a] and Dang et al. [2020] proved the soundness of Rust's ownership type system and Matsushita et al. [2022] and Gäher et al. [2024] established functional verification of Rust programs.

Authors' Contact Information: Yusuke Matsushita, Kyoto University, Kyoto, Japan, ymat@fos.kuis.kyoto-u.ac.jp; Takeshi Tsukada, Chiba University, Chiba, Japan, tsukada@math.s.chiba-u.ac.jp.

PLDI '25, Seoul, South Korea

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The above-mentioned Rust verification projects dealt only with *safety* properties, such as memory safety and *partial* correctness, telling about the absence of bad behavior (e.g., invalid memory access) or what happens *if* the program terminates. Perhaps surprisingly, *termination* verification of Rust programs could not be accomplished by a simple application of the existing studies for a well-known but rather technical reason, which we will discuss later.

In this paper, we propose a new general framework, nicknamed *Nola*, built on top of the Iris separation logic framework [Jung et al. 2015, 2018b]. Nola solves existing challenges and enables verification of *termination* of programs in the presence of advanced features such as shared mutable references and Rust's ownership types. With the power of Nola, we have developed *RustHalt*, the first semantic and mechanized foundation for total correctness verification of Rust programs. In this section, we first discuss the existing techniques and remaining challenges (§ 1.1) and then present an overview of our solution, Nola (§ 1.2).

1.1 Existing Techniques and Challenges

Total correctness verification in separation logic. First, we briefly explain an existing framework for total correctness verification in separation logic. We write $\left[P\right] e \left[\lambda v.\, Q_v\right]$ for a total Hoare triple, meaning that if the program e is executed from a state satisfying P, then it is guaranteed to terminate, with $e \hookrightarrow^* v$ for some value v, and the resulting state will satisfy Q_v. Remarkably, we can prove total correctness using induction in the *meta-logic*. For example, if we are developing the total correctness proof in Rocq (formerly known as Coq), we can freely use native induction tactics of Rocq. Such a technique can be found in, e.g., [Charguéraud 2011].

Example 1.1 (Total correctness proof by meta-logic induction). For a simple example, let us consider the following recursive function:

$$\texttt{fn decrloop(r) \{ if *r > 0 \{ r := *r - 1; decrloop(r) \} \}}$$

We have the total Hoare triple $\left[\texttt{r} \mapsto n\right] \texttt{decrloop(r)} \left[\lambda\_.\, \texttt{r} \mapsto 0\right]$ for every natural number $n \in \mathbb{N}$, i.e., `decrloop(r)` terminates setting `r` to 0 whatever the initial natural number stored at `r`. To prove this, we can appeal to the induction on n in the *meta-logic* as follows. For the base case $n = 0$, then the program immediately terminates with the points-to token $\texttt{r} \mapsto 0$. For the step case $n = k + 1$ for $k \in \mathbb{N}$, we start with the state $\texttt{r} \mapsto (k+1)$. We perform the decrement $\texttt{r} := *\texttt{r} - 1$ and reach the same function call `decrloop(r)` with the updated state $\texttt{r} \mapsto k$, where we can use the induction hypothesis for the case $n = k$, which completes the proof.

Notably, we can also easily prove *unbounded* termination, where the number of program steps for termination is unbounded.

Example 1.2 (Proving unbounded termination). For example, we can prove the following assertion: $\left[\texttt{r} \mapsto v\right]$ `*r = ndnat; decrloop(r)` $\left[\lambda\_.\, \texttt{r} \mapsto 0\right]$. It says that `decrloop(r)` terminates, with the value at `r` set to 0, after initializing the value at `r` with a non-deterministic natural number **`ndnat`**. This is an example of unbounded termination, because the number of program steps depends on the output of **`ndnat`**, which is unbounded. The proof goes simply by combining the standard proof rules (especially $\left[\top\right]$ **`ndnat`** $\left[\lambda v.\, v \in \mathbb{N}\right]$ for non-determinism) with the total Hoare assertion that we proved in Example 1.1 by meta-logic induction.

Invariants and higher-order ghost state. Separation logic features the points-to token $\texttt{r} \mapsto v$, which is useful for reasoning about exclusive mutable state, but it cannot be directly *shared* while retaining mutability. To tackle *shared mutable state*, advanced separation logics use *(shared) invariants* [Hobor et al. 2008; Buisse et al. 2011; Svendsen et al. 2013; Svendsen and Birkedal 2014], whose modern usage was established by Iris [Jung et al. 2015, 2018b]. Roughly speaking, the

invariant token $\boxed{P}$ asserts that P is an invariant, i.e., the situation described by the SL proposition $P \in \mathit{iProp}$ is always kept during the program execution.[1] Therefore, when $\boxed{P}$ holds, we can assume P at any moment, but at the same time, we need to ensure that P holds at all times. The key property is that $\boxed{P}$ claims no ownership (even if P claims some); it can be duplicated $\boxed{P} \Leftrightarrow \boxed{P} * \boxed{P}$ and hence be freely shared, especially among multiple threads.

For a simple example, one can express a *shared mutable reference* storing a boolean by:

$$\mathtt{r} : \mathtt{ref\ bool} \triangleq \boxed{\mathtt{r} \mapsto \mathtt{true} \lor \mathtt{r} \mapsto \mathtt{false}} \tag{1.1}$$

It means that the memory cell at $\mathtt{r}$ always stores true or false. We can create this invariant by allocating a reference (e.g., $\lceil \top \rceil\ \mathtt{ref\ true}\ \lceil \lambda \mathtt{r}.\ \boxed{\mathtt{r} \mapsto \mathtt{true} \lor \mathtt{r} \mapsto \mathtt{false}} \rceil$). Under this invariant, we can safely store any boolean to $\mathtt{r}$ (e.g., $\lceil \boxed{\mathtt{r} \mapsto \mathtt{true} \lor \mathtt{r} \mapsto \mathtt{false}} \rceil\ {*}\mathtt{r} = \mathtt{false}\ \lceil \top \rceil$) and load a boolean from $\mathtt{r}$ (e.g., $\lceil \boxed{\mathtt{r} \mapsto \mathtt{true} \lor \mathtt{r} \mapsto \mathtt{false}} \rceil\ {*}\mathtt{r}\ \lceil \lambda \mathtt{v}.\ \mathtt{v} = \mathtt{true} \lor \mathtt{v} = \mathtt{false} \rceil$), and conversely cannot store non-boolean values (e.g., 42) to $\mathtt{r}$.

Also, nested shared mutable references can be naturally expressed, for example:

$$\mathtt{r} : \mathtt{ref\ (ref\ bool)} \triangleq \boxed{\exists \mathtt{s}.\ \mathtt{r} \mapsto \mathtt{s} * \boxed{\mathtt{s} \mapsto \mathtt{true} \lor \mathtt{s} \mapsto \mathtt{false}}} \tag{1.2}$$

Notably, we can nest the invariant token, because the invariant token itself is an SL proposition. Here, the inner reference should be wrapped into the invariant token for it to be sharable.

Another important pattern of shared mutable state is Rust-style *borrows*. It is a mechanism for temporarily borrowing access permissions on some objects, causing a form of sharing between borrowers and lenders. This more complex form of sharing was given a semantic model in separation logic by the *lifetime logic* of *RustBelt* [Jung et al. 2018a; Jung 2020], which proved the memory and thread safety guarantee of Rust's ownership type system. The lifetime logic, or its borrow machinery, has proved vital for semantically modeling Rust's ownership types, especially mutable $\&\alpha\ \mathtt{mut}\ \mathtt{T}$ and shared $\&\alpha\ \mathtt{T}$ references, and verify key properties about Rust [Dang et al. 2020; Yanovski et al. 2021; Matsushita et al. 2022; Gäher et al. 2024].

Invariants and borrows mentioned above are examples of a more general framework known as *higher-order ghost state* [Jung et al. 2016, 2018b], logical state whose structure depends on SL propositions. It has brought success to verification of challenging goals, such as the non-interference of an information flow control system [Gregersen et al. 2021], the purity of Haskell's ST monad [Timany et al. 2018; Jacobs et al. 2022], and key properties about Rust [Jung et al. 2018a; Dang et al. 2020; Matsushita et al. 2022; Gäher et al. 2024].

The naive access rule and its unsoundness. The intuition about accessing invariants described above can be formalized using the following plausible rule:

$$\frac{\lceil P * Q \rceil\ \mathtt{ae}\ \lceil \lambda \mathtt{v}.\ P * \Psi\,\mathtt{v} \rceil}{\lceil \boxed{P} * Q \rceil\ \mathtt{ae}\ \lceil \Psi \rceil}\ \textsc{thoare-inv-naive}\ ?$$

This rule means that, if the invariant $\boxed{P}$ is in the precondition, we can assume P in the precondition but we should ensure that P holds after the execution of e.

Unfortunately, this rule is unsound, if any arbitrary logical formula can be used as P. This fact can be demonstrated using the well-known technique of creating loops with functional references, known as *Landin's knot*:

Paradox 1.3 (Naive Invariant Paradox on Landin's Knot). *Let* `landin` *be the following program:*

$$\mathtt{let\ r\ =\ ref\ fn\,()\,\{\}\ in\ \ {*}r\ =\ fn\,()\,\{\,({*}r)\,()\,\};\ \ ({*}r)\,()}$$

[1] In this paper, we use *iProp* for the set of SL propositions, following the convention of Iris ('i' actually stands for Iris).

$$\frac{\left[P * Q\right]\ \mathtt{ae}\ \left[\lambda \mathtt{v}.\ P * \Psi\,\mathtt{v}\right]}{\left[\boxed{P} * Q\right]\ \mathtt{ae}\ \left[\Psi\right]}\ \textsc{thoare-inv-naive}\ ? \qquad \frac{\left[\triangleright P * Q\right]\ \mathtt{ae}\ \left[\lambda \mathtt{v}.\ \triangleright P * \Psi\,\mathtt{v}\right]}{\left[\boxed{\triangleright P} * Q\right]\ \mathtt{ae}\ \left[\Psi\right]}\ \textsc{thoare-linv}$$

Fig. 1. Invariant access rules: Naive (unsound) vs. Later-weakened (existing).

Also, let $\mathrm{reff}_{\mathtt{r}} \triangleq \exists \mathtt{f}.\ \mathtt{r} \mapsto \mathtt{f} * \left[\top\right] \mathtt{f}() \left[\top\right]$, *which means that* `r` *points to a terminating function. Suppose that we can use the naive later-free access rule* THOARE-INV-NAIVE *for an invariant token* $\boxed{\mathrm{reff}_{\mathtt{r}}}$. *Then we can wrongly prove the termination of Landin's knot:* $\left[\top\right]$ `landin` $\left[\top\right]$.

Proof. By allocating the invariant $\boxed{\mathrm{reff}_{\mathtt{r}}}$ after the initialization `let r = ref fn () {}`. The naive rule THOARE-INV-NAIVE (paradoxically) allows proving the termination $\left[\boxed{\mathrm{reff}_{\mathtt{r}}}\right]$ `(*r)()` $\left[\top\right]$, hence justifying the update `*r = fn () { (*r) }`. □

Later modality. The existing approach to justify higher-order ghost state resorted to the *later modality* $\triangleright : \mathit{iProp} \to \mathit{iProp}$ for soundness [Nakano 2000; Appel and McAllester 2001]. For example, instead of THOARE-INV-NAIVE, the existing approach used the rule THOARE-LINV in Fig. 1, with the shared content P weakened by the later modality $\triangleright$. To clarify this fact, we also write the invariant token as $\boxed{\triangleright P}$ (with a gray later modality $\triangleright$ on P) instead of $\boxed{P}$. For example, the nested reference (1.2) is actually modeled as follows, if we explicitly write the later modality:

$$\mathtt{r} : \mathtt{ref}\ (\mathtt{ref}\ \mathtt{bool}) \quad \triangleq \quad \boxed{\triangleright\ \exists \mathtt{s}.\ \mathtt{r} \mapsto \mathtt{s} * \boxed{\triangleright\ (\mathtt{s} \mapsto \mathtt{true} \vee \mathtt{s} \mapsto \mathtt{false})}} \tag{1.3}$$

We have the later-weakened proof rule THOARE-LINV in Fig. 1. It provides access to the later-weakened shared content $\triangleright P$ on executing an atomic expression `ae` (an expression that takes only one program step).

The later modality in THOARE-INV is necessary for soundness, but it becomes an obstacle for reasoning. Especially, for *nested references* like (1.3), the existing approach suffers from the later modality that is stuck to inner references and blocks further access. For example:

$$\left[\ \boxed{\triangleright\ \exists \mathtt{s}.\ \mathtt{r} \mapsto \mathtt{s} * \boxed{\triangleright \Phi\,\mathtt{s}}}\ \right]\ *\mathtt{r}\ \left[\ \lambda \mathtt{s}.\ \triangleright\ \boxed{\triangleright \Phi\,\mathtt{s}}\ \right].$$

The postcondition only ensures that the inner reference `s` satisfies $\triangleright\ \boxed{\triangleright \Phi\,\mathtt{s}}$, not $\boxed{\triangleright \Phi\,\mathtt{s}}$, and the later modality $\triangleright$ here blocks access to the content of the inner invariant.

Step-indexing and its limitations. To strip off the later modality, the existing studies used a technique called *step-indexing* [Nakano 2000; Appel and McAllester 2001]. Step-indexing ties program steps to the later modality in some way, allowing the later modality to be stripped off as the program executes.

However, step-indexing is actually a fundamental obstacle for the purpose of proving *termination*, or more generally *liveness* properties (e.g., termination preservation).[2] This problem has been well known. For example, the paper [Gäher et al. 2022] on the Simuliris separation logic, says as follows (§1.1, references relabeled):

> 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.

[2] In the scope of this paper, Nola supports only a simple type of (non-fair) termination, although we speculate about richer applications of Nola. Please see Current status and future applications in § 1.2 for detailed discussions.

$$\frac{\left[[\![P]\!] * Q\right]\ \mathsf{ae}\ \left[\lambda \mathsf{v}.\ [\![P]\!] * \Psi\,\mathsf{v}\right]^{\mathrm{Winv}\,[\![\,]\!]}}{\left[\boxed{P} * Q\right]\ \mathsf{ae}\ \left[\Psi\right]^{\mathrm{Winv}\,[\![\,]\!]}}\ \textsc{thoare-inv} \qquad \begin{array}{l} P \in \mathit{Fml} \\ [\![\,]\!] : \mathit{Fml} \to \mathit{iProp} \end{array}$$

Fig. 2. Invariant access rule: Nola (ours, key points highlighted).

Spies et al. [2021a] explains this fundamental limitation in relation to the lack of the *existential property* in Iris: $\vdash \exists n \in \mathbb{N}.\, P_n$ (existential quantification in Iris) does not imply $\vdash P_n$ for some $n \in \mathbb{N}$ (meta-level existential quantification).

Due to this technical challenge, existing approaches to termination can be categorized into the following three types:

(1) Use fragments where the later modality is unnecessary. In Gäher et al. [2022], verification is conducted without using invariant connective $\boxed{P}$ and any other higher-order ghost states. This also applies to other recent separation logics targeting liveness properties [Song et al. 2023; Lee et al. 2023].

(2) Give a *finite* bound of the number of execution steps. Mével et al. [2019] developed in Iris the machinery of *time credits* [Atkey 2010; Charguéraud and Pottier 2015, 2019], allowing reasoning about bounded termination. To prove termination in this logic, the verifier must explicitly specify the *finite* number of execution steps.

(3) Use *transfinite* step-indexing [Spies et al. 2021b] instead of finite step-indexing (by natural numbers) used by Iris. This approach gave rise to a variant of Iris, Transfinite Iris [Spies et al. 2021a]. In particular, Transfinite Iris allows extending the idea of time credits to use *transfinite* bounds (by ordinal numbers) for proving termination.

Unfortunately, none of these approaches are sufficient for verifying the termination of programs in the presence of complex memory management mechanisms, such as Rust's ownership types. Rust's memory invariants are complex, and there is no known way to reason about them without using higher-order ghost state, making the approach (1) infeasible. Also, the approach (2) cannot be applied to programs where the number of steps for termination cannot be bounded: for example, the number of steps for the program `*r = *s; decrloop(r)` (using `decrloop` from Example 1.1) to terminate cannot be bounded when `s` is a *shared mutable reference* (e.g., `s`$: \mathbf{ref\,int} \triangleq \boxed{\triangleright \exists n \in \mathbb{Z}.\, \mathtt{s} \mapsto n}$) to which other processes can freely write unboundedly large integers. Moreover, explicitly providing a finite bound on termination is not always easy. The approach (3) may seem more promising, but in reality, it involves subtle technical challenges. Under transfinite indexing, the later modality loses the vital commutativity laws with the existential quantifier ($\triangleright\,(\exists a.\, P_a) \;\Leftrightarrow\; \exists a.\, \triangleright\, P_a$) and with the separating conjunction ($\triangleright\,(P * Q) \Leftrightarrow \triangleright\, P * \triangleright\, Q$). Many Iris developments, notably including RustBelt's *lifetime logic* [Jung et al. 2018a], rely on these commutativity laws, and thus unfortunately stop working if ported to Transfinite Iris. Therefore, Transfinite Iris cannot be applied to termination verification of Rust programs under RustBelt's approach. Also, termination verification by transfinite time credits typically requires careful management of transfinite bounds inside the *separation logic*, which is not as natural and smooth as verification by arbitrary induction in the *meta-logic*.

In summary, there is a challenge in reconciling termination verification (allowing meta-logic induction) and higher-order ghost state (such as invariants and borrows).

1.2 Our Solution, Nola

As a novel approach to solve this challenge, we propose *Nola*.[3] Intuitively, in Nola, we can customize the later modality $\triangleright$: *iProp* $\to$ *iProp* originally used for higher-order ghost state into a user-defined

[3] Its name comes from 'No later' and the nickname of New Orleans, Louisiana, in memory of POPL 2020 held in that city.

function (namely the *semantics* $[\![\ ]\!]\colon \mathit{Fml} \to \mathit{iProp}$) with a user-defined substitute of SL (separation logic) propositions *iProp* (namely *SL formulas Fml*). A well-behaved variant of the later modality leaves a larger class of SL propositions (including the invariant $\boxed{P}$) unweakened, and thus enables *termination* verification. Now we overview Nola in more detail.

Core idea: Parameterization. Nola's core idea is simple. It introduces two new user-defined *parameters* to higher-order ghost state. The first is the set of *SL formulas*, or the data type that encodes SL propositions, named *Fml*. The second is the *semantics* $[\![\ ]\!]\colon \mathit{Fml} \to \mathit{iProp}$, i.e., how to decode *Fml*. Intuitively, they are user-defined replacements for *iProp* and the later modality $\triangleright$, respectively. By this, Nola attains the *later-free* invariant access rule THOARE-INV shown in Fig. 2, which is obtained by replacing $P \in \mathit{iProp}$ and $\triangleright$ in THOARE-LINV (Fig. 1) with $P \in \mathit{Fml}$ and $[\![\ ]\!]$.

This parameterization brings rich expressivity. For example, we can set *Fml* to a *predicative* (or *syntactic*) data type. In particular, we can add to *Fml* the constructor $\boxed{P}$ interpreted as the *later-free* invariant token $[\![\boxed{P}]\!] \triangleq \boxed{P}$. This achieves termination verification in the presence of nested references. For example, to the nested reference (1.3), Nola can give the following *later-free* model that works in total correctness verification (here we use *Fml* of Fig. 6, § 2.2):

$$\mathtt{r} : \mathtt{ref}\ (\mathtt{ref}\ \mathtt{bool}) \quad \triangleq \quad \boxed{\exists\, \mathtt{s}.\ \mathtt{r} \mapsto \mathtt{s} \ *\ \boxed{\mathtt{s} \mapsto \mathsf{true} \ \vee\ \mathtt{s} \mapsto \mathsf{false}}} \tag{1.4}$$

We can even set *Fml* to a hybrid of predicative and impredicative (or semantic) constructions, where we can express any SL propositions under the later modality $\triangleright P$, subsuming the existing approach.

Key aspect: Soundness. If there were no restriction on *Fml* and $[\![\ ]\!]$ at all, the logic would be unsound, as one could set $\mathit{Fml} \triangleq_? \mathit{iProp}$ and $[\![P]\!] \triangleq_? P$ to have naive later-free higher-order ghost state of THOARE-INV-NAIVE. Fortunately, Nola is designed to be sound just by simple restrictions on *Fml* and $[\![\ ]\!]$. We discuss this in more details in § 2.

Our contributions. Our contributions are in summary as follows:

- We propose Nola, a novel approach to higher-order ghost state that clears the later modality via parameterization. We present Nola's later-free versions of Iris's *invariants* [Jung et al. 2015] and RustBelt's *borrows* [Jung et al. 2018a]. We demonstrate their power, especially in the context of *termination verification* of programs with shared mutable state.
- We closely analyze the expressivity of Nola's approach, especially in light of Paradox 1.3.
- We propose a general technique to semantically alter SL formulas of higher-order ghost state, which builds on a general mathematical theory.
- We have fully mechanized Nola in Rocq (formerly known as Coq) as a library of Iris [Matsushita and Tsukada 2025]. Moreover, we have developed *RustHalt*, the first semantic and mechanized foundation for total correctness verification of Rust programs.

Limitations. Naturally, the later-free ghost state of Nola has some limitations. Clearly, storing arbitrary SL propositions $P \in \mathit{iProp}$ in invariants leads to the unsoundness Paradox 1.3 of the naive rule THOARE-INV-NAIVE. Therefore, even in Nola, not all *iProp* can be handled in a later-free manner. Typical constructs that Nola cannot handle in a later-free manner include total Hoare triples, view shifts, and impredicative quantifiers; please see § 4 for more details. Handling all *iProp* requires *impredicativity*, which is why existing approaches resorted to the later modality $\triangleright$.

Still, notably, arbitrarily SL propositions (which can include impredicative quantifiers etc.) can be used for reasoning in separation logic just as usual, outside of higher-order ghost state. Also, as mentioned above, any SL propositions $P \in \mathit{iProp}$ can be stored in ghost state like invariants under the guard of the later modality (i.e., in forms like $\boxed{\triangleright P}$), subsuming existing approaches.

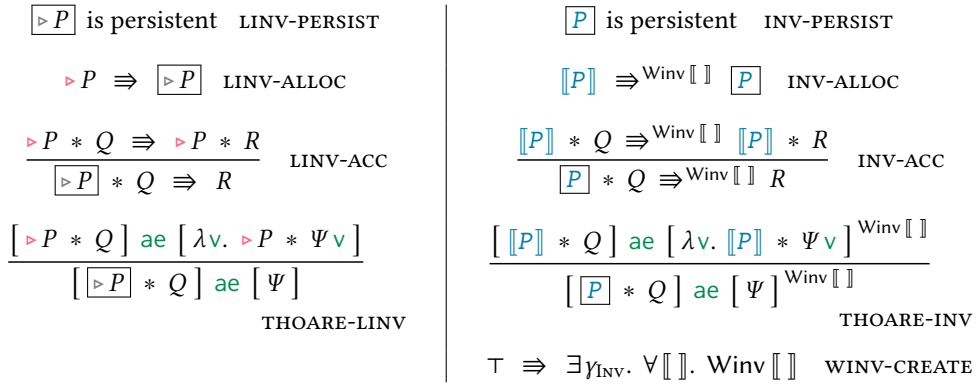

Fig. 3. Selected invariant proof rules: Later-weakened vs. Nola (key points highlighted).

Current status and future applications. Currently, in the scope of this paper, Nola supports only a simple type of termination. For concurrent programs, the current program logic does *not* assume *fair* thread scheduling for termination. Thus, the termination proof currently requires that each thread eventually terminates on its own (independent of actions of other threads), since the scheduler is allowed to run only that thread and no other threads.

Nevertheless, we speculate that Nola can be used for richer *liveness* properties. One exciting future direction is to extend existing separation logics for advanced liveness properties, such as Simuliris [Gäher et al. 2022], Conditional Contextual Refinement [Song et al. 2023], and Fairness Logic [Lee et al. 2023], with Nola's higher-order ghost state to reason about advanced features, especially Rust's ownership types. In particular, we hope this can possibly reveal the currently unknown formal relationship of Rust's ownership type system to Simuliris, which verifies advanced concurrent program optimizations for Rust under the model of Stacked Borrows [Jung et al. 2020a]. Another interesting future direction is to extend existing SL-based automated verification platforms such as VeriFast [Jacobs et al. 2011] and Viper [Müller et al. 2016] with Nola's later-free higher-order ghost state, not giving up termination and liveness verification.

Paper organization. Section 2 presents an overview of Nola's later-free invariants and Section 3 presents its model. Section 4 analyzes Nola's expressivity. Section 5 presents Nola's later-free borrows. Section 6 presents a technique to semantically alter SL formulas of higher-order ghost state. Section 7 discusses RustHalt, a semantic foundation for total correctness verification of Rust programs. Section 8 reports on our Rocq mechanization. Section 9 discusses related work.

2 Nola's Later-Free Invariant

Now we present the interface of Nola's later-free invariants as a central example of higher-order ghost state (§ 2.1), present a useful construction of the SL formulas and their semantics (§ 2.2), and show examples of using it for termination verification (§ 2.3). Here we assume basic knowledge of Iris. Please refer to [Jung et al. 2018b] etc. for introductory materials.

2.1 Interface of Nola's Invariant

Proof rules. Figure 3 shows selected proof rules of Nola's later-free invariants in parallel to the existing approach's later-weakened invariant.[4] As previewed in § 1.2, Nola's invariant machinery

[4] For simplicity, we omit the namespace $\mathcal{N}$ and mask $\mathcal{E}$ parameters in the presentation, following the convention. Technically, these parameters are used to prohibit access races on the contents of invariants.

$$\frac{P \Rrightarrow^W P' \qquad \left[P'\right]\ \mathsf{e}\ \left[\Psi\right]^W}{\left[P\right]\ \mathsf{e}\ \left[\Psi\right]^W}\ \textsc{vs-thoare} \qquad \frac{\left[P\right]\ \mathsf{e}\ \left[\Psi'\right]^W \qquad \forall \mathsf{v}.\ \Psi'\,\mathsf{v} \Rrightarrow^W \Psi\,\mathsf{v}}{\left[P\right]\ \mathsf{e}\ \left[\Psi\right]^W}\ \textsc{thoare-vs}$$

$$\frac{P \Rrightarrow^W Q}{P \Rrightarrow^{W*W'} Q}\ \textsc{vs-expand} \qquad \frac{\left[P\right]\ \mathsf{e}\ \left[\Psi\right]^W}{\left[P\right]\ \mathsf{e}\ \left[\Psi\right]^{W*W'}}\ \textsc{thoare-expand}$$

Fig. 4. Selected proof rules for view shifts and Hoare triples.

is parameterized over the set of SL formulas $\mathit{Fml} \ni P, Q$ and the semantics $[\![\ ]\!]\colon \mathit{Fml} \to \mathit{iProp}$, which were hard-coded in the existing approach.

Nola's invariant token $\boxed{P} \in \mathit{iProp}$ asserts that the situation described by the SL formula $P \in \mathit{Fml}$ holds as an invariant. The invariant token $\boxed{P}$ is *persistent* (INV-PERSIST), which intuitively means that it claims *no ownership* and always holds regardless of state mutation. A persistent proposition P can be duplicated ($P \Leftrightarrow P * P$) and hence be freely shared, especially among multiple threads. Nola's invariants are accessed with the *view shift* $P \Rrightarrow^{\mathsf{Winv}[\![\ ]\!]} Q$, roughly meaning a logical step from P to Q under a global imaginary store called the *world satisfaction* $\mathsf{Winv}[\![\ ]\!]$. Hoare triples $\left[P\right]\ \mathsf{e}\ \left[\Psi\right]^{\mathsf{Winv}[\![\ ]\!]}$ absorb the view shift at any point of execution (see VS-THOARE, THOARE-VS in Fig. 4). One can create a new invariant $\boxed{P}$ for an SL formula $P \in \mathit{Fml}$ by storing its semantics $[\![P]\!] \in \mathit{iProp}$ (INV-ALLOC). Under the invariant $\boxed{P}$, one can get access to the shared content $[\![P]\!]$ via a view shift (LINV-ACC) and thus via a total Hoare triple (THOARE-LINV).

A key to the adequacy involving Nola's invariants is WINV-CREATE, which creates the world satisfaction $\mathsf{Winv}[\![\ ]\!]$ before any invariants are established (see Theorem 2.1 to grasp how this works). Here, it freshly takes a ghost name γ_{Inv}, upon which the invariant token $\boxed{P}$ and the world satisfaction $\mathsf{Winv}[\![\ ]\!]$ implicitly depend. Technically, the semantics $[\![\ ]\!]$ can depend on the ghost name γ_{Inv} thanks to the rule's universal quantification over the semantics $[\![\ ]\!]\colon \mathit{Fml} \to \mathit{iProp}$.

Parameterized view shifts and Hoare triples. We also *parameterize* view shifts and Hoare triples over the custom *world satisfaction* $W \in \mathit{iProp}$, which was hard-coded to a pre-installed one Wlinv (see Fig. 8) in the existing approach. The world satisfaction serves as a global imaginary store for shared contents of higher-order ghost state. Technically, the view shift $\Rrightarrow^W$ with a custom world satisfaction $W \in \mathit{iProp}$ is derived from a plain view shift $\Rrightarrow$ as follows:[5]

$$P \Rrightarrow^W Q \quad \triangleq \quad P * W \Rrightarrow Q * W.$$

Hoare triples $\left[P\right]\ \mathsf{e}\ \left[\Psi\right]^W$ are modeled using view shifts $\Rrightarrow^W$. As shown in Fig. 4, Hoare triples can absorb view shifts of the same world satisfaction (VS-THOARE, THOARE-VS), and world satisfactions can be merged with the separating conjunction $*$ (VS-EXPAND, THOARE-EXPAND).

The parameterized total Hoare triple naturally supports the standard proof rules for total correctness verification. For example, Examples 1.1 and 1.2 work for the parameterized total Hoare triple. In our Rocq mechanization, we prove counterparts to the CFML-style proof rules [Charguéraud 2011] found in [Charguéraud 2025, Appendix - The Full Construction].

Moreover, we provide new adequacy theorems for parameterized Hoare triples. In particular, we have the following adequacy theorem for the parameterized total Hoare triple:

[5] For ease of implementation, we reuse the existing work's view shift $\Rrightarrow$ with the pre-installed world satisfaction Wlinv, instead of directly replacing Wlinv with the parameter W.

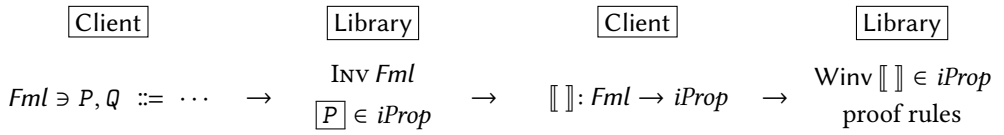

Fig. 5. Dependencies between the components from Nola's client and invariant library.

Theorem 2.1 (Termination Adequacy of the Parameterized Total Hoare Triple). *An expression* e *always terminates if the following holds:* $\top \Rrightarrow \exists W.\ W * \left[\top\right] \mathsf{e} \left[\top\right]^{W}$.

Importantly, the theorem requires the creation of the custom world satisfaction W, which can be chosen freely. We can discharge this requirement by creation rules for world satisfactions such as WINV-CREATE. For example, using Theorem 2.1 and WINV-CREATE, we can prove the following: $\left[\top\right] \mathsf{e} \left[\top\right]^{\mathsf{Winv}\,⟦\,⟧}$ implies that e always terminates.

Dependencies and restriction. For soundness, one should be careful about the following dependencies between definitions by the client and Nola's invariant library, as summarized in Fig. 5:

1. The client builds the set of *SL formulas Fml*.
2. Nola's invariant library provides the *camera* Inv *Fml* for the invariant machinery. Technically, a camera is an algebra of state resources used in Iris (a variant of partial commutative monoid, or PCM). Roughly speaking, Iris propositions are a predicate $iProp \triangleq State \to Prop^{\sim}$ over the state resources $State \triangleq \prod_i \mathcal{A}_i$ defined as the product over a user-custom family of cameras $\Sigma \triangleq (\mathcal{A}_i)_i$. The library also provides the *invariant token* $\boxed{P} \in iProp$ when the camera Inv *Fml* is included in Σ. Notably, the token $\boxed{P}$ does *not* depend on the semantics $⟦\,⟧$.
3. The client can now build the *semantics* $⟦\,⟧ : Fml \to iProp$ of the SL formulas *Fml*, which typically depends on the invariant token $\boxed{P}$.
4. Finally, Nola's invariant library gives the *world satisfaction* $\mathsf{Winv}\,⟦\,⟧ \in iProp$. The library also provides the later-free *proof rules* (Fig. 3) for view shifts and Hoare triples with the world satisfaction $\mathsf{Winv}\,⟦\,⟧$, which depends on the whole semantics $⟦\,⟧$.

We also have simple restrictions on the parameters *Fml* and $⟦\,⟧$ for soundness. First, the set of formulas *Fml* may refer to *iProp*, but in a *contractive* way, or roughly speaking, under the guard of the *later functor* $\triangleright$. The *later functor* $\triangleright A$ [Gianantonio and Miculan 2002; Birkedal et al. 2012] is a data type of items of the form next a (for $a \in A$) for which the indexed equality $\tilde{=}$ is weakened by the later modality $\triangleright$: $(\mathsf{next}\ a) \tilde{=} (\mathsf{next}\ a') \triangleq \triangleright (a \tilde{=} a')$. This makes the domain equation for *Fml* and *iProp* solvable by [America and Rutten 1989]. Second, the semantics $⟦\,⟧ : Fml \to iProp$ should be *non-expansive*, i.e., respect the indexed equality: $P \tilde{=} Q \to ⟦P⟧ \tilde{=} ⟦Q⟧$. This is a key to the soundness of the access rules like INV-ACC and THOARE-INV (see the proof of Theorem 3.1).

2.2 Constructing the SL Formulas and Their Semantics

To use Nola's higher-order ghost state like invariants, we should instantiate the parameters *Fml* and $⟦\,⟧$. For that, we can design *Fml* as the set of *abstract syntax trees* encoding SL propositions and define the semantics $⟦\,⟧$ by *induction*. A typical example is shown in Fig. 6.

For a simple case, binary logical connectives such as $*$, $-\!\!*$ and $\vee$ can be represented by binary constructors $*, -\!\!*, \vee : Fml \times Fml \to Fml$, and the semantics $⟦\,⟧$ over them can be defined naturally. Also, the quantifiers $\forall, \exists$ can be encoded by higher-order abstract syntax, i.e., constructors $\forall, \exists$

$$\mathit{Fml} \ni P, Q \quad ::=_{\nu,\mu} \quad P * Q \mid P \mathrel{-\!\!*} Q \mid P \vee Q \mid \forall_A \Phi \mid \exists_A \Phi \mid \phi \ \ (\phi \in \mathit{Prop}) \mid r \mapsto\!\!\!\!\rightarrow v \mid \boxed{P} \ \ (P \in_\nu \mathit{Fml}) \mid \check{\triangleright}\, \hat{P} \ \ (\hat{P} \in \blacktriangleright \mathit{iProp}) \qquad \triangleright P \triangleq \check{\triangleright}\, (\mathsf{next}\, P)$$

$$[\![P * Q]\!] \triangleq [\![P]\!] * [\![Q]\!] \qquad [\![P \mathrel{-\!\!*} Q]\!] \triangleq [\![P]\!] \mathrel{-\!\!*} [\![Q]\!] \qquad [\![P \vee Q]\!] \triangleq [\![P]\!] \vee [\![Q]\!]$$

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

$$[\![r \mapsto\!\!\!\!\rightarrow v]\!] \triangleq r \mapsto v \qquad [\![\boxed{P}]\!] \triangleq \boxed{P} \qquad [\![\check{\triangleright}\, \hat{P}]\!] \triangleq \check{\triangleright}\, \hat{P} \qquad \check{\triangleright}\, (\mathsf{next}\, P) \triangleq \triangleright P$$

Fig. 6. Construction of *Fml* and $[\![\ ]\!]$ for Nola's invariants.

taking the mapping $\Phi : A \to \mathit{Fml}$ over the domain set A.[6] Any pure proposition $\phi \in \mathit{Prop}$ can be embedded into *Fml*. We can also support basic tokens such as the points-to token $r \mapsto v$ by adding a constructor $r \mapsto\!\!\!\!\rightarrow v$ to *Fml*. In general, we can freely *extend Fml* and $[\![\ ]\!]$ by adding new constructors to *Fml*, as long as the semantics $[\![\ ]\!]$ is well-defined for those constructors (we have a limitation due to the paradoxes like Paradox 1.3; we closely analyze the expressivity later in § 4).

Remarkably, we can encode the invariant token as a constructor $\boxed{P}$ in *Fml*. This enables nesting invariants in a *later-free* way, like the nested reference (1.4) in § 1.2. Thanks to the invariant token $\boxed{P}$ *not* depending on the semantics $[\![\ ]\!]$ at all, the equation $[\![\boxed{P}]\!] \triangleq \boxed{P}$ is well-defined. Because this is well-defined even if P is not structurally smaller than $\boxed{P}$, technically, we can make *Fml* a *coinductive-inductive* data type where the constructor $\boxed{P}$ is coinductive, i.e., guarding (as expressed with ν in Fig. 6). This allows construction of *infinite* syntax trees (e.g., (list) in § 2.3).

Subsuming the existing approach. Notably, we can also add an *impredicative*, *semantic* constructor $\triangleright P \in \mathit{Fml}$, embedding any SL proposition $P \in \mathit{iProp}$ into *Fml* under the later modality, not only predicative, syntactic constructors independent of *iProp* discussed above. By this, Nola pleasantly subsumes the existing approach: we can freely express $\triangleright P$ in *Fml* without adding to *Fml* constructors specifically designed for any $P \in \mathit{iProp}$.

For this, we add to *Fml* a constructor $\check{\triangleright}\, \hat{P}$ for $\hat{P} \in \blacktriangleright \mathit{iProp}$, from which we derive $\triangleright P \triangleq \check{\triangleright}\, (\mathsf{next}\, P)$ (for $P \in \mathit{iProp}$). Technically, the guard of the later functor $\blacktriangleright$ here makes *Fml contractive* over *iProp*, which is crucial for soundness. We also define the semantics of $\check{\triangleright}$ by $\check{\triangleright} : \blacktriangleright \mathit{iProp} \to \mathit{iProp}$, which is the composite of the later modality $\triangleright$ and the inverse of next. Technically, the function $\triangleright$ is *non-expansive*, and this makes $[\![\ ]\!]$ non-expansive, which is vital for soundness.

Extensible construction of *Fml* and $[\![\ ]\!]$. Actually, Nola provides a general library for the data type $\mathit{Fml}_{\mathit{Con}}$ with the interpretation $[\![\ ]\!]_{\mathit{Con}}$ that is *parameterized* over the choice of the set of constructors *Con*, just like Martin-Löf's W-types [Martin-Löf 1982]. This library allows further *extensibility* in constructing the syntax. Each development using Nola can be parameterized by the constructor information *Con* under premises on what constructors should be in *Con* for the proof. We can get a closed proof by instantiating *Con* to what satisfies the premises, which is easy and can be automated. This is pretty like how Iris achieves extensibility by parameterization over the family of cameras for ghost state Σ. Our library is remarkable in allowing both inductive and coinductive constructors while retaining extensibility. Technically, we model coinductive data types using an *indexing* (separate from the one used for the later modality $\triangleright$), based on the guarded type theory [Birkedal et al. 2010, 2012], which enables *semantic* reasoning about the productivity of coinductive

[6] For universe consistency, the domain set A here should be taken from a universe smaller than that of *Fml*. This rules out impredicative quantification, but that is a reasonable limitation due to paradoxes (see § 4).

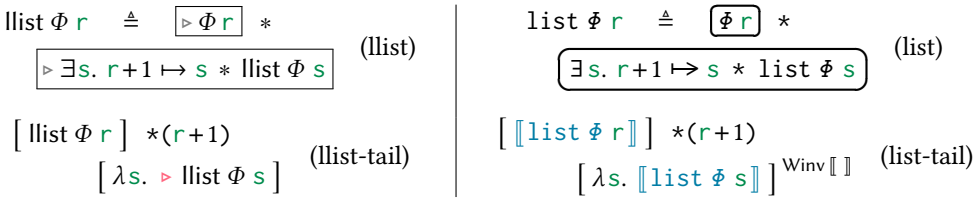

Fig. 7. Models and tail access rules for shared mutable lists: Later-weakened vs. Nola.

definitions. Technically, this is crucial for supporting *recursive types*, especially in our primary case study, RustHalt (§ 7). Please refer to our Rocq mechanization for the details of this.

2.3 Verification Examples

As a simple, interesting target of verification, we consider *shared mutable infinite singly linked lists*, which are modeled by *infinitely nested invariants*.

First, we consider the existing approach. We can define the SL proposition $\mathsf{llist}\ \Phi\ \mathtt{r} \in \mathit{iProp}$ for a shared mutable infinite list starting at $\mathtt{r}$ whose elements satisfy the data invariant $\Phi\colon \mathit{Addr} \to \mathit{iProp}$, by the recursive definition (llist) in Fig. 7, which has a unique solution by the contractiveness of $\boxed{\triangleright -}$. Here, *nested invariants* are effectively used to make every part of lists sharable and mutable. Unfortunately, in the existing approach, each reference to the head and the tail is weakened by the *later modality* $\triangleright$, which blocks *termination* verification. In particular, as shown in (llist-tail) in Fig. 7, when we access the tail of the list $\mathsf{llist}\ \Phi\ \mathtt{s}$, it is weakened by the later modality due to the later-weakened rule THOARE-LINV. There is no chance to strip off the later modality in the non-step-indexed total Hoare triple (recall the discussion in § 1.1).

Nola's later-free invariants solve this situation. The recursive definition (list) in Fig. 7 gives the SL *formula* $\mathtt{list}\ \dot{\Phi}\ \mathtt{r} \in \mathit{Fml}$ for shared mutable singly linked lists,[7] which has a unique solution as an *infinite* tree by the guard of the invariant constructor. Here, we use the design of *Fml* and $⟦\,⟧$ by Fig. 6, with the shorthand $\exists\, a.\, P_a \triangleq \exists(\lambda a.\, P_a)$. Now each reference to the head and the tail is *not* weakened by the later modality, thanks to Nola. As shown in (list-tail) in Fig. 7, we can directly access the tail of the list $⟦\mathtt{list}\ \Phi\ \mathtt{s}⟧$ *without* the later modality thanks to the later-free rule THOARE-INV. This naturally enables *termination* verification.

For example, let us consider the following function $\mathtt{iterc(f,c,r)}$ for iterating over the list:

$$\mathtt{fn\ iterc(f,c,r)\ \{\ if\ *c > 0\ \{\ f(r);\ *c = *c - 1;\ iterc(f,c,*(r+1))\ \}\ \}}.$$

It applies the function $\mathtt{f}$ to the first $\mathtt{*c}$ elements of the given list at $\mathtt{r}$, where $\mathtt{c}$ stores a counter.

Remarkably, using Nola's invariants, we can verify the following *total* Hoare triple assertion, notably for *any SL parameterized formula* $\dot{\Phi}\colon \mathit{Addr} \to \mathit{Fml}$ and *any function* $\mathtt{f}$:

$$\frac{\forall \mathtt{r}.\ \left[\ \boxed{\dot{\Phi}\ \mathtt{r}}\ \right]\ \mathtt{f(r)}\ \left[\ \top\ \right]^{\mathsf{Winv}⟦\,⟧}}{\left[\ ⟦\mathtt{list}\ \dot{\Phi}\ \mathtt{r}⟧ * \mathtt{c} \mapsto n\ \right]\ \mathtt{iterc(f,c,r)}\ \left[\ \mathtt{c} \mapsto 0\ \right]^{\mathsf{Winv}⟦\,⟧}} \tag{2.1}$$

This says that the function $\mathtt{iterc(f,c,r)}$ terminates on a list at $\mathtt{r}$ under the premise that $\mathtt{f}$ safely terminates under the data invariant $\dot{\Phi}$. For example, the premise is satisfied for $\dot{\Phi}\ \mathtt{r} \triangleq \exists k.\, \mathtt{r} \mapsto 3k$ and $\mathtt{f} \triangleq \mathtt{fn\ (r)\ \{\ *r = (*r)+3\ \}}$, i.e., when each element is a multiple of three and $\mathtt{f(r)}$ increments the integer stored at $\mathtt{r}$ by three. The proof of (2.1) goes very naturally by *meta-level induction* over the natural number $n \in \mathbb{N}$ stored at the counter $\mathtt{c}$. This is thanks to the *later-free* tail access

[7] Technically, we can create $⟦\mathtt{list}\ \dot{\mathtt{r}}⟧$ from cyclic references using an advanced variant of INV-ALLOC. Please refer to our Rocq mechanization for the details.

$$\textsc{lInv} \triangleq \textsc{Auth}\left(\mathbb{N} \xrightarrow{\text{fin}} \textsc{Ag}\left(\triangleright\, \mathit{iProp}\right)\right) \qquad \textsc{Inv}\ \mathit{Fml} \triangleq \textsc{Auth}\left(\mathbb{N} \xrightarrow{\text{fin}} \textsc{Ag}\ \mathit{Fml}\right)$$

$$\boxed{\triangleright P} \triangleq \exists \iota.\ \boxed{\circ\,[\iota \leftarrow \mathsf{ag}\,(\mathsf{next}\,P)]}^{\gamma_{\textsc{lInv}}} \qquad \boxed{P} \triangleq \exists \iota.\ \boxed{\circ\,[\iota \leftarrow \mathsf{ag}\,P]}^{\gamma_{\textsc{Inv}}}$$

$$\mathsf{Wlinv} \triangleq \exists\, \hat{I} : \mathbb{N} \xrightarrow{\text{fin}} \triangleright\, \mathit{iProp}. \qquad \mathsf{Winv}\ [\![\,]\!] \triangleq \exists\, I : \mathbb{N} \xrightarrow{\text{fin}} \mathit{Fml}.$$

$$\boxed{\bullet\,\mathsf{ag}\,\hat{I}}^{\gamma_{\textsc{lInv}}} * \mathop{\Large *}_{\iota \in \mathrm{dom}\,\hat{I}} \left(\left(\check{\triangleright}\,\hat{I}\,\iota * \boxed{\mathrm{D}}_\iota\right) \vee \boxed{\mathrm{E}}_\iota\right) \qquad \boxed{\bullet\,\mathsf{ag}\,I}^{\gamma_{\textsc{Inv}}} * \mathop{\Large *}_{\iota \in \mathrm{dom}\,I} \left(\left([\![\,I\,\iota\,]\!] * \boxed{\mathrm{D}}_\iota\right) \vee \boxed{\mathrm{E}}_\iota\right)$$

Fig. 8. Invariant models: Later-weakened vs. Nola.

(list-tail). Please note that the counterpart of (2.1) for the later-weakened list $\mathsf{llist}\ \Phi\ \mathtt{r}$ cannot be verified, due to the later modality $\triangleright$ blocking the access to the tail list (llist-tail).

Remarkably, we can also verify various *concurrent* programs. For example, from (2.1) we can derive the following *total* Hoare triple on a program where a list is concurrently mutated:

$$\left[\,[\![\,\mathtt{list}\ \Phi\ \mathtt{r}\,]\!] * \mathtt{c} \mapsto m * \mathtt{c}' \mapsto n\,\right]\ \mathbf{fork}\ \{\ \mathtt{iterc(f,c,r)}\ \};\ \mathtt{iterc(f,c',r)}\ \left[\,\mathtt{c}' \mapsto 0\,\right]^{\mathsf{Winv}\,[\![\,]\!]}.$$

Using (2.1), we can also verify a larger program that spawns an unbounded number of threads that mutate an unbounded number of elements of a list, taking non-deterministic natural numbers (**ndnat**); see our Rocq mechanization for the details.

For a more advanced example, we can consider the following variant `iterc2` of the function `iterc`, using two counters under the *lexicographic order* for termination:

```
fn iterc2(f,c,c',r) { if *c > 0 || *c' > 0 { f(r);
    if *c > 0 { *c = *c − 1; *c' = ndnat } else { *c' = *c' − 1 };
    iterc2(f,c,c',*(r+1)) } }
```

Notably, when decrementing the first counter `c`, we set the second counter `c'` to a non-deterministic natural number **ndnat**, which can be arbitrarily large. Using Nola, we can also easily prove the following total Hoare assertion for this function, again for any Φ and `f`:

$$\frac{\forall \mathtt{r}.\ \left[\,\boxed{\Phi\ \mathtt{r}}\,\right]\ \mathtt{f(r)}\ \left[\,\top\,\right]^{\mathsf{Winv}\,[\![\,]\!]}}{\left[\,[\![\,\mathtt{list}\ \Phi\ \mathtt{r}\,]\!] * \mathtt{c} \mapsto m * \mathtt{c}' \mapsto n\,\right]\ \mathtt{iterc2(f,c,c',r)}\ \left[\,\mathtt{c} \mapsto 0 * \mathtt{c}' \mapsto 0\,\right]^{\mathsf{Winv}\,[\![\,]\!]}} \tag{2.2}$$

The proof of (2.2) goes in a natural way by *induction in the meta-logic*. We can take advantage of the fact that each update of the pair $(m, n) \in \mathbb{N} \times \mathbb{N}$ of the values for the two counters follows the lexicographic order, which is *well-founded*. Aside from the induction strategy, the whole proof is analogous to (2.1), especially in the use of the later-free access by (list-tail). We can also prove a variant of `iterc` and `iterc2` using an arbitrary step function `s` that decreases some state with respect to an arbitrary well-founded relation. Again, the proof of it goes naturally by meta-logic induction; please refer to our Rocq mechanization for the details.

3 Model of Nola's Invariant

Now we briefly explain the semantic model of Nola's invariants presented in § 2.

Model. Figure 8 presents the invariant models of the existing later-weakened approach and Nola in parallel. Please refer to [Jung et al. 2018b] etc. for technical backgrounds.

Fortunately, Nola's model is a straightforward generalization of Iris's model, turning $\triangleright\, \mathit{iProp}$ to Fml and $\check{\triangleright}\colon \triangleright\, \mathit{iProp} \to \mathit{iProp}$ to $[\![\,]\!]\colon \mathit{Fml} \to \mathit{iProp}$. Also, the model satisfies the dependencies described in Fig. 5. In particular, the invariant token $\boxed{P}$ does not depend on the semantics $[\![\,]\!]$. As previewed in § 1.2, the *contractiveness* of the set of SL formulas Fml over iProp is vital for the

domain equation on *iProp* to be solvable, technically because *iProp* depends on the global camera in a negative (not strictly positive) position.

Soundness. Now that we have the model, we can prove the soundness of Nola's invariants.

Theorem 3.1 (Soundness of Nola's Invariant). *The proof rules for Nola's invariants shown in Fig. 3 are sound with respect to the model of Fig. 8, for any choice of Fml and* $[\![\ ]\!] : \mathit{Fml} \to \mathit{iProp}$.

Proof. Straightforward, largely just like Iris's invariant. For the access rules inv-acc and thoare-inv, we use the fact that the semantics $[\![\ ]\!]$ is *non-expansive*, i.e., $P \mathrel{\dot{=}} Q$ implies $[\![P]\!] \mathrel{\dot{=}} [\![Q]\!]$, as the invariant token $\boxed{P}$ observes an indexed agreement. For winv-create, we set $I : \mathbb{N} \xrightarrow{\text{fin}} \mathit{Fml}$ in $\mathrm{Winv}\,[\![\ ]\!]$ to the empty map, which enables the universal quantification over $[\![\ ]\!]$. □

4 Expressivity

Nola gives higher-order ghost state parameterized over the set of formulas *Fml* and their semantics $[\![\ ]\!] : \mathit{Fml} \to \mathit{iProp}$, where the image $\{[\![P]\!] \mid P \in \mathit{Fml}\}$ determines the class of SL propositions that can be expressed. As we see in § 2, we can design *Fml* and $[\![\ ]\!]$ to support nested references, recursive definitions, and any SL propositions under the later modality. However, there are also fundamental limitations due to paradoxes like Paradox 1.3. What are the limits of expressivity?

Our general observation is as follows: expressivity can be extended as long as *Fml* and $[\![\ ]\!]$ are *well-defined*, which crucially depends on the absence of *circularity* in definition. Now we analyze this observation more closely.

Impossibility to express arbitrary SL propositions. First of all, it is impossible for Nola's *Fml* and $[\![\ ]\!]$ to express arbitrary SL propositions; there is no free lunch. An attempt to set $\mathit{Fml} \triangleq_? \mathit{iProp}$ and $[\![P]\!] \triangleq_? P$ fails, because this violates the contractiveness restriction of *Fml* over *iProp*, necessary for breaking bad circularity between *Fml* and *iProp* for the well-definedness of the two. Another attempt to set $\mathit{Fml} \triangleq \triangleright \mathit{iProp}$ and $[\![\mathsf{next}\ P]\!] \triangleq P$ also fails, because the semantics $[\![\ ]\!]$ does not satisfy the non-expansiveness restriction.

Inexpressibility of the total Hoare triple and view shift. Nola avoids the paradox of Landin's knot Paradox 1.3, because total Hoare triples cannot be embedded into $\mathit{Fml}/[\![\ ]\!]$ without the guard of the later modality, because that would cause bad *circularity* in the definition of the semantics $[\![\ ]\!] : \mathit{Fml} \to \mathit{iProp}$. More explicitly, if *Fml* had a later-free constructor $\mathtt{thoare}\ P\ \mathtt{e}\ \Psi$, the expected semantics $[\![\mathtt{thoare}\ P\ \mathtt{e}\ \Psi]\!] \triangleq_? \left[[\![P]\!]\right] \mathtt{e} \left[\lambda \mathtt{v}.\ [\![\Psi\ \mathtt{v}]\!]\right]^{W\,[\![\ ]\!]}$ involving the *world satisfaction* $W : (\mathit{Fml} \to \mathit{iProp}) \to \mathit{iProp}$ is *ill-defined*, due to the *circular* reference to $[\![\ ]\!]$ in $W\,[\![\ ]\!]$.

For a similar reason, we cannot add to *Fml* a later-free constructor for the view shift $P \Rrightarrow^{W\,[\![\ ]\!]} Q$. Technically, we also found a purely logical version of Paradox 1.3 for the view shift, which accounts for this limitation. This purely logical paradox is based on the paradox found earlier by Krebbers et al. [2017, § 5] but enjoys much simpler construction, not storing invariants and impredicative quantifiers inside invariants unlike theirs.

For the *partial* Hoare triple, it suffices to use the later-weakened version $\triangleright \{P\}\ \mathtt{e}\ \{\Psi\}^{\mathrm{Winv}\,[\![\ ]\!]}$, since the partial Hoare triple admits step-indexing.

Inexpressibility and expressibility of impredicative quantifiers. Nola's approach cannot generally express *unguarded impredicative quantifiers*. For example, if we had a formula $\exists X \in \mathit{Fml}.\ X * \boxed{X}$, its expected semantics $[\![\exists X \in \mathit{Fml}.\ X * \boxed{X}]\!] \triangleq_? \exists P \in \mathit{Fml}.\ [\![P]\!] * \boxed{P}$ is ill-defined due to the circular reference to the whole semantics $[\![\ ]\!]$. Indeed, unrestricted second-order quantifiers in *Fml* would cause Paradox 1.3, because the world satisfaction $\mathrm{Winv}\,[\![\ ]\!]$ is built with a second-order quantifier over *Fml* (see the model Fig. 8). Still, impredicative quantifiers whose variables occur

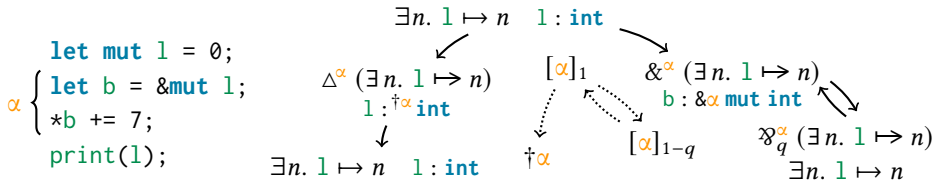

Fig. 9. Basic example of Rust's borrow and its model by Nola's borrows.

only in *guarded* positions can generally be expressed, because the guards avoid circularity in the semantics $[\![\ ]\!]$. For example, the formula $\exists\, X \in \mathit{Fml}.\, \boxed{X}$ can be safely added to Fml, as its semantics $[\![\exists\, X \in \mathit{Fml}.\, \boxed{X}]\!] \triangleq \exists P \in \mathit{Fml}.\, \boxed{P}$ is well-defined.

Expressibility via stratification. As a further extension, by *stratifying* SL formulas [Ahmed et al. 2002], Nola can support rich connectives such as the total Hoare triple and (unguarded) higher-order quantifiers. For example, we can build two sets of SL formulas, Fml_0 and Fml_1, where Fml_1 contains the total Hoare triple $\mathtt{thoare}^0\ P_1\ \mathtt{e}\ \Psi_1$ that supports invariants on Fml_0 (but not Fml_1), supporting invariant tokens $\boxed{P_i}$ on $P_i \in \mathit{Fml}_i$ for both $i = 0, 1$. This is well-defined, because we can first build the level-0 semantics $[\![\ ]\!]_0 \colon \mathit{Fml}_0 \to \mathit{iProp}$ and then build the level-1 semantics $[\![\ ]\!]_1 \colon \mathit{Fml}_1 \to \mathit{iProp}$, where the total Hoare triple semantics $[\![\mathtt{thoare}^0\ P\ \mathtt{e}\ \Psi]\!]_1 \triangleq \left[[\![P]\!]_1\right] \mathtt{e} \left[[\![\Psi]\!]_1\right]^{\mathrm{Winv}[\![\ ]\!]_0}$ has no circularity. Note that this avoids the paradox of Landin's knot Paradox 1.3.

5 Nola's Later-Free Borrow

Now we present Nola's *later-free borrow*, a later-free version of RustBelt's borrow [Jung et al. 2018a; Jung 2020]. Borrow is an advanced type of higher-order ghost state that serves as the semantic foundation for the ownership type system of Rust [Matsakis and Klock 2014].

5.1 Overview

What are borrows? Rust's borrowing machinery [Matsakis and Klock 2014] is a successful feature for managing ownership. It originates from Cyclone [Grossman et al. 2002] and earlier studies on region types [Tofte and Talpin 1994, 1997; Gay and Aiken 1998].

Figure 9 shows a basic example of a borrow in Rust. First, `l : int` gets the exclusive ownership of a new memory cell initialized with `0` (for simplicity, we consider an unbounded integer type `int`). Then the ownership of `l` is temporarily *borrowed* by a newly created *mutable reference* `b : &α mut int`. The period of the borrow is denoted by the *lifetime* α. During the lifetime α, the borrower `b` has the exclusive ownership of the memory cell, being able to update its content (`*b += 7`), while the lender `l` gets *frozen* (`l :†α int`), with its access all suspended. After the lifetime α ends, the lender `l` retrieves the ownership, being able to access the content again (`print(l)`).

Remarkably, there is *no direct communication* needed for the borrower `b : &α mut T` to return the ownership to the lender `l :†α T`. The mutable reference `&α mut T` can simply be thrown away at any time. For this advanced mechanism, in a sense, the borrowed object is *shared* between the borrower and the lender, with its access controlled by the *lifetime* α.

Modeling borrows. To model Rust's borrows, RustBelt developed the *lifetime logic* [Jung et al. 2018a; Jung 2020], an Iris library providing higher-order ghost state for Rust-style borrows. It features the *borrower token* $\&^\alpha \triangleright P \in \mathit{iProp}$ (called full borrow and written as $\&^\alpha_{\mathrm{full}}\, P$ by RustBelt), which asserts that an object satisfying $P \in \mathit{iProp}$ is borrowed under the lifetime α, which can be used to model the mutable reference type `&α mut T`. RustBelt gave a semantic proof of the memory

$$\top \Rrightarrow \exists \alpha.\, [\alpha]_1 \quad \textsc{lft-alloc} \qquad [\alpha]_1 \Rrightarrow \dagger\alpha \quad \textsc{lft-end}$$

$$[\![P]\!] \Rrightarrow^{\mathrm{Wbor}\,[\![\,]\!]} \&^\alpha P * \triangle^\alpha P \quad \textsc{bor-lend-new} \qquad \dagger\alpha * \triangle^\alpha P \Rrightarrow^{\mathrm{Wbor}\,[\![\,]\!]} [\![P]\!] \quad \textsc{lend-back}$$

$$\&^\alpha P * [\alpha]_q \Rrightarrow^{\mathrm{Wbor}\,[\![\,]\!]} \aleph^\alpha_q P * [\![P]\!] \quad \textsc{bor-open} \qquad \aleph^\alpha_q P * [\![P]\!] \Rrightarrow^{\mathrm{Wbor}\,[\![\,]\!]} \&^\alpha P * [\alpha]_q \quad \textsc{obor-close}$$

$$\top \Rrightarrow \exists \gamma_{\mathrm{Bor}}.\, \forall [\![\,]\!].\, \mathrm{Wbor}\,[\![\,]\!] \quad \textsc{wbor-create}$$

Fig. 10. Selected proof rules for Nola's borrows.

and thread safety guarantee of Rust's ownership type system. It was later extended to verify various interesting properties about Rust's ownership type system [Dang et al. 2020; Yanovski et al. 2021; Matsushita et al. 2022; Gäher et al. 2024]. Still, unfortunately, for the same reason as invariants $\boxed{\triangleright P}$, RustBelt's lifetime logic weakened the shared content by the *later modality* $\triangleright$ and suffered from later-weakened proof rules, and its application has been limited to safety verification.

Nola solves this situation by building later-free borrows. Actually, the approach is quite similar to later-free invariants presented in § 2. Now we explain this machinery more in detail.

5.2 Nola's Later-Free Borrows

Nola-specific aspects. Nola-specific aspects of Nola's borrows are quite the same as Nola's invariants explained in § 2. Nola parameterizes RustBelt's borrow machinery over the set of SL formulas *Fml* and the semantics $[\![\,]\!] : \mathit{Fml} \to \mathit{iProp}$ to achieve later-free proof rules. In particular, we provide a new borrower token $\&^\alpha P \in \mathit{iProp}$ that takes an SL formula $P \in \mathit{Fml}$, borrowing its semantics $[\![P]\!]$ under α. Nola's borrows restrict *Fml* to be contractive over *iProp* and $[\![\,]\!]$ to be non-expansive. The dependencies like Fig. 5 hold also for Nola's borrows.

How does the logic work? Figure 9 illustrates the flow of the ownership for the basic example in Nola's borrow machinery. Here, we assume a formula data type *Fml* like Fig. 6. By borrowing a mutable integer object `l : int`, modeled as the points-to token $\exists n.\ \mathtt{l} \mapsto n$, we get a *borrower token* $\&^\alpha (\exists n.\ \mathtt{l} \mapsto n)$ for the borrower `b : &α mut int` and the *lender token* $\triangle^\alpha (\exists n.\ \mathtt{l} \mapsto n)$ for the frozen original owner `l : †α int`. The borrower can get access to the content $\exists n.\ \mathtt{l} \mapsto n$ by depositing the *live lifetime token* $[\alpha]_q$ as a fractional witness that the lifetime α is still ongoing. While using the content, the borrower temporarily turns into the *open borrower token* $\aleph^\alpha_q (\exists n.\ \mathtt{l} \mapsto n)$. By storing back the content $\exists n.\ \mathtt{l} \mapsto n$, the borrower turns back into $\&^\alpha (\exists n.\ \mathtt{l} \mapsto n)$ and retrieves $[\alpha]_q$. When ending the lifetime α, the full token $[\alpha]_1$ is consumed into the *dead lifetime token* $\dagger\alpha$, which is a persistent witness that α has ended. After $\dagger\alpha$ is obtained, the lender token $\triangle^\alpha (\exists n.\ \mathtt{l} \mapsto n)$ can be turned into $\exists n.\ \mathtt{l} \mapsto n$, modeling the unfrozen object `l : int`.

Figure 10 shows selected proof rules for Nola's borrows. We use LFT-ALLOC to allocate a fresh lifetime and use LFT-END to end it. Also, $[\alpha]_{q+r} \Leftrightarrow [\alpha]_q * [\alpha]_r$ holds. The rule BOR-LEND-NEW creates a new borrow over a content satisfying $P \in \mathit{Fml}$ under the lifetime α, yielding a *borrower token* $\&^\alpha P$ and a *lender token* $\triangle^\alpha P$. The borrower token accesses the content $[\![P]\!]$ with the witness $[\alpha]_q$ by the rules BOR-OPEN and OBOR-CLOSE. By the rule OBOR-CLOSE, the lender token retrieves the content $[\![P]\!]$ by the witness $\dagger\alpha$. For accessing borrows, we use the *world satisfaction* $\mathrm{Wbor}\,[\![\,]\!]$ for Nola's borrows, which can be created by WBOR-CREATE, just as in Nola's invariant machinery.

For lifetimes, we also have the *lifetime inclusion* $\alpha \sqsubseteq \beta \in \mathit{iProp}$, a persistent assertion that a lifetime α is outlived by β. The lifetime of the borrower and lender tokens can be modified by lifetime inclusion ($\alpha \sqsubseteq \beta$ implies $\&^\beta P \Rightarrow \&^\alpha P$ and $\triangle^\alpha P \Rightarrow \triangle^\beta P$). Nola's borrow machinery

also provides advanced rules for *subdividing* a borrower into borrowers on smaller parts (e.g., turning $\&\alpha$ **mut Vec**<T> to $\&\alpha$ **mut** T) and *reborrowing* a borrower (e.g., reborrowing b : $\&\alpha$ **mut** T to get c : $\&\beta$ **mut** T and b : $\dagger\beta\,\&\alpha$ **mut** T), like RustBelt's lifetime logic. Please refer to our Rocq mechanization for more details.

Verification examples. Now we present some examples of verification using Nola's borrows.

Example 5.1 (Dereference of a nested borrow). For an advanced example, we can reason about *nested borrows* using Nola's later-free borrows. We can model a nested mutable reference as the following SL proposition, where the ownership type T is modeled as a parameterized SL formula $\mathit{Addr} \to \mathit{Fml}$ (we extend Fig. 6 with a borrower constructor $\&^\alpha P$ interpreted as $[\![\&^\alpha P]\!] \triangleq \&^\alpha P$):

$$\mathtt{r} : \&\beta\ \mathbf{mut}\ \&\alpha\ \mathbf{mut}\ \mathtt{T} \quad \triangleq \quad \mathrm{bb}_{\mathtt{r}}^{\beta,\alpha}\,\mathtt{T} \quad \triangleq \quad \&^\beta\left(\exists\, \mathtt{s}.\ \mathtt{r} \mapsto \mathtt{s} * \&^\alpha(\mathtt{T}\,\mathtt{s})\right).$$

Remarkably, we can verify an expected *total* Hoare triple for the *dereference* of a nested mutable reference $*\mathtt{r} : \&\beta$ **mut** T:

$$\left[\ \beta \sqsubseteq \alpha * [\beta]_q * \mathrm{bb}_{\mathtt{r}}^{\beta,\alpha}\,\mathtt{T}\ \right]\ *\mathtt{r}\ \left[\ \lambda \mathtt{s}.\ [\beta]_q * \&^\beta(\mathtt{T}\,\mathtt{s})\ \right]^{\mathrm{Wbor}\,[\![\,]\!]}.$$

Notably, we can get access to the inner reference $\&^\beta(\mathtt{T}\,\mathtt{s})$ *not* weakened by the later modality $\triangleright$, thanks to Nola's later-free borrows (recall (llist-tail) vs. (list-tail) in § 2.3). Technically, to prove this, we use advanced rules for borrow subdivision and reborrowing, omitted from Fig. 10.

Rust also supports shared borrows $\&\alpha$ T, which enables *sharing* the borrowed objects. We can model them by combining Nola's borrows with Nola's *invariants* $\boxed{P}$ explained in § 2.1.

Example 5.2 (Shared borrows over integers). For a simple example, an immutable shared borrow over an integer r: $\&\alpha$ **int** can be modeled as $\boxed{\&^\alpha\, \mathtt{r} \mapsto n}$,[8] an invariant storing a borrower. We can read from r under this assertion (recall that world satisfactions can be combined by THOARE-EXPAND):

$$\left[\ \boxed{\&^\alpha\, \mathtt{r} \mapsto n} * [\alpha]_q\ \right]\ *\mathtt{r}\ \left[\ \lambda \mathtt{v}.\ n = \mathtt{v} * [\alpha]_q\ \right]^{\mathrm{Winv}\,[\![\,]\!] * \mathrm{Wbor}\,[\![\,]\!]}.$$

Remarkably, we can also model *mutable* shared borrows (or *interior mutability*) in Rust.

Example 5.3 (Shared borrows over mutexes). For example, a shared borrow over a *mutex* can be modeled as follows, writing $\mathtt{T}_\mathtt{r}$ for $\mathtt{T}\,(\mathtt{r}+1)$:

$$\mathtt{r} : \&\alpha\ \mathtt{Mutex}\langle\mathtt{T}\rangle \quad \triangleq \quad \mathrm{mb}_{\mathtt{r}}^{\alpha}\,\mathtt{T} \quad \triangleq \quad \boxed{\&^\alpha\left((\mathtt{r} \mapsto \mathrm{false} * \&^\alpha\,\mathtt{T}_\mathtt{r}) \vee \mathtt{r} \mapsto \mathrm{true}\right)}.$$

This model is a bit tricky, but its basic idea is to use an invariant for the disjunction over the two states (unlocked and locked) of the mutex, with the twist of lifetime-based control by borrower tokens. The reference r can get access to the content by acquiring the lock:

$$\left[\ \mathrm{mb}_{\mathtt{r}}^{\alpha}\,\mathtt{T} * [\alpha]_q\ \right]\ \mathbf{cas}\ \mathrm{true}\ \mathrm{false}\ \left[\ \lambda \mathtt{v}.\ \left((\mathtt{v} = \mathrm{true} * \&^\alpha\,\mathtt{T}_\mathtt{r}) \vee \mathtt{v} = \mathrm{false}\right) * [\alpha]_q\ \right]^{\mathrm{Winv}\,[\![\,]\!] * \mathrm{Wbor}\,[\![\,]\!]}.$$

We can newly create a shared borrow of Mutex<T> along with a frozen lender as follows:

$$\mathtt{r} \mapsto b * [\![\mathtt{T}_\mathtt{r}]\!] * [\alpha]_q \Rrightarrow^{\mathrm{Winv}\,[\![\,]\!] * \mathrm{Wbor}\,[\![\,]\!]} \mathrm{mb}_{\mathtt{r}}^{\alpha}\,\mathtt{T} * \triangle^\alpha\left(\exists\, b'.\ \mathtt{r} \mapsto b' * \mathtt{T}_\mathtt{r}\right) * [\alpha]_q.$$

6 Semantic Alteration by Magic Derivability

We discuss a crucial challenge about higher-order ghost state, *semantic alteration*, and present our general solution to it, nicknamed *magic derivability*.

[8] Precisely speaking, to support lifetime weakening, we can modify the invariant with $\exists\alpha' \sqsupseteq \alpha$ and replace α with α'. Also, technically, to allow non-atomic accesses, we can use the fractional points-to token, like $\boxed{\exists\, q.\ \&^\alpha\, \mathtt{r} \overset{q}{\mapsto} n}$.

$$\mathsf{der}\ J \ \Rightarrow\ [\![J]\!]^{+}_{\mathsf{der}} \quad \textsc{der-sound} \qquad\qquad \mathsf{der} \in \mathit{Deriv} \quad \textsc{der-deriv}$$

$$\big(\forall\, \delta \in \mathit{Deriv}.\ \ast^{n}_{i=0} [\![J'_i]\!]^{+}_{\delta} \mathrel{-\!\!\ast} [\![J]\!]^{+}_{\delta}\big) \ \Rightarrow\ \forall\, \delta \in \mathit{Deriv}.\ \ast^{n}_{i=0} \delta\, J'_i \mathrel{-\!\!\ast} \delta\, J \quad \textsc{deriv-map}$$

Fig. 11. Selected proof rules on der and *Deriv*.

Challenge: Semantic alteration. Nola's clients can use *syntactic* SL formulas $P \in \mathit{Fml}$ for higher-order ghost state such as invariants $\boxed{P}$, which clears the later modality $\triangleright$ and helps verification especially of termination. For advanced usages, however, we often further want the power to *semantically alter* such SL formulas. For example, suppose that we want to give a *semantic* soundness proof of a type system with the shared mutable reference type $\texttt{ref}\ \mathsf{T}$, which is modeled using an invariant $\boxed{P}$. A key challenge is in verifying the *subtyping* rule over references, which derives $\texttt{ref}\ \mathsf{T} \le \texttt{ref}\ \mathsf{U}$ from $\mathsf{T} \le \mathsf{U}$ and $\mathsf{U} \le \mathsf{T}$.

To achieve such high-level goals, higher-order ghost state like invariants should flexibly allow various forms of *semantic alteration* of the SL formulas, such as the following:

$$[\![\boxed{P \ast Q}]\!] \Leftrightarrow [\![\boxed{Q \ast P}]\!] \qquad \Big[\!\Big[\boxed{\boxed{P \ast Q}}\Big]\!\Big] \Leftrightarrow \Big[\!\Big[\boxed{\boxed{Q \ast P}}\Big]\!\Big] \qquad \frac{\alpha \sqsubseteq \beta \qquad \beta \sqsubseteq \alpha}{[\![\boxed{\&^{\alpha} P}]\!] \Leftrightarrow [\![\boxed{\&^{\beta} P}]\!]} \tag{6.1}$$

In the first case, for example, the formulas $P \ast Q$ and $Q \ast P$ are syntactically different but semantically equivalent, so the equivalence should hold. We also want to perform such semantic alteration over nested invariants, as in the second case. Moreover, the notion of semantic equivalence over SL formulas should be able to depend on persistent SL hypothesis, as in the third case.

Unfortunately, the direct model considered so far, $[\![\boxed{P}]\!] \triangleq \boxed{P}$, does not support semantic alteration like (6.1). This is because the token $\boxed{P}$ is modeled based on the plain agreement on *Fml*, not considering its semantics (recall § 3). To resolve this, we need to relax the agreement on *Fml* into a *semantic* one. The naive starting point is as follows:

$$[\![\boxed{P}]\!] \quad \triangleq_{?} \quad \exists\, Q \ \text{s.t.}\ [\![P]\!] \Leftrightarrow [\![Q]\!].\ \boxed{Q} \tag{6.2}$$

This uses a semantic agreement $[\![P]\!] \Leftrightarrow [\![Q]\!]$ described by the semantics $[\![\ ]\!]$. Unfortunately, this naive semantics (6.2) is ill-defined, because it has a *circular* self-reference $[\![Q]\!]$. So we need a good substitute, or under-approximation, for the real semantic agreement $[\![P]\!] \Leftrightarrow [\![Q]\!]$.

A possible approach could be to build a *syntactic* proof system for the semantic agreement. However, this really hurts the modularity and extensibility, because the soundness of the system should be proved again every time we add new proof rules or constructors. Can we do better?

Our solution: Magic derivability. To solve this challenge, we found a general mechanism, nicknamed *magic derivability*. Intuitively, it gives an approximate solution to naive recursive equations like (6.2), based on a fixed point construction.

To use magic derivability, we first design the *judgment* $J \in \mathit{Judg}$, the input of the derivability. For example: $\mathit{Judg} \ni J ::= P \Leftrightarrow Q$. Then we *parameterize* the formula semantics $[\![\ ]\!]_{\delta} : \mathit{Fml} \to \mathit{iProp}$ over *derivability candidates* $\delta : \mathit{Judg} \to \mathit{iProp}$. For the running example, it is defined as follows, modifying the semantics $[\![\ ]\!]$ of Fig. 6 (excerpts):

$$[\![\boxed{P}]\!]_{\delta} \quad \triangleq \quad \exists\, Q \ \text{s.t.}\ \delta\,(P \Leftrightarrow Q).\ \boxed{Q} \tag{6.3}$$

$$[\![P \ast Q]\!]_{\delta} \quad \triangleq \quad [\![P]\!]_{\delta} \ast [\![Q]\!]_{\delta} \qquad\qquad [\![\mathsf{r} \mapsto \mathsf{v}]\!]_{\delta} \quad \triangleq \quad \mathsf{r} \mapsto \mathsf{v}$$

Finally, we can define a *parameterized judgment semantics* $[\![\ ]\!]^{+} : (\mathit{Judg} \to \mathit{iProp}) \to (\mathit{Judg} \to \mathit{iProp})$. For the running example, it is defined as follows: $[\![P \Leftrightarrow Q]\!]^{+}_{\delta} \triangleq [\![P]\!]_{\delta} \Leftrightarrow [\![Q]\!]_{\delta}$.

Now our magic derivability machinery gives the *derivability* $\mathrm{der} : \mathit{Judg} \to \mathit{iProp}$ with the set of derivability candidates $\mathit{Deriv} \subseteq (\mathit{Judg} \to \mathit{iProp})$. The derivability der is roughly an approximate fixed point of $[\![\,]\!]^+ : (\mathit{Judg} \to \mathit{iProp}) \to (\mathit{Judg} \to \mathit{iProp})$ (see Model below for the details). The rules presented in Fig. 11 always holds. Remarkably, the derivability der is always *sound* DER-SOUND. We also have the rule DERIV-MAP for deducing a derivability candidate $\delta\, J$ by *semantics* $[\![\,]\!]^+$.

For the invariant $\boxed{P}$ interpreted as (6.3), the access rules analogous to the original invariant (e.g., THOARE-INV) is obtained by the soundness DER-SOUND, and the allocation rule like the original invariant (INV-ALLOC) is obtained by the rule DERIV-MAP with $n = 0$ (we define the standard semantics by the derivability, i.e., $[\![\,]\!] \triangleq [\![\,]\!]_{\mathrm{der}}$):

$$\frac{\left[\,[\![P]\!] * Q\,\right]\ \mathsf{ae}\ \left[\,\lambda \mathsf{v}.\ [\![P]\!] * \Psi\,\mathsf{v}\,\right]^{\mathrm{Winv}[\![\,]\!]}}{\left[\,[\![\boxed{P}]\!] * Q\,\right]\ \mathsf{ae}\ \left[\,\Psi\,\right]^{\mathrm{Winv}[\![\,]\!]}} \qquad [\![P]\!] \Rrightarrow^{\mathrm{Winv}[\![\,]\!]} [\![\boxed{P}]\!]$$

Also, thanks to DERIV-MAP, we have the following semantic alteration rule for the invariant:

$$\left(\,\forall\, \delta \in \mathit{Deriv}.\ [\![P]\!]_\delta \Leftrightarrow [\![Q]\!]_\delta\,\right) \Rightarrow \forall\, \delta \in \mathit{Deriv}.\ [\![\boxed{P}]\!]_\delta \Leftrightarrow [\![\boxed{Q}]\!]_\delta \tag{6.4}$$

Using (6.4), we can easily derive the rules of (6.1), even generally for $[\![\,]\!]_\delta$ of any $\delta \in \mathit{Deriv}$.

With magic derivability, we can now semantically and flexibly model and verify *subtyping* rules. For example, we can define the semantics of subtyping $\mathsf{T} \le \mathsf{U}$ as follows, an implication universally quantified over the choice of $\delta \in \mathit{Deriv}$: $[\![\mathsf{T} \le \mathsf{U}]\!] \triangleq \forall \delta \in \mathit{Deriv}, v.\ [\![\mathsf{T}\,v]\!]_\delta \Rightarrow [\![\mathsf{U}\,v]\!]_\delta$. Now we can smoothly verify the subtyping rules over references, such as $[\![\mathsf{T} \le \mathsf{U}]\!] \wedge [\![\mathsf{U} \le \mathsf{T}]\!] \Rightarrow [\![\mathbf{ref}\ \mathsf{T} \le \mathbf{ref}\ \mathsf{U}]\!]$. This technique is used in our primary case study, RustHalt (§ 7).

Model. Technically, *Deriv* is defined as the smallest set closed under the application of $[\![\,]\!]^+$ ($\delta \in \mathit{Deriv}$ implies $[\![\,]\!]^+_\delta \in \mathit{Deriv}$) and under the conjunction over any subset ($S \subseteq \mathit{Deriv}$ implies $(\lambda J. \forall \delta \in S.\, \delta\, J) \in \mathit{Deriv}$). Then, der is defined as the conjunction over *Deriv* ($\mathrm{der} \triangleq \lambda J. \forall \delta \in \mathit{Deriv}.\, \delta\, J$). This model is close to the standard iterative computation of greatest fixed points. Indeed, der becomes the greatest fixed point of $[\![\,]\!]^+$ if $[\![\,]\!]^+$ happens to be monotone.

7 RustHalt: A Semantic Foundation for Termination-Sensitive Rust Verification

To demonstrate the power of Nola, as our primary case study, we developed *RustHalt*, the first semantic and mechanized foundation for total correctness verification of Rust programs.

High-level overview. There is a rich body of work on functional verification of Rust programs [Ullrich 2016; Astrauskas et al. 2019; Matsushita et al. 2020; Ho and Protzenko 2022; Matsushita et al. 2022; Denis et al. 2022; Lattuada et al. 2023; Gäher et al. 2024]. Of particular note is RustHorn [Matsushita et al. 2020, 2021], which found a general way to encode mutable references as first-class values, which uses *prophecies* [Abadi and Lamport 1988; Vafeiadis 2008; Jung et al. 2020b], imaginary values for future information. Its key idea can be summarized as follows: a prophecy can let the borrower virtually communicate to the lender the final result of the mutation. This simple idea has proved widely useful in existing work [Matsushita et al. 2020; Denis et al. 2022; Skotåm 2022; Fiala et al. 2023; Gäher et al. 2024].

However, proving the soundness of this approach is challenging, because it involves a subtle interaction between borrows and prophecies. RustHornBelt [Matsushita et al. 2022] extended RustBelt [Jung et al. 2018a] to prove the soundness of RustHorn-style functional verification. Its idea has been recently inherited by RefinedRust [Gäher et al. 2024], focusing more on automation. However, their approach could not support *termination* verification, again due to the *later modality*. This is a real problem, since total correctness verification is an actively used feature in real-world Rust verifiers [Ho and Protzenko 2022; Denis et al. 2022; Lattuada et al. 2023].

$$\mathtt{a} : \mathbf{int},\ \mathtt{b} : \mathbf{int} \vdash \mathtt{a} + \mathtt{b} \dashv \mathtt{r}.\ \mathtt{r} : \mathbf{int} \rightsquigarrow \lambda post, [a, b].\ post\,[a + b] \quad \textsc{int-add}$$

$$\mathtt{a} : \mathbf{Box}\langle \mathtt{T} \rangle \vdash \mathbf{\&mut}\ *\mathtt{a} \dashv \mathtt{r}.\ \mathtt{r} : \&_\alpha\ \mathbf{mut}\ \mathtt{T},\ \mathtt{a} :^{\dagger_\alpha} \mathbf{Box}\langle \mathtt{T} \rangle \rightsquigarrow \lambda post, [a].\ \forall a'.\ post\,[(a, a'), a'] \quad \textsc{box-mut-borrow}$$

$$\mathtt{a} : \&_\alpha\ \mathbf{mut}\ \mathtt{T},\ \mathtt{b} : \mathtt{T} \vdash *\mathtt{a} = \mathtt{b} \dashv \mathtt{a} : \&_\alpha\ \mathbf{mut}\ \mathtt{T} \rightsquigarrow \lambda post, [(a, a'), b].\ post\,[(b, a')] \quad \textsc{mutref-write}$$

$$\mathtt{a} : \&_\alpha\ \mathbf{mut}\ \mathtt{T} \vdash \&*\mathtt{a} \dashv \mathtt{r}.\ \mathtt{r} : \&_\alpha\ \mathtt{T} \rightsquigarrow \lambda post, [(a, a')].\ a' = a \to post\,[a] \quad \textsc{mutref-share}$$

$$\frac{\Gamma \vdash \mathtt{e}\left[\overline{\mathtt{a}/\mathtt{x}},\ \mathbf{fn}\ \mathtt{f}(\bar{\mathtt{x}})\{\ \mathtt{e}\ \}/\mathtt{f}\right] \dashv \mathtt{r}.\ \Gamma' \rightsquigarrow pre}{\Gamma, \Gamma_+ \vdash (\mathbf{fn}\ \mathtt{f}(\bar{\mathtt{x}})\{\ \mathtt{e}\ \})(\bar{\mathtt{a}}) \vdash \mathtt{r}.\ \Gamma', \Gamma_+ \rightsquigarrow \lambda post, [\bar{a}, \bar{c}].\ pre\,(\lambda \bar{b}.\ post\,[\bar{b}, \bar{c}])\,[\bar{a}]} \quad \textsc{fn-rec-call}$$

$$\frac{\mathtt{T} \succ get \qquad \forall c.\ \left(\mathtt{a} : \mathtt{T}, \Gamma \vdash \mathtt{e} \vdash \mathtt{r}.\ \Gamma' \rightsquigarrow \lambda post, [a, \bar{b}].\ get\,a = c \wedge pre\,post\,[a, \bar{b}]\right)}{\mathtt{a} : \mathtt{T}, \Gamma \vdash \mathtt{e} \vdash \mathtt{r}.\ \Gamma' \rightsquigarrow pre} \quad \textsc{real}$$

Fig. 12. Selected type-spec rules in RustHalt. Lifetime constraints implicit in Rust are omitted.

Our RustHalt solves this situation by reformulating RustHornBelt [Matsushita et al. 2022] using our framework Nola for later-free higher-order ghost state.

Type-spec system. To model functional verification based on Rust's ownership types, RustHalt builds a *type-spec system*, which extends Rust's typing judgments with functional specifications, inheriting RustHornBelt's approach. Here we give a brief explain of this system.

Like in RustHornBelt, RustHalt's type-spec judgment has the following form: $\Gamma \vdash_\gamma \mathtt{e} \dashv \mathtt{r}.\ \Gamma' \rightsquigarrow pre$. What comes before $\rightsquigarrow$ is a usual ownership typing judgment, consisting of the input type context Γ, the expression $\mathtt{e}$, and the output type context Γ' which can refer to the return variable $\mathtt{r}$. Each element of a type context is of form $\mathtt{a} : \mathtt{T}$ or $\mathtt{a} :^{\dagger_\alpha} \mathtt{T}$ (as in § 5), where the first form can be read as usual and the second form means that $\mathtt{a}$ is currently lending its object $\mathtt{T}$ under the lifetime α. The subscript γ of a type judgment is the lifetime for the current execution (typically the conjunction of all available lifetimes), which we usually omit for presentation. What comes after $\rightsquigarrow$ is the *functional specification*, *pre*: $(\lfloor \Gamma' \rfloor \to Prop) \to (\lfloor \Gamma \rfloor \to Prop)$. Technically, it is the predicate transformer (as in Dijkstra's weakest precondition calculus [Dijkstra 1976]), calculating the precondition $pre\,post : \lfloor \Gamma \rfloor \to Prop$ for each postcondition $post : \lfloor \Gamma' \rfloor \to Prop$; more intuitively, it is a kind of functional program in the continuation passing style with the continuation *post*. The representation sort $\lfloor \Gamma \rfloor$ for a type context is the product (or heterogeneous list) of the representation sort $\lfloor \mathtt{T} \rfloor$ for the type of each element $\mathtt{a} :^? \mathtt{T}$ in the type context. The representation sort for core types is as follows, for example: $\lfloor \mathbf{int} \rfloor \triangleq \mathbb{Z}$, $\lfloor (\mathtt{T}, \mathtt{U}) \rfloor \triangleq \lfloor \mathtt{T} \rfloor \times \lfloor \mathtt{U} \rfloor$, $\lfloor \mathbf{Box}\langle \mathtt{T} \rangle \rfloor \triangleq \lfloor \mathtt{T} \rfloor$, $\lfloor \&_\alpha\ \mathtt{T} \rfloor \triangleq \lfloor \mathtt{T} \rfloor$, $\lfloor \&_\alpha\ \mathbf{mut}\ \mathtt{T} \rfloor \triangleq \lfloor \mathtt{T} \rfloor \times \lfloor \mathtt{T} \rfloor$. Most interestingly, each mutable reference $\&_\alpha\ \mathbf{mut}\ \mathtt{T}$ is functionally represented as the pair $(a, a') \in \lfloor \mathtt{T} \rfloor \times \lfloor \mathtt{T} \rfloor$ of the current value $a \in \lfloor \mathtt{T} \rfloor$ and the *prophecy* $a' \in \lfloor \mathtt{T} \rfloor$, which fetches the final value of the borrowed object ahead of the time.

Figure 12 lists selected type-spec rules (typing rules with specifications) in RustHalt. As a warm-up, integer addition (INT-ADD) is functionally modeled by inputting the values $a, b \in \mathbb{Z}$ and passing the output $a + b \in \mathbb{Z}$ to the continuation *post*. Things become really interesting with mutable borrows. On mutable borrowing (BOX-MUT-BORROW), a fresh *prophecy* a' is taken, which is shared between the borrower (mutable reference) modeled as (a, a') and the lender modeled as a'. When we write to the mutable reference (MUTREF-WRITE), where the prophecy a' is retained. When the mutable reference finally loses its write access (for example, MUTREF-SHARE), it *resolves* the

prophecy a' to the real value a at that time, observing $a' = a$. We also have advanced rules about mutable references involving borrow subdivision and reborrowing.

What is really new about RustHalt is that it allows verifying *total correctness*. For that, RustHalt introduces a new rule FN-REC-CALL for the recursive function call, which just unfolds the function definition framing some part. We can prove total correctness through arbitrary induction in the *meta-logic* (or more specifically, Rocq). Also, for that, RustHalt introduces a new auxiliary rule REAL. The judgment $\mathtt{T} \succ \mathit{get}$ says that we can take out the real, non-prophetic part from the representation of $\mathtt{T}$ by the function $\mathit{get} \colon \lfloor \mathtt{T} \rfloor \to C$ (for some sort C). For example, $\&\alpha\ \mathtt{mut\ int} \succ \lambda(n, n').\, n$ holds. The rule REAL enables meta-level case analysis over the real part c of an object $\mathtt{a} : \mathtt{T}$.

Verification examples. For example, let us consider the following function for iterating over a singly linked list (a variant of `iterc` in § 2.3):

```
fn iter(f,l) { match l { Nil ⇒ (), Cons(a,l') ⇒ { f(a); iter(f,*l') } } }
```

Here, we define the list type `List<T>` as the following recursive data type:

```
enum List<T> { Nil, Cons(T, Box<List<T>>) }
```

We functionally represent `List<T>` using mathematical lists: $\lfloor \mathtt{List<T>} \rfloor \triangleq \mathit{List} \lfloor \mathtt{T} \rfloor$.

Remarkably, in RustHalt, we can verify the following total correctness assertion, notably for any Rust ownership type `T`, any Rust function `f`, and any pure function $f \colon \lfloor \mathtt{T} \rfloor \to \lfloor \mathtt{T} \rfloor$:

$$\frac{\forall \mathtt{a}.\ \ \mathtt{a} : \&\alpha\ \mathtt{mut\ T} \vdash \mathtt{f(a)} \dashv \_.\ \rightsquigarrow\ \lambda \mathit{post}, [(a, a')].\ a' = f\ a \to \mathit{post}\ []}{\mathtt{l} : \&\alpha\ \mathtt{mut\ List<T>} \vdash \mathtt{iter(f,l)} \dashv \_.\ \rightsquigarrow\ \lambda \mathit{post}, [(l, l')].\ l' = \mathsf{map}\ f\ l \to \mathit{post}\ []} \tag{7.1}$$

The premise of (7.1) says that the function `f` always *terminates* when called with a mutable reference $\mathtt{a} : \&\alpha\ \mathtt{mut\ T}$, performing the update specified by the function $f \colon \lfloor \mathtt{T} \rfloor \to \lfloor \mathtt{T} \rfloor$ (setting the prophecy a' to $f\ a$). The assertion (7.1) says that, under this premise, the function call `iter(f,l)` always *terminates* under $\mathtt{l} : \&\alpha\ \mathtt{mut\ List<T>}$, performing the update specified by the function $\mathsf{map}\ f \colon \mathit{List} \lfloor \mathtt{T} \rfloor \to \mathit{List} \lfloor \mathtt{T} \rfloor$. We can prove this total correctness in RustHalt as follows: first apply REAL to get the length $n \in \mathbb{N}$ of the input list l (as $\&\alpha\ \mathtt{mut\ List<T>} \succ \lambda(l, l').\, \mathsf{length}\ l$ holds for any `T`), and then simply do mathematical induction over n in the *meta-logic*.

We have also verified a richer variant of this example that uses a modified list type that uses the mutable reference type $\&\alpha\ \mathtt{mut\ U}$ instead of the owned pointer type `Box<T>` for self-reference. We have also verified functions implementing the Ackermann function (one in a usual style and one using a mutable reference for the output), using meta-level induction on the lexicographic order over pairs of natural numbers. Please see our Rocq mechanization for the details.

Semantic model by Nola. RustHalt proved the soundness of this type-spec system by modeling Rust's ownership types as (parameterized) *syntactic SL formulas*, instead of semantic SL propositions as in the existing work. More specifically, RustHalt sets the domain RustTy_A for Rust's ownership types (where A is the representation sort) as the following record:

$$\left\{ \begin{array}{l} \mathsf{size} : \mathbb{N};\ \ \mathsf{own} : \mathit{Clair}\ A \times \mathit{List}\ \mathit{Val} \times \mathit{ThreadId} \times \mathbb{N} \to \mathit{Fml}; \\ \qquad\qquad\ \ \mathsf{shr} : \mathit{Addr} \times \mathit{Lft} \times \mathit{Clair}\ A \times \mathit{ThreadId} \times \mathbb{N} \to \mathit{Fml};\ \ \right\}. \end{array}$$

The field size tells the low-level size in the memory. The field own models the ownership of the type `T` in separation logic. The field shr is a variant of that, used for shared references $\&\alpha\ \mathtt{T}$. This is the same as the domain used by RustHornBelt, just except that the syntactic formula *Fml* is used instead of the semantic proposition *iProp*. Also, the model of each Rust ownership type in RustHalt is pretty much the same as in RustHornBelt, except that later-free constructors in *Fml* are used for

higher-order ghost state. In particular, to model mutable and shared references in Rust, we use the borrower token $\&^{\alpha} P$ from Nola's later-free borrows (§ 5).

Remarkably, RustHalt enjoys *semantic* and *extensible* type soundness proof, or *semantic typing* [Timany et al. 2023], just as in RustBelt and RustHornBelt. To prove each type-spec rule, we just need to separately prove its semantic interpretation in the separation logic Iris. So we can easily add new Rust features and typing rules, enjoying a high level of extensibility. For that, each type-spec judgment is *semantically interpreted* as an SL assertion, namely as follows:

$$[\![\Gamma \vdash_{\gamma} \mathtt{e} \dashv \mathtt{r}.\Gamma' \rightsquigarrow pre]\!] \triangleq \forall \hat{post}, t, q.\ \left[\ \exists \bar{\hat{a}}.\ \left\langle \lambda\pi.\ pre\,(\hat{post}\,\pi)\,(\overline{\hat{a}\,\pi})\right\rangle * [\gamma]_q * [t] * [\![\Gamma]\!](\bar{\hat{a}}, t)\ \right] \mathtt{e} \left[\ \lambda \mathtt{r}.\ \exists \bar{\hat{b}}.\ \left\langle \lambda\pi.\ \hat{post}\,\pi\,(\overline{\hat{b}\,\pi})\right\rangle * [\gamma]_q * [t] * [\![\Gamma']\!](\bar{\hat{b}}, t)\ \right]^{\mathsf{Wrh}\,[\![\,]\!]}.$$

Again, this is almost the same as RustHornBelt's model, just except that we use the *total* Hoare triple with a custom world satisfaction $\mathsf{Wrh}\,[\![\,]\!]$, the separating conjunction of required world satisfactions including $\mathsf{Winv}\,[\![\,]\!]$ and $\mathsf{Wbor}\,[\![\,]\!]$. Here, $[\![\overline{\mathtt{a} :^{?} \mathtt{T}}]\!](\bar{\hat{a}}, t)$ is defined as the separating conjunction $\ast\, \overline{[\![\mathtt{a} :^{?} \mathtt{T}]\!](\hat{a}, t)}$ of the semantics for each object, where $[\![\mathtt{a} : \mathtt{T}]\!](\hat{a}, t)$ is defined as $\exists d.\ [\![\mathtt{T}\ (\hat{a}, [\mathtt{a}], t, d)]\!]$, where d is the depth of the object, an auxiliary parameter inherited from RustHornBelt. The SL assertion $\left\langle \lambda\pi.\ \phi_\pi \right\rangle$ is a *prophecy observation*, taken from the parametric prophecies machinery proposed by RustHornBelt [Matsushita et al. 2022]. Notably, in Nola, we have built a general mechanism called *prophetic borrows*, which sophisticates RustHornBelt's approach of mixing borrows and parametric prophecies, and RustHalt takes advantage of this mechanism. Please refer to the Rocq mechanization for the details.

8 Mechanization

Here we report on our Rocq mechanization for Nola, published as the artifact [Matsushita and Tsukada 2025]. Our core achievement is the mechanization of Nola's parameterized higher-order ghost state of invariant (§ 2, § 3) and borrow (§ 5), including the soundness of each proof rule with respect to the semantic model. For that, we mechanized a new library for parameterized view shifts and Hoare triples with a custom world satisfaction including new adequacy theorems (§ 2.1), and also extended λ_{Rust}, the core language of RustBelt [Jung et al. 2018a], to support parameterized Hoare triples. We also mechanized general libraries for magic derivability (§ 6) and for extensible construction of the formula data type and interpretation (explained in § 2.2). Using these libraries, we mechanized the verification examples discussed in the paper, including the simple examples of total correctness proof Examples 1.1 and 1.2 (§ 1.1), the examples on shared mutable lists (§ 2.3), the examples on borrows Examples 5.1 to 5.3 (§ 5.2), and the semantic alteration examples (6.1) (§ 6). We also mechanized paradoxes mentioned in the paper, including Paradox 1.3. Last but not least, we mechanized our primary case study, RustHalt (§ 7), including the type-spec rules and the verification examples discussed in the paper. Our Rocq mechanization does not depend on any axioms except the axiom of UIP (the uniqueness of identity proofs).

9 Related Work

Invariants with later-free rules. Some existing studies [Swamy et al. 2020; Svendsen and Birkedal 2014; Svendsen et al. 2013; Spies et al. 2022] provide separation logic with invariants with later-free proof rules, but none of them has been applied to termination verification. Moreover, all of them either use step-indexed program logic or restrict nesting of invariants, unlike Nola.

SteelCore [Swamy et al. 2020] is an automated SL-based verification framework that provides nested invariants with later-free proof rules. However, its program logic is actually step-indexed. As discussed in [Swamy et al. 2020, § 4.4], it uses "monotonic state" modeled by Ahman et al. [2017], which "has a 'later' modality in disguise". Moreover, it lacks a formal proof of the soundness.

The later credit $£\, n$ [Spies et al. 2022], a feature recently introduced to Iris, provides the *prepaid invariant* $\boxed{P}_{\mathsf{pre}}$ with a later-free access rule InvPreOpen for the partial Hoare triple $\{P\}\ \mathsf{ae}\ \{\Psi\}$ [Spies et al. 2022, § 6.1]. However, the later credit fundamentally depends on step-indexing of the program logic and thus does not work well in termination verification. In particular, we can prove that the prepaid invariant that works in the total Hoare triple causes a paradox of wrongly proving the termination of Landin's knot, by modifying the discussion of Paradox 1.3.

Separation logics iCAP [Svendsen and Birkedal 2014] and HOCAP [Svendsen et al. 2013] support invariants with later-free proof rules, whose soundness does not seem to rely on step-indexing. HOCAP is step-indexed, but Svendsen et al. [2013, § 4] says that this is just for supporting nested Hoare triples. So iCAP and HOCAP may be able to be modified for termination verification. Still, they *do not support genuine nesting of invariants*. iCAP does not allow any kind of nesting of the invariant connective. HOCAP prohibits nesting invariants of overlapping *region types* t, which is vital to HOCAP's soundness, because such nesting introduces "self-referential region assertions", which generally "do not admit modular stability proofs" [Svendsen et al. 2013, § 2.2]. Unlike theirs, Nola allows genuine nesting of invariants (e.g., infinite singly linked lists in § 2.3).

Liveness verification in step-indexed logic. As discussed in § 1.1, the existing approach to verifying liveness properties in step-indexed program logic, transfinite indexing [Spies et al. 2021b] and its application Transfinite Iris [Spies et al. 2021a], suffers from some fundamental limitations.

Some studies propose step-indexed separation logic that can verify termination-preserving refinements [Tassarotti et al. 2017; Timany et al. 2024]. To achieve this, they reduced such refinements (which may appear to be a liveness property) to a *safety* property, by imposing a strong constraint requiring that the stuttering of the target program with respect to the source should be bounded.

On the other hand, as we saw in earlier sections, Nola naturally supports verification of unbounded (non-fair) termination, which is a simple but genuine liveness property. (See also Current status and future applications in § 1.2 for detailed discussions.)

Tackling the later modality in safety verification. As mentioned in § 1.1, the later modality $\triangleright$ can be problematic even in *safety* verification. Although some workarounds have been proposed, they are generally cumbersome and limited in applicability.

An early example comes from RustBelt [Jung et al. 2018a]. They needed to convert a mutable reference $\&\alpha\ \mathtt{mut}\ \mathsf{T}$ into a shared reference $\&\alpha\ \mathsf{T}$. Naively, this requires traversing over the entire structure of the object T, but that was blocked by the later modality in the existing approach. Instead, RustBelt used a workaround called *delayed sharing* [Jung 2020, Chapter 12]: they gave up traversal in one go and instead performed the conversion only when each subobject gets accessed, like lazy evaluation. Unfortunately, this makes the model much more complicated, and in fact this delaying technique does not work for all types of ghost operations, as we will see below.

Another example comes from RustHornBelt [Matsushita et al. 2022]. They had to traverse the entire object to get the witness that each prophecy in the object is unresolved, but the delaying technique as used by RustBelt did not work because the witness should be obtained immediately. RustHornBelt tackled this by a technique dubbed *flexible step-indexing*: enrich the program logic so that *increasingly many laters* for each program step (more specifically, k laters at the k-th step) can be stripped, with the help of *time receipts* $⧖\, n$ [Mével et al. 2019] for lower-bounding k. Still, it requires a cumbersome bookkeeping of the number of program steps.

One recent progress is the later credit $£\, n$ [Spies et al. 2022], which owns the right to eliminate n laters under a modified view shift $\Rrightarrow|_{\mathsf{le}}$, satisfying $£\, n * {\triangleright}^n P \Rrightarrow|_{\mathsf{le}} P$. Still, even with the later credit, it is rather hard to traverse nested data structures *unboundedly many times* (e.g., RustHornBelt's case), because that requires an unbounded amount of later credit.

References

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. doi: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. doi:10.1109/LICS.2002.1029818

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. doi: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. doi:10.1145/504709.504712

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 147:1–147:30. doi:10.1145/3360573

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. doi:10.1007/978-3-642-11957-6_6

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). doi: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. doi:10.1016/j.tcs.2010.07.010

Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent Separation Logic. *ACM SIGLOG News* 3, 3 (2016), 47–65. doi: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. doi: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. doi:10.1016/j.entcs.2011.09.018

Arthur Charguéraud. 2011. Characteristic formulae for the verification of imperative programs. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 418–430. doi:10.1145/2034773.2034828

Arthur Charguéraud and François Pottier. 2015. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9236)*, Christian Urban and Xingyuan Zhang (Eds.). Springer, 137–153. doi:10.1007/978-3-319-22102-1_9

Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *J. Autom. Reason.* 62, 3 (2019), 331–365. doi:10.1007/S10817-017-9431-7

Arthur Charguéraud. 2025. *Separation Logic Foundations*. Software Foundations, Vol. 6. https://softwarefoundations.cis.upenn.edu/slf-current/ Version 2.3.

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. doi: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. doi:10.1007/978-3-031-17244-1_6

Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall. https://www.worldcat.org/oclc/01958445

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. doi:10.1145/3591278

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. doi:10.1145/3209108.3209174

Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1115–1139. doi:10.1145/3656422

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. doi: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. doi:10.1145/277650.277748

Pietro Di Gianantonio and Marino Miculan. 2002. A Unifying Approach to Recursive and Co-recursive Definitions. In *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers (Lecture Notes in Computer Science, Vol. 2646)*, Herman Geuvers and Freek Wiedijk (Eds.). Springer, 148–161. doi:10.1007/3-540-39185-1_9

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. doi: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. doi:10.1145/512529.512563

Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP (2022), 711–741. doi:10.1145/3547647

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. doi:10.1007/978-3-540-78739-6_27

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. doi: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. doi:10.1145/3527326

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. doi: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. doi: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. doi: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. doi: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. doi: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. doi:10.1145/2676726.2676980

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. 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. doi:10.1007/978-3-662-54434-1_26

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1

(2023), 286–315. doi:10.1145/3586037

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. doi:10.1145/3591253

Per Martin-Löf. 1982. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science VI*, L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 104. Elsevier, 153–175. doi:10.1016/S0049-237X(09)70189-2

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. doi:10.1145/2663171.2663188

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. doi:10.1145/3519939.3523704

Yusuke Matsushita and Takeshi Tsukada. 2025. *Artifact for "Nola: Later-Free Ghost State for Verifying Termination in Iris"*. doi:10.5281/zenodo.15050271

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. doi: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. doi:10.1145/3462205

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. doi:10.1007/978-3-030-17184-1_1

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. doi: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. doi:10.1109/LICS.2000.855774

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. doi:10.1007/978-3-540-28644-8_4

Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95. doi: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. doi: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. doi:10.1007/3-540-44802-0_1

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. doi:10.1109/LICS.2002.1029817

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

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL (2023), 1121–1151. doi:10.1145/3571232

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. doi: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. doi: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. doi:10.1145/3434294

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. doi: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. doi: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. doi: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. doi: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. doi: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. doi:10.1145/3158152

Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the Typed Call-by-Value lambda-Calculus using a Stack of Regions. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin (Eds.). ACM Press, 188–201. doi:10.1145/174675.177855

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based Memory Management. *Inf. Comput.* 132, 2 (1997), 109–176. doi:10.1006/INCO.1996.2613

Sebastian Ullrich. 2016. *Simple Verification of Rust Programs via Functional Purification.* Master's thesis. Karlsruhe Institute of Technology.

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

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. doi:10.1145/3473597

Received ; accepted