Mizzle: A Complete Concurrent Incorrectness Logic for Preventing False Alarms in Agentic Bug Finding

ALEXANDRE MOINE, New York University, USA
SAM WESTRICK, New York University, USA
JOSEPH TASSAROTTI*, New York University, USA

Large language models are increasingly used to find bugs in real-world programs, but they also produce a flood of false alarms that waste developers' time. We propose a method to prevent these false alarms by requiring an LLM to accompany each bug report with a machine-checked proof, in a program logic, that the reported bug is real. We follow the approach of incorrectness logics, whose under-approximate reasoning establishes that a claimed behavior is genuinely reachable, and hence a true positive. In our case, however, the logic must model a realistic programming language, have a mechanization so that proofs can be checked, and be complete, so that no real bug is ruled out for want of a derivation.

We present Mizzle, an incorrectness separation logic for concurrent programs written in a substantial subset of OCaml, parametric in the notion of incorrectness. We mechanize Mizzle in the Rocq proof assistant on top of the Iris framework, and we prove that it is both sound (that is, it never justifies a false alarm) and complete (that is, every incorrect execution admits a derivation). We instantiate Mizzle with three notions of incorrectness: stuckness (triggering undefined behavior), the non-linearizability of a data structure, and the presence of a race. As a proof of concept, we illustrate how an LLM can use Mizzle in order to certify the existence of a bug.

1 Introduction

False alarms in bug-finding tools waste developers' time. Interpreting a tool's output and determining whether a bug is real or spurious can sometimes take a great deal of effort. If a tool raises too many false alarms, a developer might begin to ignore its findings or stop using it altogether. As a result, some bug-finding tools are designed to avoid all false alarms and only report *true positives*: when they claim there is a bug, then the program really exhibits the buggy behavior. But how can one show formally that the analyses conducted by these tools really result in only true positives? Traditionally, doing so required ad-hoc proofs that could be quite challenging; in particular, manually crafting a formal semantics reduction of the program reaching a buggy state is prohibitively tedious. An alternative that has emerged in recent years is to use *Incorrectness Logic* [O'Hearn 2020] as a formal foundation for justifying that a program analysis only yields true positives. Whereas traditional Hoare logic over-approximates a program's behaviors, so that specifications show that a program's behaviors must belong to some set, incorrectness logics instead use *under*-approximation, so that a specification shows that some set of behaviors is a subset of a program's behaviors; *i.e.*, they are actual observable behaviors. By reformulating an analysis in terms of derivations in an incorrectness logic, one can thereby show that the analysis does not generate false alarms. This approach has been successfully applied to a number of static-analysis and bug-finding methods [Raad et al. 2020; Le et al. 2022; Raad et al. 2022, 2023].

Today, developers are awash in bug reports coming from a whole new class of bug-finding tools: large language models. Unfortunately, just as with traditional bug-finding tools, false alarms from LLMs *also* waste developers' time. Indeed, developers have become so inundated with spurious reports from LLMs that some popular open source projects like `curl` have announced closing their

*Also affiliated with Amazon Web Services. This paper does not reflect the views of Amazon Web Services.

Authors' Contact Information: Alexandre Moine, alexandre.moine@nyu.edu, New York University, New York, USA; Sam Westrick, shw8119@nyu.edu, New York University, New York, USA; Joseph Tassarotti, jt4767@nyu.edu, New York University, New York, USA.

bug bounty programs [Stenberg 2026]. At the same time, LLMs are remarkably good at finding subtle and serious bugs that are beyond the scope of traditional methods. Thus, a natural question is whether we can find a way to retain the benefits of LLM bug finding while reducing the incidence of false alarms. Today, one popular method for doing so involves throwing yet more LLMs at the problem, say by asking multiple models to scrutinize a bug report. While this may help filter out some false alarms, it does not offer any guarantees. Another approach is to require the LLM to construct a reproducible test case that triggers the bug. Unfortunately, for some classes of bugs, reliable reproductions can be hard to construct. For example, concurrency race bugs might only be triggered non-deterministically under certain execution schedules.

This paper proposes an alternate approach of using incorrectness logic to prevent false alarms in LLM bug reports. Of course, LLMs are a black box, so unlike previous efforts to apply incorrectness logic to bug-finding tools, we cannot use incorrectness logic to justify the kinds of analyses that LLMs perform. Instead, we propose that an LLM producing a bug report should be prompted to produce a derivation in an incorrectness logic showing that the bug is real. In particular, by mechanizing the incorrectness logic inside of a proof assistant such as Rocq, and requiring the LLM to carry out a proof using that embedding, we can use the proof assistant to check that the proof is valid. Moreover, if the LLM's first attempt turns out not to describe a real bug, it can interactively improve this first attempt until a true bug is found.

For this approach to be viable, we need an incorrectness logic with several properties. First, it must handle programs written in a real programming language, modeling a substantial portion of that language's features. Finding bugs in programs written in an idealized toy language does not address the problem. Second, the incorrectness logic must have a mechanization in a proof assistant or some other tool that makes it possible to check that the derivations are constructed correctly. Third, the logic should be *complete*: if a bug exists, then it must be possible to construct a derivation in the logic showing so. Otherwise, we risk ruling out a real bug simply because it is impossible to use the logic to exhibit it.

Unfortunately, to the best of our knowledge, no existing incorrectness logic has these properties. To address this, we present Mizzle, a concurrent incorrectness separation logic for a substantial subset of OCaml 5, and demonstrate how Mizzle can be used by an LLM to generate machine-checked proofs of incorrectness properties. More precisely, we make the following contributions:

- Mizzle, an incorrectness separation logic for concurrent programs written in a substantial subset of OCaml 5, parametric in the notion of incorrectness.
- An embedding of Mizzle in the Rocq proof assistant with proofs of *soundness* (no false alarms) and *completeness* (every incorrect execution admits a derivation) [Anon. 2026].
- An instantiation of Mizzle with three notions of incorrectness: stuckness (that is, triggering undefined behavior), the non-linearizability of a data structure, and the presence of a memory race.
- An evaluation showing how an LLM can use Mizzle to prove incorrectness.

Note that this paper does not claim to improve LLMs' ability to find bugs, nor do we argue that LLMs are a better method of finding bugs than other tools. Rather, given that LLMs are being used for bug finding, our goal with Mizzle is to provide a way to provably show that an alleged bug is not a false alarm.

2 Key Ideas

In this section, we present the key ideas of Mizzle. We start with a motivating example: a concrete buggy concurrent stack in OCaml 5 (§ 2.1), and a companion test client in which only some interleavings can trigger the bug. To reason about this program, we use ZooLang [Allain and

```
let create () = Atomic.make []

let rec push t v =
  let xs = Atomic.get t in
  let ys = v :: xs in
  (* Should be: Atomic.compare_and_set t xs ys *)
  if not (Atomic.compare_and_set t (Atomic.get t) ys)
  then push t v

let rec pop t =
  match Atomic.get t with
  | [] -> None
  | hd :: tl as xs ->
    if Atomic.compare_and_set t xs tl
    then Some hd else pop t

let client () =
  let t = create () in
  let f i = push t i; assert (pop t <> None) in
  ignore (Domain.spawn (fun () -> f 1));
  f 2
```

Fig. 1. A buggy concurrent stack

Scherer 2026] as a formal operational model of OCaml 5 (§2.2). Next, we present the key rules of Mizzle and explain how to prove the existence of the bug by showing a stuck configuration is reachable (§2.3). Mizzle is in fact parameterized by the notion of incorrectness being looked for, and stuckness is just one instantiation. We illustrate this generality with a second instantiation: non-linearizability, and how it can be applied to our buggy concurrent stack (§2.4). This is a key novelty of Mizzle over previous incorrectness logics, as it allows for non-linearizability to be exhibited directly, rather than having to encode non-linearizability as some other form of observable fault. We then state the soundness and completeness theorems of Mizzle, specialized to both stuckness and non-linearizability (§2.5).

2.1 A Buggy Concurrent Stack

While the ideas behind Mizzle are language-agnostic, we show its use with OCaml. We focus in particular on OCaml 5, which natively supports shared-memory concurrency via coarse-grained threads called *domains* [Sivaramakrishnan et al. 2020].

Figure 1 presents an implementation of a (buggy) variant of Treiber's concurrent stack [Treiber 1986]. The API of the stack is composed of three functions: `create ()`, which creates a new stack, `push t v`, which pushes a value `v` on the stack `t`, and `pop t`, which pops a value from the stack `t`, wrapped in an option type to capture the case of an empty stack. The stack is implemented as a reference to an immutable list of values. The function `create ()` returns a reference to an empty list. The function `push t v` is buggy: it loads the content of the stack `t` in a local variable `xs`, then builds a new list `ys` by consing `v` to `xs`. It then attempts to atomically install `ys` with a compare-and-set, retrying on failure. As we will see, the bug is that the compare-and-set tests against a *fresh* read of `t` instead of the snapshot `xs`. The function `pop t` is correct: it reads the stack and, if non-empty, atomically replaces it with its tail using a compare-and-set, retrying on failure.

Figure 1 then presents a client which might trigger the bug. Two threads, one created with `Domain.spawn` (returning a handle for a possible later join, which we ignore here), push a value and then pop one. Since each thread pops only after its own push, each pop should return `Some` of a value. Yet, some interleavings of our client yield a pop operation returning `None`, which is a symptom of the buggy `push`. Concretely, the bug arises when both pushes take their snapshot `xs` of the empty stack before either commits. The first compare-and-set installs a single-element list. The second then runs against this updated stack. Because it compares the content of the stack against a *fresh* read of `t`, with the single-element list, the compare-and-set succeeds and silently overwrites the first push. The stack thus holds a single element instead of two, so the second pop

finds it empty, returns **None**, and fails the assertion. On a standard laptop, it often takes more than a thousand runs to trigger the assertion failure.

2.2 ZooLang as a Model of OCaml 5

In order to formally reason about OCaml 5 programs, we need a *model* of the language, that is, a formal description of its syntax and semantics. We base Mizzle on ZooLang, a recently developed model of OCaml 5 allowing for the verification of fine-grained concurrent OCaml 5 algorithms with the Zoo program logic [Allain and Scherer 2026]. Zoo comes with a tool to translate OCaml 5 programs into ZooLang expressions in Rocq. ZooLang captures many of the pertinent details of the concurrency semantics of OCaml 5, for example including thread-local storage and a subtle definition of physical equality. Indeed, physical equality is underspecified in OCaml 5 for values involving immutable blocks, as sharing or other optimizations may render two apparently equal values physically different, and apparently different values physically equal. As Allain and Scherer [2026] mention, this is a problem while reasoning about concurrent primitives such as compare-and-set, which test for physical equality. This behavior, which Mizzle can test, makes our buggy stack example potentially even more buggy.

We present the syntax of ZooLang in Section 3.1. ZooLang is equipped with a small-step interleaving semantics, which we reuse with small modifications described later in Section 3.2. We write a single step with the reduction relation $c_1 \longrightarrow c_2$, where c_1 and c_2 are two *configurations*, recording all the threads currently executing, the heap, and a ghost trace of certain actions. We write the reflexive-transitive closure of the reduction relation as $c_1 \longrightarrow^* c_2$.

2.3 Mizzle at a Glance

Intuitively, Mizzle is an *angelic* logic: it allows the user to prove the existence of one specific interleaving exhibiting a bug. As such, our approach is inspired by the Angelic logic of Moine et al. [2026], which we adapted to the setting of bug finding.

Mizzle is built using the Iris separation logic framework [Jung et al. 2018], and we follow its syntax for separation logic assertions. We denote an Iris assertion by φ, a separating conjunction by $\varphi * \varphi'$, and a separating implication (also known as magic wand) by $\varphi \mathrel{-\!\!*} \varphi'$. A meta-logical proposition U is called *pure* and written $\ulcorner U \urcorner$. The *points-to assertion* $\ell \mapsto \vec{v}$ says that the memory location ℓ is the first location of a memory block that contains the values $\vec{v}$, that is, for any offset i such that $0 \le i < |\vec{v}|$, the location $\ell + i$ contains the value $\vec{v}(i)$. Mizzle is not meant to be used by hand, but rather by an automated tool, and in particular an LLM. For simplicity, we will say "the user" throughout the paper to refer to the agent using Mizzle.

To show that e_c, the ZooLang translation of the client expression in Figure 1, has a bug, the user must prove the following entailment in Mizzle, which features the logic's three key assertions:

$$\mathsf{thread}\ \pi_0\ \epsilon\ e_c\ *\ \mathsf{remaining}\ (\mathsf{witness}\ \mathsf{Stuck})\ \vdash\ \mathfrak{goal}$$

Let us present the three assertions in this entailment from left to right. First, the assertion thread $\pi\,\vec{\kappa}\,e$ says that a thread with identifier π is executing the expression e under evaluation context $\vec{\kappa}$. In the above entailment, π is instantiated with π_0, the initial identifier, $\vec{\kappa}$ with ϵ, the empty evaluation context, and e with e_c, the client expression. Second, the assertion remaining φ states the proof obligation that the user must show about the program's execution: the user has to exhibit φ to complete the proof. In the above entailment, φ is instantiated with (witness Stuck), where Stuck is a predicate over configurations asserting that some thread is stuck, that is, there exists a thread that has not reached a value but cannot take a step. In particular, any configuration with an `assert false` is stuck.

$$\textsc{M-Assert}\quad \mathsf{thread}\,\pi\,\vec{\kappa}\,(\mathsf{assert}\,\mathsf{true}) \;*\; (\mathsf{thread}\,\pi\,\vec{\kappa}\,() \mathrel{-\!\!*} \mathfrak{goal}) \;\vdash\; \mathfrak{goal}$$

$$\textsc{M-CAS-Succ}\quad \left(\begin{array}{r}\mathsf{thread}\,\pi\,\vec{\kappa}\,(\mathsf{CAS}\,(\ell,i)\,v_1\,v_2) \;*\\ \ell \mapsto \vec{v} \;*\; \ulcorner 0 \le i < |\vec{v}| \;\wedge\; (\vec{v}(i) \approx v_1)\urcorner \;*\\ (\ell \mapsto [i := v_2]\vec{v} \mathrel{-\!\!*} \mathsf{thread}\,\pi\,\vec{\kappa}\,\mathsf{true} \mathrel{-\!\!*} \mathfrak{goal})\end{array}\right) \vdash \mathfrak{goal}$$

$$\textsc{M-CAS-Fail}\quad \left(\begin{array}{r}\mathsf{thread}\,\pi\,\vec{\kappa}\,(\mathsf{CAS}\,(\ell,i)\,v_1\,v_2) \;*\\ \ell \mapsto \vec{v} \;*\; \ulcorner 0 \le i < |\vec{v}| \;\wedge\; (\vec{v}(i) \not\approx v_1)\urcorner \;*\\ (\ell \mapsto \vec{v} \mathrel{-\!\!*} \mathsf{thread}\,\pi\,\vec{\kappa}\,\mathsf{false} \mathrel{-\!\!*} \mathfrak{goal})\end{array}\right) \vdash \mathfrak{goal}$$

$$\textsc{M-Fork}\quad \left(\begin{array}{r}\mathsf{thread}\,\pi\,\vec{\kappa}\,(\mathsf{fork}\,e) \;*\\ (\forall \pi'.\ \mathsf{thread}\,\pi'\,\epsilon\,e \mathrel{-\!\!*} \mathsf{thread}\,\pi\,\vec{\kappa}\,() \mathrel{-\!\!*} \mathfrak{goal})\end{array}\right) \vdash \mathfrak{goal}$$

$$\textsc{AssertStuck}\quad \mathsf{thread}\,\pi\,\vec{\kappa}\,(\mathsf{assert}\,v) \;*\; \ulcorner v \neq \mathsf{true}\urcorner \;\vdash\; \mathsf{witness}\,\mathsf{Stuck}$$

$$\textsc{RemainingMono}\quad \mathsf{remaining}\,\varphi_1 \;*\; (\varphi_1 \mathrel{-\!\!*} \varphi_2) \;\vdash\; \mathsf{remaining}\,\varphi_2$$

$$\textsc{M-Conclude}\quad \mathsf{remaining}\,\top \;\vdash\; \mathfrak{goal}$$

Fig. 2. Key reasoning rules of Mizzle: the assert function, the remaining assertion, forks, and compare-and-set

One might wonder: if this remaining φ assertion describes what the user has to *show*, why is it on the left side of the entailment and not on the right? The answer is that Mizzle uses a continuation-passing style, with the $\mathfrak{goal}$ assertion on the right side of the entailment as an opaque placeholder. During a Mizzle proof, the right side of the entailment is always this $\mathfrak{goal}$ assertion, while the context holds one or more thread assertions. To step a thread, a rule consumes its thread assertion and asks the user to re-establish $\mathfrak{goal}$ under an updated thread, with the remaining assertion telling us what we need to show to discharge $\mathfrak{goal}$.

Key reasoning rules. Figure 2 presents a selection of reasoning rules of Mizzle, allowing us to verify the above entailment. The upper part of the figure covers the reasoning rules for advancing the execution of a thread. M-Assert handles the case where the assertion succeeds: some thread π is executing assert true under evaluation context $\vec{\kappa}$, and the conclusion is $\mathfrak{goal}$. Then, the user can proceed with the execution by assuming that the thread π faces the unit value () under evaluation context $\vec{\kappa}$, and $\mathfrak{goal}$ remains the target. M-CAS-Succ and M-CAS-Fail handle a compare-and-set $\mathsf{CAS}\,(\ell, i)\,v_1\,v_2$, which attempts to replace the value stored at location $\ell + i$ by v_2, and succeeds only if that value is currently the expected value v_1. They both require the points-to assertion $\ell \mapsto \vec{v}$ and that the offset i is in bounds. The value currently stored at the location is thus $\vec{v}(i)$. The two rules compare it against the expected value v_1 using physical equality (§2.2). A succeeding comparison is captured by the relation $\approx$ and a failed one by the relation $\not\approx$. Note that, because of the underspecification of physical equality in OCaml 5, the two relations $\approx$ and $\not\approx$ are not complementary. On success ($\vec{v}(i) \approx v_1$), the block is updated by storing v_2 at offset i; on failure ($\vec{v}(i) \not\approx v_1$), the block is left unchanged. M-Fork spawns a new thread. The premise produces a fresh, distinct identifier π' running e under the empty evaluation context ϵ, while the forking thread π resumes with ().

The lower part of Figure 2 covers the reasoning rules for concluding a proof. AssertStuck is a stuckness rule: it allows for concluding witness Stuck if some thread π is executing assert v under evaluation context $\vec{\kappa}$, and v is a value distinct from the Boolean true. RemainingMono allows for weakening the remaining assertion by discharging part of the requirement it represents. For example, the user can use this rule to deduce $\mathsf{remaining}\,(\mathsf{witness}\,\mathsf{Stuck}) \mathrel{-\!\!*} \mathsf{witness}\,\mathsf{Stuck} \mathrel{-\!\!*} \mathsf{remaining}\,\top$, where $\top$ is the always-true assertion. M-Conclude is the sole rule of Mizzle allowing one to finish the proof: it consumes $\mathsf{remaining}\,\top$, the witness that no proof obligation remains, and finally

produces the target $\mathfrak{goal}$. M-Conclude is the only rule that produces $\mathfrak{goal}$ in its conclusion, without any precondition requiring the user to derive $\mathfrak{goal}$ itself.

Proving that our Stack is Buggy. We now sketch how these rules let us prove that the client of Figure 1 can get stuck. For clarity, in this section, we write in plain letters list constructors `Nil` and `Cons`. Our target is to establish the entailment presented earlier:

$$\mathsf{thread}\ \pi_0\ \epsilon\ e_c\ *\ \mathsf{remaining}\ (\mathsf{witness\ Stuck})\ \vdash\ \mathfrak{goal}$$

Allocating the stack yields a points-to assertion $\ell \mapsto$ `[Nil]` for a one-cell block whose single value is the empty list, and M-Fork spawns the second thread, so that two threads are both about to push, leading to the following entailment:

$$\mathsf{thread}\ \pi_0\ \vec{\kappa}_a\ (\mathsf{push}\ \ell\ 2)\ *\ \mathsf{thread}\ \pi_1\ \vec{\kappa}_a\ (\mathsf{push}\ \ell\ 1)\ *\ \ell \mapsto [\mathtt{Nil}]\ *\ \mathsf{remaining}\ (\mathsf{witness\ Stuck})\ \vdash\ \mathfrak{goal}$$

where $\vec{\kappa}_a$ captures the pending assert for both threads. The user is free to advance the threads in any order by applying the rules from Figure 7, choosing the interleaving that exhibits the bug. We chose to make both threads enter the body of `push`, and load ℓ (using M-LoadMut from Figure 7). They observe the `Nil` constructor, and build their respective new lists—that is, the list `(Cons 2 Nil)` for thread π_0, and the list `(Cons 1 Nil)` for thread π_1. We now focus on thread π_1. It performs a load on ℓ again, reading again the empty list, and faces now a compare-and-set operation:

$$\begin{array}{c}\mathsf{thread}\ \pi_0\ \vec{\kappa}_a\ (\mathsf{push}\ \ell\ 2)\ *\\ \mathsf{thread}\ \pi_1\ \vec{\kappa}_i\ (\mathsf{Atomic.compare\_and\_set}\ \ell\ \mathtt{Nil}\ (\mathtt{Cons\ 1\ Nil}))\ *\quad \vdash\ \mathfrak{goal}\\ \ell \mapsto [\mathtt{Nil}]\ *\ \mathsf{remaining}\ (\mathsf{witness\ Stuck})\end{array}$$

where $\vec{\kappa}_i$ captures the pending "if not" (which will fail) and the follow-up assert. Its compare-and-set succeeds by M-CAS-Succ, storing `(Cons 1 Nil)` in ℓ. We then make π_1 return from push and advance to its assert operation. We now focus on thread π_0. It performs a load on ℓ, but this time loads `(Cons 1 Nil)`, and faces its own compare-and-set operation.

$$\begin{array}{c}\mathsf{thread}\ \pi_0\ \vec{\kappa}_i\ (\mathsf{Atomic.compare\_and\_set}\ \ell\ (\mathtt{Cons\ 1\ Nil})\ (\mathtt{Cons\ 2\ Nil}))\ *\\ \mathsf{thread}\ \pi_1\ \epsilon\ (\mathsf{assert}\ (\mathsf{pop}\ \ell\ <>\ \mathsf{None}))\ *\qquad \vdash\ \mathfrak{goal}\\ \ell \mapsto [\mathtt{Cons\ 1\ Nil}]\ *\ \mathsf{remaining}\ (\mathsf{witness\ Stuck})\end{array}$$

Here again, because the buggy code compares against this *fresh* read of ℓ rather than its previous snapshot, M-CAS-Succ applies, and ℓ is overwritten. The stack thus holds a single value `(Cons 2 Nil)`. After both pushes return, thread π_0 runs `pop t`, which reads the singleton, succeeds with its compare-and-set, and returns **`Some`**; its assertion holds by M-Assert. Thread π_1 then runs `pop t` on the now-empty stack, which returns **`None`**, so its assertion reduces to assert false. At this point, the target is:

$$\mathsf{thread}\ \pi_0\ \epsilon\ ()\ *\ \mathsf{thread}\ \pi_1\ \epsilon\ (\mathsf{assert\ false})\ *\ \ell \mapsto [\mathtt{Nil}]\ *\ \mathsf{remaining}\ (\mathsf{witness\ Stuck})\ \vdash\ \mathfrak{goal}$$

We can now use AssertStuck to deduce witness Stuck, and RemainingMono to consume the remaining (witness Stuck) obligation, leaving remaining $\top$. A final application of M-Conclude produces the target $\mathfrak{goal}$, completing the proof that the client can get stuck.

2.4 Proving Non-Linearizability with Mizzle

In order to show that the stack of Figure 1 is buggy, we had to exhibit a particular client. Yet, the defect lies not in the client, but in the stack itself: it is *not* linearizable. Linearizability [Herlihy and Wing 1990] is a key correctness criterion for concurrent data structures. Intuitively, an operation on a data structure is linearizable if, even though it may execute multiple computation steps, its effect on the data structure appears to take place instantaneously at a single point between its call and return, the so-called *linearization point*. Alternatively, a history of concurrent operations on

$$\begin{array}{rcl} \text{Model} & \mathcal{M} & = \left(\mathbb{S},\ s_0 \in \mathbb{S},\ \text{Valid} \subseteq \mathit{Val},\ \xrightarrow{\Rightarrow} \subseteq \mathbb{S} \times \mathit{Val} \times \mathit{Val} \times \mathbb{S}\right) \\ \text{Event} & \eta & \triangleq \ \text{Call}\, v \mid \text{Lin}\, v\, v' \mid \text{Ret}\, v' \\ \text{History} & \theta & \in \ \text{List}(\mathit{Tag} \times \text{Event}) \end{array}$$

$$\textsc{M-Emit}\quad \frac{}{\text{thread}\,\pi\,\vec{\kappa}\,(\text{emit}\,v) \ast \text{trace}\,\theta \ast (\text{trace}\,(\theta +\!\!+ [v]) \mathrel{-\!\!\ast} \text{thread}\,\pi\,\vec{\kappa}\,() \mathrel{-\!\!\ast} \mathfrak{goal}) \vdash \mathfrak{goal}}$$

$$\textsc{NonLin}\quad \frac{}{\ulcorner \neg(\text{Linearizable}\,\theta)\urcorner \ast \text{trace}\,\theta \vdash \text{witness NonLinearizable}}$$

Fig. 3. Reasoning rules for non-linearizable traces

a data structure is linearizable if it can be explained as a sequential history. Non-linearizability constitutes the real issue with our running example: the failing assertion of Section 2.1 is only a symptom, exhibited by one particular client. Indeed, the stack admits histories that no sequential stack could produce.

Linearizability is a *trace* property, as one needs to consider the trace of function calls and returns to determine whether a history is linearizable. Thus, to expose the trace, we make use of a primitive emit v, which emits the value v to the trace, a piece of ghost state.

We give the formal definition of linearizability later in Section 4.2. For now, Figure 3 presents the relevant aspects needed to understand how a user proves non-linearizability. First, we consider a *model* $\mathcal{M}$ of the data structure, which is a specification of how a sequential version of the data structure, such as a stack or queue, behaves. A model is a tuple of a set of states $\mathbb{S}$, an initial state s_0, a set Valid of valid arguments for the operation, and a relation $s \xrightarrow{v \Rightarrow v'} s'$ describing the permissible transitions from state s to state s' when a call with argument v returns v'. We then define the notion of an *event* η: either a call with some argument v, a linearization of a call with argument v returning v', or the effective return point of a call, returning v'. We modify operations to emit the call and return events. Meanwhile, the linearization event is not emitted; instead, given a trace, the definition of linearizability requires annotating the trace with linearization events in a way that is consistent with some sequential behavior of the model. A *history* θ is a list of tagged events, where the tag, a natural number, allows for matching up call events to their corresponding returns in the history. We write that a history θ is *linearizable* as $\text{Linearizable}\,\theta$, with the formal definition in Section 4.2.

For dealing with traces, Mizzle provides the assertion $\text{trace}\,\theta$, witnessing that the observation trace is currently θ. The lower part of Figure 3 presents two reasoning rules of Mizzle, key for proving non-linearizability using traces. First, M-Emit appends a value to the trace. It requires the assertion $\text{trace}\,\theta$, and returns it updated to $\text{trace}\,(\theta +\!\!+ [v])$, the thread reducing to the unit value $()$. Second, NonLin allows for concluding the assertion (witness NonLinearizable), if we can deduce the pure meta-logic fact that the current trace θ is not linearizable.

Putting these pieces together, to prove that a data structure is not linearizable, it suffices to exhibit an execution that generates a non-linearizable trace, that is, a succession of calls and returns that cannot be explained as a sequential history.

The most general client. We cannot just consider any possible client that might interact with the data structure: a client that violates the abstraction boundaries of the data structure and directly modifies its internal state is not a demonstration of a bug in the data structure. We avoid this issue of invalid clients and relieve the user from having to construct clients explicitly by defining a function called the *most general client* (a standard technique, see Dongol and Derrick [2015]), presented in Figure 4. For simplicity, we suppose that the data structure is equipped with a single operation taking a single argument and returning a value. (Data structures with multiple operations can be emulated by passing a tuple with a tag to indicate which operation to select.)

```
let most_general_client init op gen =
  let obj = init () in
  let onecall i =
    let x = gen () in emit (i, Call x);
    let r = op obj x in emit (i, Ret r) in
  let rec client i =
    if nondet_bool () then () else
    (ignore (Domain.spawn (fun () -> onecall i)); client (i + 1)) in
  client 1
```

Fig. 4. The most general client

The function `most_general_client` takes three arguments: `init`, initializing the data structure, `op`, applying an operation on the data structure, and `gen`, generating arguments for the operation. The `most_general_client` initializes the data structure, then repeatedly non-deterministically spawns threads. Each thread runs one function call, emitting a **`Call`** event before the call and a **`Ret`** event after it. We call this the most general client because, for every execution of a valid client of the data structure, there is some choice of `gen` and a corresponding execution of the `most_general_client` that yields the same interleaving of calls.

Proving Non-Linearizability for our Buggy Stack. We instantiate the most general client with our stack, taking `init` to be `create` and letting `op` dispatch on a tagged argument to invoke either `push` or `pop`. We use a `gen` function that randomly generates either pops or pushes with an argument of `1` or `2`. We then use the rules of Mizzle to generate an execution that produces the trace

$$\theta \triangleq [\,(1, \mathsf{Call}\,(\mathsf{push}\,1));\ (1, \mathsf{Ret}\,());\ (2, \mathsf{Call}\,(\mathsf{push}\,2));\ (2, \mathsf{Ret}\,());\\ (3, \mathsf{Call}\,\mathsf{pop});\ (3, \mathsf{Ret}\,(\mathsf{Some}\,2));\ (4, \mathsf{Call}\,\mathsf{pop});\ (4, \mathsf{Ret}\,\mathsf{None})\,]$$

where both pushes return before either pop is called. This history is not linearizable: since the two pops follow both completed pushes, any sequential explanation would have to pop the two pushed values, returning Some twice. Yet, operation 4 observes None, the symptom of the lost push. Hence $\neg(\mathsf{Linearizable}\,\theta)$ holds. In particular, in our Rocq mechanization, since the definitions of the sequential stack model are computable, the user can apply a decision procedure that shows $\neg(\mathsf{Linearizable}\,\theta)$.

The proof that there exists an interleaving of the call to the most general client exhibiting the above trace reuses the lost-update interleaving of Section 2.3. Let us write e_{mgc} for the expression that invokes the most general client with the arguments described above. The initial entailment to prove is:

$$\mathsf{thread}\,\pi_0\,\epsilon\,e_{mgc} \;*\; \mathsf{trace}\,\epsilon \;*\; \mathsf{remaining}\,(\mathsf{witness}\,\mathsf{NonLinearizable}) \vdash \mathfrak{goal}$$

With helper lemmas provided by Mizzle (one for starting the most general client, one for initiating a call, and one for returning from a call), we then drive the four threads so as to produce θ. Each `emit` is justified by M-Emit: it consumes the current trace θ' and returns trace $(\theta' +\!\!+ [v])$, appending one event at a time until the trace is exactly θ. Once the full history has been emitted, NonLin applies: it requires trace θ and the pure fact $\neg(\mathsf{Linearizable}\,\theta)$ established above, and produces $(\mathsf{witness}\,\mathsf{NonLinearizable})$, witnessing non-linearizability. This discharges the remaining proof obligation, so RemainingMono leaves remaining $\top$ and M-Conclude produces the target $\mathfrak{goal}$.

2.5 Soundness and Completeness

Mizzle is sound—intuitively, if the user proves a $\mathfrak{goal}$ assertion, then they have effectively constructed a buggy execution—and complete—intuitively, if there exists a buggy execution, there exists

a Mizzle derivation. Sections 4.3 and 4.4 present a general soundness theorem and a general completeness theorem, parameterized by the notion of "incorrectness" being considered. In this section we state corollaries of these theorems for our two instantiations, stuckness and non-linearizability.

Soundness. The soundness theorem says that a Mizzle proof with a remaining (witness φ) proof obligation effectively constructs a reduction path starting from $\mathsf{init}(e)$ (the initial configuration with an empty heap and empty trace) that witnesses the incorrectness property φ. For stuckness, the constructed reduction path reaches a stuck configuration, whereas for non-linearizability, the reduction path instead emits a non-linearizable history.

Corollary 2.1 (Soundness for Stuckness). *Let e be an expression and π_0 the initial thread identifier. If thread π_0 ∈ e ∗ remaining (witness Stuck) ⊢ $\mathfrak{goal}$ holds, then there exists a configuration c such that init(e) ⟶* c and Stuck c hold.*

Corollary 2.2 (Soundness for Non-Linearizability). *Let e be a call to the most general client and π_0 the initial thread identifier. If thread π_0 ∈ e ∗ trace ∈ ∗ remaining (witness NonLinearizable) ⊢ $\mathfrak{goal}$ holds, then there exists a configuration c such that init(e) ⟶* c and NonLinearizable c hold.*

Completeness. Conversely, the completeness theorem assumes the existence of a reduction path from the initial configuration $\mathsf{init}(e)$ reaching an incorrect configuration c, and concludes that there exists a Mizzle derivation of $\mathfrak{goal}$. The one caveat is that it assumes that e contains no hard-coded locations, because Mizzle has no rules to reason about them. This is not a serious restriction, since a valid OCaml 5 program that is translated with Zoo's translation tool into ZooLang cannot have such locations, because well-typed OCaml 5 programs do not provide a way to refer directly to the memory address backing a reference.

Corollary 2.3 (Completeness for Stuckness). *Let e be an expression and c a configuration. If locs$(e) = \emptyset$, and if init(e) ⟶* c holds with Stuck c, then thread π_0 ∈ e ∗ remaining (witness Stuck) ⊢ $\mathfrak{goal}$ holds.*

Corollary 2.4 (Completeness for Non-Linearizability). *Let e be a call to the most general client and c a configuration. If locs$(e) = \emptyset$, and if init(e) ⟶* c holds with c containing a location-free, non-linearizable history, then thread π_0 ∈ e ∗ trace ∈ ∗ remaining (witness NonLinearizable) ⊢ $\mathfrak{goal}$ holds.*

Note that the corollary for non-linearizability assumes that the emitted history contains no locations. This precondition is a simplification: as we explain in Section 4.4, the general completeness theorem makes strict assumptions about locations because Mizzle treats them abstractly.

3 Syntax and Semantics

In this section, we describe ZooLang [Allain and Scherer 2026], a logical model of OCaml 5, which Mizzle reasons about. We first describe its syntax (§ 3.1), a call-by-value lambda calculus with mutable state and concurrency. We then give its operational semantics (§3.2), a small-step reduction relation modeling the interleaving concurrency of OCaml 5's domains.

3.1 Syntax

Figure 5 presents the syntax of ZooLang, a call-by-value lambda calculus with mutable state and concurrency, faithfully modeling OCaml 5's semantics. A value $v \in \mathit{Val}$ is either a Boolean $b \in \mathbb{B}$, an idealized integer $i \in \mathbb{Z}$, a location ℓ from an infinite set of locations Loc, a recursive function $\overline{\mathsf{rec}}\ f\ x = e$, or an immutable *block* $\langle n \mid \vec{v} \rangle$, a record carrying a constructor tag $n \in \mathbb{N}$ and a sequence of

Mutability	$\rho \triangleq$ mut | imm			
Unary ops	$\ominus \triangleq \sim \mid - \mid$ is_imm			
Binary ops	$\oplus \triangleq + \mid - \mid \times \mid$ quot $\mid$ rem $\mid$ land $\mid$ lor $\mid$ lsl $\mid$ lsr $\mid \leq \mid < \mid \geq \mid >$			
Values *Val*	$v \triangleq b \in \mathbb{B} \mid i \in \mathbb{Z} \mid \ell \in Loc \mid \overline{\text{rec}}\, f\, x = e \mid \langle n \mid \vec{v} \rangle$			
Branches	$br \triangleq n\, \overline{x} \Rightarrow e$			
Expressions	$e \triangleq v, w$	*value*	block $\rho\, n\, \vec{e}$	*block construction*
	x	*variable*	alloc $e\, e$	*array allocation*
	rec $f\, x = e$	*abstraction*	match e with $\vec{br}$	*pattern matching*
	$e\, e$	*call*	tag e	*block tag*
	let $x = e$ in e	*let binding*	size e	*block size*
	$\ominus\, e$	*unary operation*	$e.[e]$	*load*
	$e \oplus e$	*binary operation*	$e.[e] \leftarrow e$	*store*
	$e == e$	*physical equality*	CAS $e\, e\, e$	*compare-and-set*
	if e then e else e	*conditional*	fork e	*fork*
	emit e	*trace emission*		
Contexts	$\kappa \triangleq \square\, v$	$\mid e\, \square$	$\mid$ let $x = \square$ in e	$\mid \ominus\, \square$
	$\mid$ if $\square$ then e else e	$\mid \square \oplus v$	$\mid e \oplus \square$	$\mid \square == v$
	$\mid e == \square$	$\mid$ match $\square$ with $\vec{br}$	$\mid$ block $\rho\, n\, (\vec{e} +\!\!+ \square +\!\!+ \vec{v})$	$\mid$ tag $\square$
	$\mid$ size $\square$	$\mid \square.[v]$	$\mid e.[\square]$	$\mid \square.[v] \leftarrow v$
	$\mid e.[\square] \leftarrow v$	$\mid e.[e] \leftarrow \square$	$\mid$ CAS $\square\, v\, v$	$\mid$ CAS $e\, \square\, v$
	$\mid$ CAS $e\, e\, \square$	$\mid$ alloc $\square\, v$	$\mid$ alloc $e\, \square$	

Fig. 5. Syntax of ZooLang

fields $\vec{v}$. Mutable blocks are instead heap-allocated and denoted by their location ℓ. Constructor tags allow distinguishing between different constructors of an algebraic data type. For example, in the case of the type `option` in OCaml, the data **`None`** is represented by the block $\langle 0 \mid \rangle$, while the data **`Some`** v is represented by the block $\langle 1 \mid v \rangle$. The Rocq implementation of ZooLang offers notations and machinery to manipulate OCaml's algebraic constructors instead of raw tags and blocks.

A computation is described by an expression e, whose syntax is mostly standard. Unary and binary primitive operations $\ominus\, e$ and $e \oplus e$ cover the usual arithmetic, bitwise and comparison operators, while $e == e$ tests physical equality. Blocks model both records and the constructors of algebraic data types: block $\rho\, n\, \vec{e}$ builds a block with tag n and fields $\vec{e}$, either mutable (ρ = mut) or immutable (ρ = imm). Mutable blocks are meant to be allocated on the heap, while immutable blocks will be evaluated to a value, after a single step. The expression match e with $\vec{br}$ inspects a block by its tag and binds its fields; and tag e and size e retrieve a block's tag and number of fields. So far, blocks have a statically-known size. The expression alloc $n\, m$ allocates a fresh block of m uninitialized fields (all set to ()), where m may be a runtime value. The expression $e.[e]$ reads a field at a given offset of a block, $e.[e] \leftarrow e$ writes it, and the atomic primitive CAS $e\, e\, e$ provides compare-and-set, where the first argument is a tuple of a location and an offset. The primitive emit v appends the value v to a global observation trace, and returns the unit value (). Finally, fork e spawns a concurrent thread. A (non-recursive) evaluation context κ describes an expression with a hole $\square$ and dictates the right-to-left evaluation order of ZooLang. We write $\kappa[e]$ for the expression obtained by filling the hole of κ with e, and write similarly $\vec{\kappa}[e]$ when multiple context are stacked.

The previous Section 2 features the assert x function, which is defined in ZooLang as assert $x \triangleq$ if x then () else () (), the function that returns the unit value () if its argument is true and becomes stuck otherwise, by applying the unit value to itself.

Differences with the real ZooLang. For exposition purposes, this paper presents a subset of ZooLang; the mechanized development uses the full language, which differs as follows. First, we

$$\textsc{Alloc}\quad \frac{0 \le m \qquad \big(\{\ell\} \cup \{\ell + i \mid 0 \le i < m\}\big) \cap \big(\mathrm{dom}(\tau) \cup \mathrm{dom}(\sigma)\big) = \emptyset}{\mathsf{alloc}\, n\, m \,/\, \sigma \,/\, \tau \,/\, \theta \to_{\mathsf{h}} \ell \,/\, [\ell + i := () \mid 0 \le i < m]\sigma \,/\, [\ell := (n, m)]\tau \,/\, \theta \;;\; \epsilon}$$

$$\textsc{BlockMut}\quad \frac{0 < |\vec{v}| \qquad \big(\{\ell\} \cup \{\ell + i \mid 0 \le i < |\vec{v}|\}\big) \cap \big(\mathrm{dom}(\tau) \cup \mathrm{dom}(\sigma)\big) = \emptyset}{\mathsf{block\ mut}\, n\, \vec{v} \,/\, \sigma \,/\, \tau \,/\, \theta \to_{\mathsf{h}} \ell \,/\, [\ell + i := \vec{v}(i) \mid 0 \le i < |\vec{v}|]\sigma \,/\, [\ell := (n, |\vec{v}|)]\tau \,/\, \theta \;;\; \epsilon}$$

$$\textsc{BlockImmut}\quad \mathsf{block\ imm}\, n\, \vec{v} \,/\, \sigma \,/\, \tau \,/\, \theta \to_{\mathsf{h}} \langle n \mid \vec{v} \rangle \,/\, \sigma \,/\, \tau \,/\, \theta \;;\; \epsilon$$

$$\textsc{LoadMut}\quad \frac{\sigma(\ell + i) = v \qquad \tau(\ell) = (n, m) \qquad 0 \le i < m}{\ell.[i] \,/\, \sigma \,/\, \tau \,/\, \theta \to_{\mathsf{h}} v \,/\, \sigma \,/\, \tau \,/\, \theta \;;\; \epsilon}$$

$$\textsc{LoadImmut}\quad \frac{0 \le i < |\vec{v}|}{\langle n \mid \vec{v} \rangle.[i] \,/\, \sigma \,/\, \tau \,/\, \theta \to_{\mathsf{h}} \vec{v}(i) \,/\, \sigma \,/\, \tau \,/\, \theta \;;\; \epsilon}$$

$$\textsc{Store}\quad \frac{\ell + i \in \mathrm{dom}(\sigma) \qquad \tau(\ell) = (n, m) \qquad 0 \le i < m}{\ell.[i] \leftarrow v \,/\, \sigma \,/\, \tau \,/\, \theta \to_{\mathsf{h}} () \,/\, [\ell + i := v]\sigma \,/\, \tau \,/\, \theta \;;\; \epsilon}$$

$$\textsc{Fork}\quad \mathsf{fork}\, e \,/\, \sigma \,/\, \tau \,/\, \theta \to_{\mathsf{h}} () \,/\, \sigma \,/\, \tau \,/\, \theta \;;\; [e]$$

$$\textsc{Emit}\quad \mathsf{emit}\, v \,/\, \sigma \,/\, \tau \,/\, \theta \to_{\mathsf{h}} () \,/\, \sigma \,/\, \tau \,/\, \theta +\!\!+ [v] \;;\; \epsilon$$

$$\textsc{Context}\quad \frac{e \,/\, \sigma \,/\, \tau \,/\, \theta \to_{\mathsf{h}} e' \,/\, \sigma' \,/\, \tau' \,/\, \theta' \;;\; \vec{e}_f}{\vec{\kappa}[e] \,/\, \sigma \,/\, \tau \,/\, \theta \to_{\mathsf{t}} \vec{\kappa}[e'] \,/\, \sigma' \,/\, \tau' \,/\, \theta' \;;\; \vec{e}_f}$$

$$\textsc{Interleave}\quad \frac{\vec{e}(\pi) = e \qquad e \,/\, \sigma \,/\, \tau \,/\, \theta \to_{\mathsf{t}} e' \,/\, \sigma' \,/\, \tau' \,/\, \theta' \;;\; \vec{e}_f}{\vec{e} \,/\, \sigma \,/\, \tau \,/\, \theta \to [\pi := e']\vec{e} +\!\!+ \vec{e}_f \,/\, \sigma' \,/\, \tau' \,/\, \theta'}$$

Fig. 6. Selected rules of the semantics

omit *generativity*, a mechanism that gives immutable blocks an observable identity. Second, ZooLang functions are in fact *mutually recursive* bundles together with a selector index; we ignore this aspect. Third, the match construct of ZooLang additionally features a default branch, binding the scrutinee, which we elide. Fourth, we omit the atomic operations Xchg (exchange) and FAA (fetch-and-add). Fifth, we omit the bounded for loop, which is otherwise standard. Sixth, we omit thread-local state, accessed in ZooLang through dedicated getLocal and setLocal primitives. Finally, we repurpose *prophecy variables* for trace reasoning. Indeed, prophecy variables are ghost annotations used in the original Zoo paper to reason about concurrent data structures. A prophecy variable is a ghost value to which is attached a trace, and the program can "resolve" the prophecy, which appends a value to the prophecy's trace. In the context of Mizzle, we force a single ambient prophecy variable to be present in every configuration, and we implement emit e on top of the resolve primitive.

Apart from prophecies, our mechanization [Anon. 2026] supports all the features of ZooLang, and our soundness Theorem 4.1 and completeness Theorem 4.2 are verified against the full language.

3.2 Semantics

A *configuration* $c = \vec{e} \,/\, \sigma \,/\, \tau \,/\, \theta$ is a quadruple of a list of expressions $\vec{e}$, one per thread, a heap σ, a tag map τ, and a trace θ. The heap σ is a partial function mapping locations to values. The tag map τ records, for each allocated block, its header (n, m): its constructor tag and its size m. The trace θ is the list of values emitted so far.

Figure 6 presents selected rules of the semantics of ZooLang, structured as three increasingly coarse reduction relations. The *head reduction relation* $e \,/\, \sigma \,/\, \tau \,/\, \theta \to_{\mathsf{h}} e' \,/\, \sigma' \,/\, \tau' \,/\, \theta' \;;\; \vec{e}_f$ describes a single computation step of the expression e. It relates the expression and the state, including the trace θ, to their reducts, together with the (possibly empty, written ϵ) list $\vec{e}_f$ of threads spawned by the step. For example, BlockMut allocates a fresh mutable block. BlockImmut evaluates an immutable block to a value. Alloc allocates a fresh mutable block of m fields, all initialized

to (); unlike BlockMut, its size m is a runtime value and may be zero, yielding a header with no cell. LoadMut and Store read and write a field of a mutable block. LoadImmut reads a field of an immutable block, directly from its value. Emit reduces to the unit value and appends its argument v to the trace, yielding $\theta +\!\!+ [v]$. Fork spawns a thread and reduces to the unit value. The *deep reduction relation* $e \mid \sigma \mid \tau \mid \theta \to_{\mathrm{t}} e' \mid \sigma' \mid \tau' \mid \theta' \;;\; \vec{e}_f$ closes head reduction under evaluation contexts. Context locates the next redex deep inside e, following the right-to-left evaluation order dictated by the contexts κ of Figure 5, and executes one head step. The deep reduction relation describes the evolution of a single thread. The *reduction relation* $c_1 \to c_2$ operates on whole configurations. Interleave nondeterministically selects one thread π from the pool $\vec{e}$, takes a deep step on it, and updates the configuration accordingly, replacing thread π by its reduct, appending any spawned threads $\vec{e}_f$, and carrying over the updated trace θ'. This last relation models the interleaving concurrency of OCaml 5's domains.

Out-of-bound accesses. Out-of-bound accesses are a source of bugs that are not prevented by OCaml's type system. As a mitigation, OCaml's standard library functions usually come in two flavors: a default version testing that the accessed offset is within bounds, raising an exception otherwise, and an optimized version bypassing the check. For example, the `Array` module of the OCaml standard library offers two primitives to load the content of an offset within an array: `Array.get` and `Array.unsafe_get`.

Mizzle is able to catch out-of-bound errors, whether they are checked at runtime (in this case, looking for a stuckness bug suffices), or not (since such errors can be encoded as predicates over configurations, see Section 4.2). However, while an out-of-bound access can lead to a segmentation fault and the termination of the execution, it can sometimes land on another allocated object, read some data, and silently proceed. Mizzle is not able to catch bugs that are reached after such a successful out-of-bound read. Indeed, the M-BlockMut rule gives no control over the universally-quantified location, hence it is not possible to force the alignment of memory blocks. Arguably, a bug that can only be reached after a successful out-of-bound access is not the priority—the out-of-bound access is.

Hence, the reduction relation $c_1 \to c_2$ of Figure 6 prevents out-of-bound accesses: the field-access rules guard every access with an in-bounds side condition (*e.g.*, $0 \le i < m$ in LoadMut). Writing $c_1 \xrightarrow{zoo} c_2$ for the original Zoo semantics, which performs no out-of-bound check, we prove that our relation satisfies $c_1 \to c_2 \iff c_1 \xrightarrow{zoo} c_2 \;\wedge\; \mathsf{NoOoB}\, c_1$, where $\mathsf{NoOoB}\, c$ is a predicate that analyzes the next expression being reduced, and verifies that any store access is within bounds.

4 The Mizzle Program Logic

In this section, we present Mizzle in depth. We first give its reasoning rules for the cases when "things go well", that is, when no incorrectness is witnessed (§4.1). We then explain how the user defines their own notion of incorrectness, and illustrate it with stuckness, non-linearizability, and memory races (§4.2). We next state and discuss the general soundness (§4.3) and completeness (§4.4) theorems. Finally, we take a closer look at the definition of the `goal` assertion (§4.5).

4.1 Reasoning Rules When Things Go Well

To represent mutable state, Mizzle features two assertions. First, the points-to assertion, as described in Section 2.3. Second, a companion assertion header $\ell\, n\, m$ witnesses that the memory location ℓ contains a block of size m tagged with n. This assertion is *persistent*, hence in particular duplicable.

Figure 7 presents Mizzle's reasoning rules when no form of incorrectness is witnessed. In fact, each of these rules corresponds to a step of the incorrectness derivation: each premise mentions a

$$\textsc{M-BlockMut}\quad \left(\begin{array}{r}\mathsf{thread}\,\pi\,\vec{\kappa}\,(\mathsf{block\ mut}\,n\,\vec{v}) \;*\; \ulcorner 0 < |\vec{v}| \urcorner \;*\\ (\forall \ell.\ \mathsf{header}\,\ell\,n\,|\vec{v}| \mathrel{-\!\!*} \ell \mapsto \vec{v} \mathrel{-\!\!*} \mathsf{thread}\,\pi\,\vec{\kappa}\,\ell \mathrel{-\!\!*} \mathfrak{goal})\end{array}\right) \vdash \mathfrak{goal}$$

$$\textsc{M-BlockImmut}\quad \mathsf{thread}\,\pi\,\vec{\kappa}\,(\mathsf{block\ imm}\,n\,\vec{v}) \;*\; (\mathsf{thread}\,\pi\,\vec{\kappa}\,\langle n \mid \vec{v}\rangle \mathrel{-\!\!*} \mathfrak{goal}) \vdash \mathfrak{goal}$$

$$\textsc{M-Alloc}\quad \left(\begin{array}{r}\mathsf{thread}\,\pi\,\vec{\kappa}\,(\mathsf{alloc}\,n\,m) \;*\; \ulcorner 0 \le m \urcorner \;*\\ (\forall \ell.\ \mathsf{header}\,\ell\,n\,m \mathrel{-\!\!*} \ell \mapsto ()^m \mathrel{-\!\!*} \mathsf{live}\,\ell \mathrel{-\!\!*} \mathsf{thread}\,\pi\,\vec{\kappa}\,\ell \mathrel{-\!\!*} \mathfrak{goal})\end{array}\right) \vdash \mathfrak{goal}$$

$$\textsc{M-LoadMut}\quad \left(\begin{array}{r}\mathsf{thread}\,\pi\,\vec{\kappa}\,(\ell.[i]) \;*\; \ell \mapsto \vec{v} \;*\; \ulcorner 0 \le i < |\vec{v}| \urcorner \;*\\ (\ell \mapsto \vec{v} \mathrel{-\!\!*} \mathsf{thread}\,\pi\,\vec{\kappa}\,(\vec{v}(i)) \mathrel{-\!\!*} \mathfrak{goal})\end{array}\right) \vdash \mathfrak{goal}$$

$$\textsc{M-LoadImmut}\quad \left(\begin{array}{r}\mathsf{thread}\,\pi\,\vec{\kappa}\,(\langle n \mid \vec{v}\rangle.[i]) \;*\; \ulcorner 0 \le i < |\vec{v}| \urcorner \;*\\ (\mathsf{thread}\,\pi\,\vec{\kappa}\,(\vec{v}(i)) \mathrel{-\!\!*} \mathfrak{goal})\end{array}\right) \vdash \mathfrak{goal}$$

$$\textsc{M-Store}\quad \left(\begin{array}{r}\mathsf{thread}\,\pi\,\vec{\kappa}\,(\ell.[i] \leftarrow w) \;*\; \ell \mapsto \vec{v} \;*\; \ulcorner 0 \le i < |\vec{v}| \urcorner \;*\\ (\ell \mapsto [i := w]\vec{v} \mathrel{-\!\!*} \mathsf{thread}\,\pi\,\vec{\kappa}\,() \mathrel{-\!\!*} \mathfrak{goal})\end{array}\right) \vdash \mathfrak{goal}$$

$$\textsc{M-Context}\quad \mathsf{thread}\,\pi\,(\vec{\kappa}_1 +\!\!+ \kappa_2)\,e \;\dashv\vdash\; \mathsf{thread}\,\pi\,\vec{\kappa}_1\,(\kappa_2[e])$$

Fig. 7. Reasoning rules for block construction, allocation, loads, and stores

$\mathsf{thread}\,\pi\,\vec{\kappa}\,e$, some resources to execute the step faced by e and a continuation-style assertion that describes the resources and threads after the step.

M-BlockMut allocates a mutable block from the values $\vec{v}$, provided the block is non-empty ($0 < |\vec{v}|$). In exchange, the user obtains, for a freshly chosen location ℓ, the header assertion $\mathsf{header}\,\ell\,n\,|\vec{v}|$, the points-to $\ell \mapsto \vec{v}$, and the thread now reduced to the value ℓ. M-Alloc is the analogue for the alloc $n\,m$ primitive: it allocates a fresh block of m fields, all initialized to $()$. Unlike M-BlockMut, the size m is an arbitrary runtime value subject only to $0 \le m$, so the block may be empty. In exchange, the user obtains, for a freshly chosen location ℓ, the header assertion $\mathsf{header}\,\ell\,n\,m$, the points-to $\ell \mapsto ()^m$, and the thread reduced to ℓ. The additional, conditional assertion $\ulcorner m = 0 \urcorner \mathrel{-\!\!*} \mathsf{live}\,\ell$ hands back the live-location token $\mathsf{live}\,\ell$ only in the degenerate case $m = 0$. This token allows the user to separate a block with size 0 from another location, with the rule $\mathsf{live}\,\ell_1 * \ell_2 \mapsto \vec{v} \mathrel{-\!\!*} \ulcorner \ell_1 \neq \ell_2 \urcorner$. M-BlockImmut is simpler: an immutable block reduces directly to the value $\langle n \mid \vec{v}\rangle$, with no location and no points-to issued, reflecting that immutable blocks are pure values that need not be tracked in the heap. M-LoadMut reads the location $\ell + i$. It requires the assertion $\ell \mapsto \vec{v}$ and an in-bounds index ($0 \le i < |\vec{v}|$); the points-to assertion is returned unchanged, and the thread reduces to $\vec{v}(i)$. M-LoadImmut is the analogue for an immutable block: since the block is a pure value carrying its own fields, no points-to is required; the only requirement is that the offset i is in-bounds; the thread reduces to $\vec{v}(i)$. M-Store writes the value w into location $\ell + i$. It requires the assertion $\ell \mapsto \vec{v}$ and returns it updated to $\ell \mapsto [i := w]\vec{v}$, the thread reducing to $()$. M-Context allows for rearranging evaluation contexts, from the second parameter of the thread assertion to the expression itself. Note that Mizzle has no Bind rule, that is, a rule allowing for forgetting about the evaluation context while focusing on the current expression.

As we saw before, these rules are written in a continuation-passing style. Yet, they preserve the standard rules of separation logic, and in particular the frame rule, which can be stated as follows:

$$\textsc{Frame}\quad \frac{\varphi \;*\; (\varphi' \mathrel{-\!\!*} \mathfrak{goal}) \vdash \mathfrak{goal}}{\varphi \;*\; \psi \;*\; ((\varphi' \;*\; \psi) \mathrel{-\!\!*} \mathfrak{goal}) \vdash \mathfrak{goal}}$$

Reading from top to bottom, this rule asserts that, if the user can prove $\mathfrak{goal}$ from φ and a continuation $\varphi' \mathrel{-\!\!*} \mathfrak{goal}$, then they can also prove $\mathfrak{goal}$ from φ and any additional resources ψ, which are also transmitted to the continuation.

$$\begin{array}{rl}
\textsc{InterpThread} & \text{interp}\,(\vec{e} \,/\, \sigma \,/\, \tau \,/\, \theta) \;*\; \text{thread}\,\pi\,\vec{\kappa}\,e \;\vdash\; \ulcorner \vec{e}(\pi) = \vec{\kappa}[e] \urcorner \\
\textsc{InterpPointsTo} & \text{interp}\,(\vec{e} \,/\, \sigma \,/\, \tau \,/\, \theta) \;*\; \ell \mapsto \vec{v} \;\vdash\; \ulcorner \forall i.\, 0 \le i < |\vec{v}| \Rightarrow \sigma(\ell + i) = \vec{v}(i) \urcorner \\
\textsc{InterpHeader} & \text{interp}\,(\vec{e} \,/\, \sigma \,/\, \tau \,/\, \theta) \;*\; \text{header}\,\ell\,n\,m \;\vdash\; \ulcorner \tau(\ell) = (n, m) \urcorner \\
\textsc{InterpTrace} & \text{interp}\,(\vec{e} \,/\, \sigma \,/\, \tau \,/\, \theta) \;*\; \text{trace}\,\theta' \;\vdash\; \ulcorner \theta' = \theta \urcorner
\end{array}$$

Fig. 8. Reasoning rules for the interp predicate

$$\text{Stuck}\,(\vec{e} \,/\, \sigma \,/\, \tau \,/\, \theta) \triangleq \exists \pi\, e_\pi.\, \vec{e}(\pi) = e_\pi \wedge e_\pi \notin \mathit{Val} \wedge \neg(\exists e'\, \sigma'\, \tau'\, \theta'\, \vec{e}_f.\, e_\pi \,/\, \sigma \,/\, \tau \,/\, \theta \to_{\mathrm{t}} e' \,/\, \sigma' \,/\, \tau' \,/\, \theta' \;;\; \vec{e}_f)$$

$$\begin{array}{rl}
\textsc{StuckLoad} & \text{thread}\,\pi\,\vec{\kappa}\,(v_1.[v_2]) \;*\; \ulcorner (v_1 \notin \mathit{Loc} \wedge \forall n\, \vec{v}.\, v_1 \neq \langle n \mid \vec{v} \rangle) \vee v_2 \notin \mathbb{Z} \urcorner \;\vdash\; \text{witness Stuck} \\
\textsc{StuckLoadImmut} & \text{thread}\,\pi\,\vec{\kappa}\,(\langle n \mid \vec{v} \rangle.[i]) \;*\; \ulcorner i < 0 \vee |\vec{v}| \le i \urcorner \;\vdash\; \text{witness Stuck} \\
\textsc{StuckLoadMut} & \text{header}\,\ell\,n\,m \;*\; \text{thread}\,\pi\,\vec{\kappa}\,(\ell.[i]) \;*\; \ulcorner i < 0 \vee m \le i \urcorner \;\vdash\; \text{witness Stuck}
\end{array}$$

Fig. 9. The Stuck predicate, and stuckness rules for load operations

4.2 Defining Incorrectness

In Mizzle, the user can define their own notion of incorrectness, or more precisely, the property they want to hold on execution traces constructed by the user. We saw two examples of this in Section 2: stuckness and non-linearizability. This section gives more formal detail on how these notions of incorrectness are defined, and defines a third notion: memory races. We first present a key assertion: the witness assertion, together with the state interpretation predicate.

Demanding a witness with the state interpretation predicate. In order to relate separation logic assertions to the state of the operational semantics, we make available to the user the *state interpretation predicate*, written interp c, where $c = \vec{e} \,/\, \sigma \,/\, \tau \,/\, \theta$ is a configuration with threads $\vec{e}$, heap σ, tag map τ, and trace θ. Intuitively, if interp c holds, it means that the current execution trace constructed by the user reached the configuration c.

Figure 8 presents reasoning rules for exploiting the thread, points-to and header assertions, assuming that interp $(\vec{e} \,/\, \sigma \,/\, \tau \,/\, \theta)$ holds. Indeed, InterpThread allows for deducing that if thread $\pi\,\vec{\kappa}\,e$ holds, then $\vec{\kappa}[e]$ is the π-th expression of $\vec{e}$. InterpPointsTo allows for deducing that if $\ell \mapsto \vec{v}$ holds, then for any valid offset i, $\sigma(\ell + i)$ is the i-th value of $\vec{v}$. InterpHeader allows for deducing that if header $\ell\,n\,m$ holds, then $\tau(\ell)$ is the header (n, m). InterpTrace allows for deducing that if trace θ' holds, then θ' is the configuration's trace θ.

We then define the assertion witness as:

$$\text{witness}\,P \;\triangleq\; \forall c.\, \text{interp}\,c \mathrel{-\!\!*} \ulcorner P\,c \urcorner$$

Intuitively, if witness P holds, it means that under the assumption that the current execution trace reached a configuration c, the property $P\,c$ holds.

Stuckness. The first line of Figure 9 presents the definition of the Stuck predicate. The predicate Stuck c holds if there exists a thread π whose expression e_π is not a value, and that cannot take a step in the semantics.

The center part of Figure 9 presents reasoning rules for proving that a thread is stuck because of a load operation. Three cases are possible. StuckLoad captures the ill-typed case, that is, if the load operation is not applied to a block or a location, or if the offset is not an integer. StuckLoadImmut captures the out-of-bound case for an immutable block. StuckLoadMut captures the out-of-bound case for a mutable block. The out-of-boundness is exhibited thanks to the header assertion. From the definition of (witness Stuck), the reader might infer how we prove, for example, StuckLoadMut. We use the header to deduce that the offset is wrong using InterpHeader, then use the thread assertion to deduce that indeed a load operation is being executed on this offset using InterpThread, and conclude.

$$\textsc{SoundLinRefl}\ \ \mathsf{SoundLin}\ \epsilon\ s\ s \qquad \textsc{SoundLinCons}\ \ \frac{\mathsf{SoundLin}\ \theta\ s_0\ s \qquad \mathsf{Valid}\ v \qquad s \xrightarrow{v \Rightarrow v'} s'}{\mathsf{SoundLin}\ (\theta +\!\!+ [(\tau, \mathsf{Lin}\ v\ v')])\ s_0\ s'}$$

$$\begin{aligned}
\mathsf{ValidCalls}\ \theta &\triangleq \forall i\ \tau\ v.\ \theta(i) = (\tau, \mathsf{Call}\ v) \implies \mathsf{Valid}\ v \\
\mathsf{PerTag}\ \theta &\triangleq \forall \tau.\ \exists v\ v'.\ \theta\restriction_\tau \sqsubseteq [\ (\tau, \mathsf{Call}\ v);\ (\tau, \mathsf{Lin}\ v\ v');\ (\tau, \mathsf{Ret}\ v')\ ] \\
\mathsf{Linearizable}\ \theta &\triangleq \mathsf{ValidCalls}\ \theta \wedge \exists \theta'.\ \theta = (\theta'\restriction_{\mathsf{call,ret}}) \wedge \mathsf{PerTag}\ \theta' \wedge \exists s'.\ \mathsf{SoundLin}\ (\theta'\restriction_{\mathsf{lin}})\ s_0\ s' \\
\mathsf{NonLinearizable}\ (\vec{e} \,/\, \sigma \,/\, \tau \,/\, \theta) &\triangleq \neg(\mathsf{Linearizable}\ \theta)
\end{aligned}$$

Fig. 10. Linearizability as a trace property

Non-Linearizability. Figure 10 presents the formal definition of linearizability, relying on the model $\mathcal{M}$ introduced in Figure 3.

As alluded to in Figure 3, the purpose of the linearization events is to annotate the trace with markers indicating in what order the corresponding operations are applied. The upper part of the figure presents the judgment $\mathsf{SoundLin}\ \theta\ s_0\ s$, asserting that applying the linearization events of θ in the order they occur updates the model from state s_0 to state s. SoundLinRefl handles the empty trace, leaving the state unchanged. SoundLinCons extends a trace with one linearization event $(\tau, \mathsf{Lin}\ v\ v')$, provided the argument v is valid, and that the model admits the transition $s \xrightarrow{v \Rightarrow v'} s'$. The lower part of Figure 10 then defines three trace predicates. First, $\mathsf{ValidCalls}\ \theta$ requires every call event in θ to carry a valid argument. Second, $\mathsf{PerTag}\ \theta$ requires that, for each tag τ, the projection $\theta\restriction_\tau$ of θ, selecting only the events associated with that tag, is a prefix of a well-formed operation $[\ (\tau, \mathsf{Call}\ v);\ (\tau, \mathsf{Lin}\ v\ v');\ (\tau, \mathsf{Ret}\ v')\ ]$; that is, each operation calls, linearizes, then returns, in this order, with matching arguments and results. Finally, a trace θ is linearizable, written $\mathsf{Linearizable}\ \theta$, if its calls are valid, and it can be enriched with linearization events into a trace θ' that satisfies several properties. The first property requires that θ is θ' deprived of its linearization events, $\theta'\restriction_{\mathsf{call,ret}}$. Next, θ' is well-formed per tag, meaning that for each tag τ, if we project θ' down to just the events with tag τ, the projected trace is a prefix of a call/linearize/return sequence. Finally, applying those linearization events must update the model from s_0 to some state s'. Lifting this notion to configurations, a configuration is non-linearizable, written $\mathsf{NonLinearizable}\ c$, when its trace is not linearizable, that is, when no such witness θ' exists.

Races. Stuckness and non-linearizability are not the only notions of incorrectness for concurrent programs. Another important class of bugs consists of *data races*: two concurrent non-atomic accesses to a shared memory location, with at least one of them being a write operation.

We note that ZooLang does not distinguish between atomic and non-atomic accesses, and that the interleaving semantics of ZooLang essentially treats all accesses as atomic. For our purposes, we only need a notion of "racy access" which is suitable for finding real bugs. We therefore define a predicate Racy which identifies configurations where two threads are each poised to perform an access at the same location, with at least one of these accesses performing a write operation. This notion of a race is not exactly the same as the typical notion of a data race, because it identifies races even with conflicting *atomic* accesses. (Our notion of race is close to the notion of *determinacy races* [Feng and Leiserson 1997].) In our evaluation (§5.3), we only consider programs where all accesses are meant to be interpreted as non-atomic, and in this setting, Racy serves as a reasonable notion of incorrectness. Should ZooLang be extended with atomic accesses, it would be straightforward to extend our approach to restrict Racy to only consider non-atomic accesses.

In our setting, a race manifests itself purely syntactically on the configuration. The upper part of Figure 11 presents the $\mathsf{LAcc}\ e\ \ell\ b$ predicate, capturing that expression e accesses location ℓ; the

$$\textsc{LAccLoad}\quad \mathsf{LAcc}\,(\ell.[i])\,(\ell+i)\,\bot \qquad \textsc{LAccStore}\quad \mathsf{LAcc}\,(\ell.[i]\leftarrow w)\,(\ell+i)\,\top \qquad \textsc{AccCtx}\quad \frac{\mathsf{LAcc}\,e\,\ell\,b}{\mathsf{Acc}\,\vec{\kappa}[e]\,\ell\,b}$$

$$\mathsf{Racy}\,(\vec{e}\,/\,\sigma\,/\,\tau\,/\,\theta) \triangleq \exists \pi_1\,e_1\,\pi_2\,e_2\,\ell\,b_1\,b_2.\ \vec{e}(\pi_1)=e_1 \wedge \vec{e}(\pi_2)=e_2 \wedge \pi_1\neq\pi_2 \wedge \mathsf{Acc}\,e_1\,\ell\,b_1 \wedge \mathsf{Acc}\,e_2\,\ell\,b_2 \wedge (b_1\vee b_2)$$

$$\textsc{IsRacy}\quad \mathsf{thread}\,\pi_1\,\kappa_1\,e_1 \ast \mathsf{thread}\,\pi_2\,\kappa_2\,e_2 \ast \ulcorner \exists \ell\,b_1\,b_2.\ \mathsf{LAcc}\,e_1\,\ell\,b_1 \wedge \mathsf{LAcc}\,e_2\,\ell\,b_2 \wedge (b_1\vee b_2)\urcorner \vdash \mathsf{witness\ Racy}$$

Fig. 11. The LAcc and Acc access predicates, and the Racy predicate

$$\begin{aligned}\mathsf{mirror}\,(\vec{e}\,/\,\sigma\,/\,\tau\,/\,\theta) \quad \triangleq \quad & \exists \psi.\ \ulcorner \mathsf{mirror\_invs}\,\psi\,\vec{e}\,\sigma\,\tau\urcorner \ast \mathop{\Large\ast}_{(\pi\mapsto e)\,\in\,\vec{e}} \left(\mathsf{thread}\,\pi\,\epsilon\,\lfloor e\rfloor_\psi\right)\\ & \ast \mathop{\Large\ast}_{(\ell\mapsto(n,m))\,\in\,\tau} \left(\mathsf{header}\,\lfloor\ell\rfloor_\psi\,n\,m \ast \mathop{\Large\ast}_{0\le i<m} \left((\lfloor\ell\rfloor_\psi+i)\mapsto\lfloor\sigma(\ell+i)\rfloor_\psi\right)\right)\\ & \ast \mathop{\Large\ast}_{\ell\,\in\,\mathrm{dom}\,\tau\setminus\mathrm{dom}\,\sigma} \left(\mathsf{live}\,\lfloor\ell\rfloor_\psi\right) \ast \mathsf{trace}\,\lfloor\theta\rfloor_\psi\end{aligned}$$

Fig. 12. The mirror assertion

boolean b indicates whether the access may be a write operation. The assertion $\mathsf{Acc}\,e\,\ell\,b$ lifts the LAcc predicate to arbitrary evaluation contexts. Figure 11 then presents the property $\mathsf{Racy}\,(\vec{e}\,/\,\sigma\,/\,\tau\,/\,\theta)$, formalizing a race: it asserts the existence of two different threads which are accessing the same location, with at least one of them being a write operation.

The lower part of Figure 11 presents IsRacy, the main reasoning rule allowing the user to prove the (witness Racy) separation logic assertion. The rule consumes two threads π_1 and π_2 executing expressions e_1 and e_2, each one of them satisfying the LAcc predicate on the same location ℓ, with at least one of the two accesses being a write operation. In order to prove IsRacy, we unfold the definition of witness, and get that (witness Racy) is equivalent to $\forall c.\ \mathsf{interp}\,c \mathrel{-\!\!*} \ulcorner \mathsf{Racy}\,c\urcorner$. Hence, we use the thread assertion to deduce that indeed two threads are executing e_1 and e_2 under some evaluation contexts, then use the separating conjunction to deduce that $\pi_1 \neq \pi_2$, and conclude.

4.3 Soundness of Mizzle

The soundness theorem of Mizzle states that if the user can prove $\mathfrak{goal}$ from the initial thread π_0 executing e with a proof obligation (witness P), for some meta-level predicate P on configurations, then there exists a configuration c reachable from the initial configuration $\mathsf{init}(e)$ such that $P\,c$ holds. The initial configuration $\mathsf{init}(e)$ starts with the empty heap and the empty trace ϵ.

Theorem 4.1 (Soundness of Mizzle). *Let e be an expression, π_0 the initial thread identifier, and P a meta-level predicate on configurations. If $\mathsf{thread}\,\pi_0\,\epsilon\,e \ast \mathit{remaining}\,(\mathit{witness}\,P) \vdash \mathfrak{goal}$ holds, then there exists a configuration c such that $\mathit{init}(e) \to^* c$ and $P\,c$ hold.*

Using Theorem 4.1, we can easily prove the specialized Corollary 2.1 for stuckness, Corollary 2.2 for non-linearizability, and a similar theorem for races (which we omit for conciseness), by simply instantiating P to Stuck, NonLinearizable, and Racy, respectively. Note that, in the case of non-linearizability, Corollary 2.2 specializes e to be a call to the most general client. This statement actually holds without this restriction, but simply does not provide any guarantee of the placement of emit operations.

4.4 Completeness of Mizzle

The completeness theorem of Mizzle intuitively reads as "if there exists a reduction reaching a configuration satisfying some property, then there exists a Mizzle derivation with a proof obligation capturing the configuration's properties at the separation logic level". This latter part, "the

configuration's properties at the separation logic level" is captured by: (1) all the thread assertions for each thread, (2) all the points-to assertions for all the allocated memory locations, and (3) all the header assertions for each allocated block. Yet, we face a technical problem: recall that Mizzle does not give the user the ability to pick the locations of allocations: M-BlockMut returns a *universally quantified* location. If we try to construct a Mizzle derivation following the reduction path from the hypothesis, we are going to reach another configuration, similar to the one in the hypothesis, but up to a renaming of the locations. In the following, if ψ is a partial mapping from locations to locations, we write $\lfloor x \rfloor_\psi$ for the translation of x under ψ, where x is an expression or a value.

Thus, to state completeness properly, we need to restrict attention to incorrectness properties that do not depend on specific choices of allocation locations. Figure 12 presents the mirror $(\vec{e} \mid \sigma \mid \tau \mid \theta)$ assertion, which formalizes this idea. This assertion requires the existence of a bijection ψ between the locations of the configuration and the locations of the separation logic assertions. The fact that ψ is a bijection with the correct domain is captured by the pure assertion $\mathsf{mirror\_invs}\,\psi\,\vec{e}\,\sigma\,\tau$. The mirror assertion then asserts, for each thread π executing e in the configuration, the ownership of the translated thread $\pi \in \lfloor e \rfloor_\psi$. Next, the mirror assertion asserts the ownership of headers and points-to assertions for each allocated location, whose values are adequately translated. Then, the mirror assertion asserts ownership of live-location tokens for each allocated location that is not in the domain of the heap—that is, of 0-sized blocks. Finally, the mirror assertion asserts the ownership of the trace assertion $\mathsf{trace}\,\lfloor \theta \rfloor_\psi$, capturing the configuration's emitted trace, up to renaming by ψ.

We can then devote our attention to the completeness theorem of Mizzle. Formally speaking, this theorem states that, for a program e with no hard-coded locations, if there exists a configuration c reachable from the initial configuration $\mathsf{init}(e)$, then there exists a Mizzle derivation with a remaining proof obligation $\mathsf{mirror}\,c$.

Theorem 4.2 (Completeness of Mizzle). *Let e be an expression and c a configuration. If $\mathit{locs}(e) = \emptyset$, and if $\mathit{init}(e) \to^* c$ holds, then thread $\pi_0 \in e * \mathit{remaining}\,(\mathit{mirror}\,c) \vdash \mathfrak{goal}$ holds.*

We now explain how we use Theorem 4.2 to prove specialized corollaries. Intuitively, the assertion $\mathsf{mirror}\,c$ is the strongest assertion that can be proved about c in Mizzle. For each corollary, the target is hence to develop an intermediate lemma allowing us to weaken $\mathsf{mirror}\,c$ into the desired assertion. With such a lemma, we can then use RemainingMono to conclude.

To derive the completeness Corollary 2.3 for stuckness, we prove a lemma whose conclusion is $\mathsf{mirror}\,c \vdash \mathsf{witness\ Stuck}$. Its $\mathsf{Stuck}\,c$ hypothesis is exactly the hypothesis of Corollary 2.3, while the other hypotheses correspond to invariants of the reduction relation.

Lemma 4.3. *If (1) $\mathsf{Stuck}\,c$ holds, (2) every location mentioned in a thread has an associated header in the tag map, and (3) the locations in the domain of the tag map are present in the heap, then $\mathsf{mirror}\,c \vdash \mathsf{witness\ Stuck}$.*

To derive the completeness Corollary 2.4 for non-linearizability, we prove a lemma whose conclusion is $\mathsf{mirror}\,c \vdash \mathsf{witness\ NonLinearizable}$. Its hypotheses are exactly the ones from Corollary 2.4.

Lemma 4.4. *If the trace of c is non-linearizable, and if this trace contains no locations, then $\mathsf{mirror}\,c \vdash \mathsf{witness\ NonLinearizable}$.*

We can now explain the need for the trace to not include any locations. Indeed, recall that the trace assertion held by $\mathsf{mirror}\,c$ is $\mathsf{trace}\,\lfloor \theta \rfloor_\psi$, capturing the configuration's trace up to the renaming ψ. When the trace contains no locations, the renaming acts as the identity, so the trace held by $\mathsf{mirror}\,c$ is *equal* to the configuration's trace, and non-linearizability transfers directly.

As for a completeness corollary for races, which we do not state here for conciseness, we propose a lemma whose conclusion is $\mathsf{mirror}\,c \vdash \mathsf{witness\ Racy}$. The first hypothesis is exactly the

$$\mathfrak{goal} \triangleq \forall c.\ \mathsf{interp}\, c \twoheadrightarrow \Rrightarrow (\mathsf{interp}\, c * \mathsf{remaining}\, \top) \vee (\exists c'.\ \ulcorner c \rightarrow c' \urcorner * \mathsf{interp}\, c' * \mathfrak{goal})$$

Fig. 13. Definition of the $\mathfrak{goal}$ assertion

one witnessing Racy c, while the in-bounds requirement on the two accesses corresponds to an invariant of the reduction relation.

Lemma 4.5. *If c has two distinct threads $\pi_1 \neq \pi_2$ executing expressions e_1 and e_2 such that $\mathsf{LAcc}\, e_1\, \ell\, b_1$ and $\mathsf{LAcc}\, e_2\, \ell\, b_2$ hold for the same location ℓ with $b_1 \vee b_2$, and both accesses are in bounds, then $\mathsf{mirror}\, c \vdash$ witness Racy.*

4.5 Inside the Goal Assertion

The definition of $\mathfrak{goal}$ mirrors the proof of Theorem 4.1. Indeed, we define $\mathfrak{goal}$ as a *least fixpoint*, allowing us to perform reasoning by induction. Either the base case holds, and the user has fulfilled their remaining proof obligation, or there exists one possible step, after which we can apply the induction hypothesis. In the cases where we take a step, we need to reflect how that step updates the physical state of the system. Recall from Section 4.2 that the physical state and the logical state are related using the state interpretation predicate interp. Thus, as the program makes progress, one needs to update logical assertions accordingly. Such an update is possible thanks to the Iris *ghost update modality*, written $\Rrightarrow \varphi$, asserting that φ holds after such an update.

Figure 13 presents the definition of the $\mathfrak{goal}$ assertion, which is similar to the assertion of the same name developed by Moine et al. [2026], except for the base case. The definition universally quantifies over the configurations c, and requires ownership of the state interpretation predicate $\mathsf{interp}\, c$. Then, it allows for a ghost update thanks to the ghost update modality $\Rrightarrow$. Next, the definition consists of a disjunction of two cases. Either $\mathsf{interp}\, c$ holds untouched, together with the remaining $\top$ assertion, witnessing that the initial proof obligation has been fulfilled. This is the base, non-recursive case. Otherwise, the definition asserts the existence of a configuration c' reachable from c in one step. It also returns the ownership of the updated state interpretation predicate $\mathsf{interp}\, c'$, as well as the $\mathfrak{goal}$ assertion itself.

5 Case Studies

In this section, we (a) evaluate whether an LLM can in fact use Mizzle to prove the existence of a bug, and (b) assess the cost of constructing a formal proof that a bug exists. We focus on a practical, *empirical*, proof of concept of our approach. We first present our methodology (§5.1), and then two case studies: non-linearizability (§5.2) and memory races (§5.3). These case studies are not meant to be exhaustive, but rather to illustrate the ability of a particular LLM to use Mizzle in order to find a bug and prove its existence.

5.1 Methodology

The programs we consider in this section are all written in OCaml 5 and translated into ZooLang using the `ocaml2zoo` tool [Allain and Scherer 2026]. For non-linearizability bugs, we adapted implementations of concurrent data structures from the Saturn library [Saturn's contributors 2024] and manually inserted bugs. For memory races, we ported a number of examples from the *DataRaceBench* suite [Liao et al. 2017], which we describe in more detail in Section 5.3. We first confirmed that our mechanization had all of the lemmas and tactics needed to prove incorrectness of each test case by initially working interactively with LLMs to construct a *reference solution* for each of them. After confirming this way that the examples were in scope, our goal was to experimentally evaluate the cost of having an LLM prove incorrectness.

Based on our experience developing these initial solutions, we developed a *Mizzle proof guide* that we provided as a prompt to each agent. The guide summarizes how one uses Mizzle to prove incorrectness, the key lemmas the library provides, as well as general-purpose tactics. We also summarize the architecture of the project in a separate prompt file. As much as possible, we tried to keep the proof guide and tactics general, so that they can be reused for other case studies.

The tactic library provides one tactic per language primitive that applies the corresponding proof rule (*e.g.*, one tactic for executing a load operation, one for a store operation, ...). While we did initially provide tactics to execute multiple steps, we observed that the LLM systematically preferred to use per-primitive steps—perhaps because this gives more fine-grained control over how it reduces the program. A key observation we make is that an LLM is yet another user, who needs good tactic support and description. In our case, the proof guide is also augmented with a list of "gotchas", problems that we observed the model run into when we were interactively developing the Rocq formalization, and that we had to explicitly warn it about. These are mostly related to ZooLang, which sometimes has a surprising (for an LLM) semantics. For example, the semantics imposes a right-to-left evaluation order, and immutable data structures need one reduction step before being considered values.

In Sections 5.2 and 5.3, we report the results of running the models on each test case, with the machinery described above. More precisely, for a given test case, we delete its reference solution, and task the model with reading the proof guide and the architecture description, and then producing a proof of the designated top-level theorem, which it is forbidden from weakening or restating. In addition to deleting the reference solution, we also delete solutions to similar cases (*e.g.*, we delete stack-related examples when replaying the non-linearizability proof of a particular stack implementation). However, we allow the model to consult solutions of unrelated cases (*e.g.*, in the case of a stack-related problem, read solutions to queue-related problems). These serve as additional examples to help teach the agent how to use Mizzle. As part of the proof guide, the agent was instructed to review these examples.

For these case studies, we used *Claude Sonnet 5*. We disable all persistent memory and forbid access to the compiled artifacts and to the version-control history, so no prior proof state can leak into the run. The model interacts with Rocq through an MCP server [rocq-mcp's contributors 2025]. We asked the model to ensure that no axioms were used, which we then manually checked while confirming that the final proof state preserved the desired statement.

Limitations and Threats to Validity. Before discussing the results, let us first acknowledge several limitations of the experimental setup. This experiment does *not* evaluate how good LLMs are at finding unknown bugs, for several reasons: (1) *DataRaceBench* is widely used, so there may already have been leakage; (2) the non-linearizability bugs are relatively simple errors that we inserted, which may not be representative of real bugs; (3) in developing the reference solutions we already determined that the LLMs could find the bugs in all of our examples. As we stated in Section 1, the goal of this paper is *not* to assess or improve LLM bug finding. Rather, the purpose of these experiments is to assess the cost of constructing a formal proof that a bug exists.

Aside from this, these experiments have two important potential sources of bias. First, there is a *tuning* bias: we developed the proof guide and tactics interactively while working on these very case studies, so their success may partly reflect a fit to the benchmark rather than the general applicability of our approach. We tried to keep the guide and tactics as general as possible, but we did not evaluate the approach without the proof guide, nor on completely new case studies. Second, there is a *framing* bias: the model is told that the program is incorrect and asked to prove this, rather than to assess if the program is incorrect or not before proving. While we don't tell the agent where the bug is, this still potentailly reduces the overall cost compared to having to

ID	Structure	Bug	Input tokens	Output tokens	Time
001	Stack	Lost push (non-atomic store)	1.5M	14K	183s
002	Stack	Duplicated pop (non-atomic store)	1.6M	20K	269s
003	Stack	Clobbering push (fresh-read CAS)	2.0M	27K	335s
004	Stack	Lost push (discarded CAS)	1.7M	26K	292s
005	Stack	Spurious empty pop (no retry)	1.6M	19K	223s
006	Queue	Lost dequeue (no retry)	4.4M	32K	468s
007	Queue	Duplicated dequeue (blind store)	2.0M	15K	222s
008	Queue	Duplicated dequeue (fresh-read CAS)	3.6M	23K	340s
009	Queue	Lost enqueue (discarded CAS)	2.6M	31K	422s
010	Queue	Duplicated dequeue (stale peek)	4.7M	45K	627s
011	Hash Table	Lost add (blind store, add)	4.1M	42K	538s
012	Hash Table	Lost add (blind store, remove)	6.1M	46K	661s
013	Hash Table	Lost add (fresh-read CAS)	3.8M	33K	474s
014	Hash Table	Lost add (CAS-fail reports ok)	4.5M	28K	394s
015	Hash Table	Duplicated add (stale absence check)	3.6M	29K	434s

Fig. 14. Evaluation of Mizzle being used by Claude Sonnet 5 to prove non-linearizability. The minimum reported cost is $1.30, the maximum $3.50, and the average $2.30. The average cache hit rate is 95%. The average time is approximately 6 minutes.

determine first if the program has a bug or not. However, we envision that a possible workflow in practice might be to use one agent to first find bug candidates, and then use a second agent to fully determine the bug and construct the incorrectness proofs. In that case, this setup is closer to the scenario the second agent runs in.

5.2 Non-Linearizability

In Section 2.4, we explained how Mizzle can be used to prove that a concurrent data structure is non-linearizable. In this section, we test this approach on a set of case studies. They all correspond to standard concurrent data structures in which we manually inserted bugs: Treiber's stack [Treiber 1986], Michael and Scott's queue [Michael and Scott 1996], and a concurrent hash table [Saturn's contributors 2024]. We asked an agent to prove non-linearizability following our methodology (§5.1). Note that, for each data structure, we provide the sequential specification of the data structure the agent is expected to use in its proof, as explained in Section 2.4.

Figure 14 presents the results of our evaluation. Each line describes the class of data structure, the nature of the bug, and the results of the LLM run, along with statistics about token costs. Each case study is between 25 and 55 OCaml lines of code, and the model outputs between 94 and 189 lines of proof. The agent was able to prove non-linearizability in all cases, with an average token cost of $2.30. Across all cases, we measure that less than 10% of the time is spent in tool calls, and in particular in MCP calls (i.e., waiting for Rocq to process commands). Interestingly, each session shows that a pain point is the tactic tooling. Even with dedicated tactics and guidance, the agent can struggle: the rate of MCP calls that fail varies between 0% and 43.6%, with an average of 13.2%. In particular, the agent struggles to identify what is the next tactic to use (even if syntactically, a store operation appears, it may be needed to reduce its arguments to values first). We believe that with better prompting, improved error messages from tactics, and perhaps a better-suited formal language, the cost and time could be significantly reduced.

5.3 Memory Races

We now evaluate the ability of an LLM to prove the presence of a race (Section 4.2).

ID	Family	Bug	Input tokens	Output tokens	Time
009	Shared cell	Missing lastprivate	1.0M	7K	113s
011	Shared cell	Shared decrement	11.0M	62K	914s
016	Shared cell	Write-after-write	1.8M	13K	208s
018	Shared cell	Shared increment	1.7M	17K	225s
020	Shared cell	Missing private	1.5M	15K	213s
001	Loop-carried	Write-after-read	1.3M	9K	144s
003	Loop-carried	Write-after-read (2D loop)	4.1M	32K	429s
014	Loop-carried	Out-of-bounds	5.1M	39K	582s
015	Loop-carried	Out-of-bounds (var. length)	5.1M	31K	434s
024	Loop-carried	SIMD true dependence	2.6M	25K	350s
029	Loop-carried	Read-after-write	1.0M	9K	136s
005	Loop-carried (ind.)	Indirect indexing	8.1M	59K	811s
006	Loop-carried (ind.)	Indirect indexing	13.1M	98K	1323s

Fig. 15. Evaluation of Mizzle being used by Claude Sonnet 5 to prove the existence of a memory race. The minimum reported cost is $0.90, the maximum $7.10, and the average $2.70. The average cache hit rate is 95%. The average time is approximately 7 minutes.

Test cases. The test cases we use for this study are direct OCaml translations of some examples of the *DataRaceBench* (DRB) suite [Liao et al. 2017]. DRB examples are written in C/C++ and use the OpenMP library for parallelism (*e.g.*, a parallel for loop). We selected a subset of DRB examples that were easily translatable to OCaml, using the `domainslib` library [domainslib's contributors 2024] for lightweight parallelism in a similar style as OpenMP. We also selected examples that were *positive* (*i.e.*, they truly had a bug, some entries have no bugs). Moreover, DRB examples come with a "test", that is, a final assertion potentially failing due to a data race. We decided to remove this test, in order to not give the model any hint about where the race is located. (Our approach does not rely on this test to identify bugs, so it is safe to remove.) Each case study is between 4 and 18 OCaml lines of code, and the model outputs between 32 and 223 lines of proof.

The `domainslib` library is complex. In order to avoid stepping into the library using Mizzle, we axiomatize some primitives (an approach we detail in Section 7). In particular, we provide the following axiom for the parallel for loop:

$$\left(\forall \vec{\mu}\, \vec{\eta}.\ \left(\begin{array}{r}\ulcorner |\vec{\mu}| = |\vec{\eta}| = n \urcorner \ *\ (\ *_{i<n}\ \mathsf{thread}\ \vec{\mu}(i)\ \vec{\eta}(i)\ (e\ i))\ *\\ ((\ *_{i<n}\ \mathsf{thread}\ \vec{\mu}(i)\ \vec{\eta}(i)\ ())\ *\ (\mathsf{thread}\ \pi\ \vec{\kappa}\ () \mathrel{-\!\!*} \mathfrak{goal}) \mathrel{-\!\!*} \mathfrak{goal})\end{array}\right) \mathrel{-\!\!*} \mathfrak{goal}\right) \vdash \begin{array}{l}\mathsf{thread}\ \pi\ \vec{\kappa}\ (\mathsf{parallel\_for}\ n\ e)\ *\\ \mathfrak{goal}\end{array}$$

The axiom states that a thread π about to run parallel_for $n\ e$ under evaluation context $\vec{\kappa}$ may spawn n iteration threads $\vec{\mu}$, each running $e\ i$ with its own thread identifier $\vec{\mu}(i)$ and under its own evaluation context $\vec{\eta}(i)$. Once every iteration has been driven to $()$, the user regains the parent thread π, with parallel_for $n\ e$ reduced to $()$, and must re-establish $\mathfrak{goal}$.

Note that this specification does not hold as-is if the parallel for loop sequentially chunks some operations, as the `domainslib` library does. Yet, we argue that this specification is the one that the user should use for bug finding, as the size of chunks is usually a performance configuration option: the correctness of the program should *not* depend on this parameter having a particular value. This axiomatized specification allows us to search for bugs across all choices of this parameter.

Results. Figure 15 presents the results. We reuse the same identifier numbers as in the DRB repository, and classify the bugs into custom families. More precisely, we study 3 types of bugs. First, shared cell bugs, that is, parallel for loops accessing a shared cell without proper synchronization. Second, loop-carried bugs, that is, when a particular iteration i of a parallel for loop depends on

```
let main () =
  let a = Array.make 100 0 in
  let x = ref 10 in
  parallel_for 100
    (fun i -> a.(i) <- !x; x := i)
```

(a) **Shared cell** (016): concurrent reads and writes to the shared cell `x`

```
let main () =
  let n = 100 in
  let a = Array.make n 0 in
  for i = 0 to n - 1 do
    a.(i) <- i
  done;
  parallel_for (n - 1)
    (fun i -> a.(i+1) <- a.(i) + 1)
```

(b) **Loop-carried** (029): iteration `i` reads `a.(i)` written by iteration `i-1`

```
let n = 180
let main () =
  let index_set = Array.make n 0 in
  for g = 0 to 29 do
    let start = 521 + 338*g/6 + 26*(g mod 6) in
    for k = 0 to 5 do
      index_set.(6*g + k) <- start + 2*k
    done
  done;
  index_set.(5) <- 533;
  let base = Array.make (2013+12+1) 0 in
  for i = 521 to 2025 do base.(i) <- i done;
  parallel_for n (fun i ->
    let idx = index_set.(i) in
    base.(idx)    <- base.(idx)    + 1;
    base.(idx+12) <- base.(idx+12) + 3)
```

(c) **Loop-carried (ind.)** (006): the offset `idx` is computed at runtime, opening the door to collisions

Fig. 16. One representative from each bug family: a shared cell, a loop-carried dependence, and an indirect (runtime-computed) dependence

a write operation made *by another, concurrent iteration*, for example $i + 1$. Third, indirect bugs, which are similar to loop-carried bugs, but for which the offset is computed at runtime, potentially leading to complex calculations.

As in the non-linearizability case study, the cost is an average of a few dollars per example. We observe that the agent triggered a large number of Rocq MCP failures in some cases: we report a minimum of 4.5% MCP failures, a maximum of 70.0%, and an average of 31.6%. The 70.0% error rate is an outlier, the expensive case 011, which we comment on at the end of this section. Cases 014 and 015 are almost identical, except that the latter uses a variable to store the static length of the array. It is interesting to see that, even if the number of tokens used is roughly the same, there is some time difference. Below, we comment on three representative examples.

Representative examples. Figure 16 presents three examples of programs we verify to have a race. Figure 16a shows a shared-cell bug: while the ownership of the array `a` is correctly split for the parallel for loop, all iterations are accessing the same shared cell `x` for both reading and writing. By examining logs from the test run, we observed that the agent roughly spent half of its time understanding the program and surveying other DRB examples. Notably, it reports as a challenge that the race was on `x` and not on `a`. It then spent the other half of its time using Rocq interactively to prove the existence of the race. The agent reports some tactic friction (a lemma not to its taste for parallel for, closures that need one step, hence one tactic, before being considered values, ...).

Figure 16b shows a loop-carried bug: iterations of the parallel loop do have separate ownership, as iteration `i` writes to the `(i+1)`-th offset of the array `a`, while reading offset `i`, which is being concurrently written by iteration `i-1`. The agent follows the same methodology, first surveying different DRB families, and then committing to its proof strategy, this time significantly faster. Indeed, the agent recognized a for-loop pattern as in cases 016 and 018 (even if the race in these cases is on a shared cell). It reports a technical challenge about the fact that `100 - 1` (obtained after substituting `n` by `100`) is not a value and needs to step.

Figure 16c shows a more complex case: the program first allocates an array of offsets `index_set`, populates it with complex calculations, but with an injective function, next overwrites one of the offsets to create a collision, and then allocates an array `base`, before running a parallel for loop,

accessing offsets in `base` depending on the values in `index_set`. Again, the agent first surveyed other examples and tactics, spending around 35% of its time, and then drove the proof. It reports the conceptual challenge of working out the arithmetic to understand which offset had a race: iteration 0 (where `idx=521`) and iteration 5 (where `idx=533`) both touch the same offset `533=521+12`. The agent also reports multiple Rocq friction points, from tactics with bad error messages to having to deal with numbers both in $\mathbb{N}$ and $\mathbb{Z}$.

The outlier. Let us comment on the outlier, case 011, since it spent most of its time on Rocq-related problems. The code is as follows:

```
let main () =
  let len = 100 in
  let num_nodes = len in
  let num_nodes2 = ref 0 in
  let x = Array.make len 0 in
  for i = 0 to len - 1 do if i mod 2 = 0 then x.(i) <- 5 else x.(i) <- -5 done;
  Drb.parallel_for num_nodes (fun i -> if x.(i) <= 0 then num_nodes2 := !num_nodes2 - 1)
```

The race occurs because multiple offsets of `x` store negative values, passing the test `x.(i) <= 0` and hence concurrently decrementing `num_nodes2`. The challenge for the agent comes from the test `i mod 2 = 0`. This equality corresponds to OCaml's *structural* equality, which recurses through immutable data structures, as opposed to physical equality (Section 2.2). In Zoo, structural equality is implemented as explicit code, which needs verification. We had no generic specification for this operation—the agent thus spent time, mid-proof, unfolding the definition and struggled to prove it. In prior runs during our iterative development process, the agent had automatically synthesized and proved a lemma for this. Adding a dedicated lemma for this operation drove the cost to 3.8M input tokens, costing around $2.50, whereas it was $7.10 before.

6 Related Work

Under-approximate logics. The idea of an under-approximate logic, that is, a logic whose triples describe a *subset* of the reachable behaviors rather than an over-approximating superset, is generally credited to de Vries and Koutavas [2011], who introduce *reverse Hoare logic* to reason about reachability in nondeterministic programs. O'Hearn [2020] developed *Incorrectness Logic* (IL), observing that under-approximation is exactly what is needed to justify a bug-finding analysis that reports only true positives: an IL triple [*pre*] *c* [*post*] guarantees that every state in *post* is reachable from some state in *pre*. Raad et al. [2020] first brought the power of separation logic to IL and proposed *Incorrectness Separation Logic* (ISL), recovering the local, frame-based reasoning that makes separation logic scalable while soundly detecting memory-safety bugs.

Raad et al. [2022] were the first to extend under-approximate reasoning to concurrency, with *Concurrent Incorrectness Separation Logic* (CISL), and Raad et al. [2023] improved it by proposing a *Concurrent Adversarial Separation Logic* (CASL), the first general framework for under-approximate reasoning about concurrent programs that is parametric in the underlying notion of erroneous behavior. CISL introduces a useful taxonomy of concurrency bugs, split into three categories: local (that is, bugs that manifest without concurrency), global and data-agnostic (bugs manifesting due to concurrency, but not because of the exchange of information between threads), and data-dependent. CISL targets the first two categories only, while CASL also supports the third category, thanks to reasoning rules inspired by rely-guarantee reasoning [Jones 1983]. As our examples show, Mizzle supports all three categories. In particular, because Mizzle takes a `goal`-oriented approach, we have no need for rely-guarantee reasoning in the case of data-dependent bugs. Compared to CISL/CASL, Mizzle is OCaml-oriented and supports higher-order functions, but has no notion of deallocation, a

notion important for CISL/CASL as they consider a language with potential use-after-free/double-free bugs. We posit that Mizzle can be extended to support deallocation. Concerning case studies, CISL supports reasoning about data races using an encoding via traces, which we do not need. CASL allows for proving security bugs, by encoding an adversarial client as a thread running in parallel with the client—we would like to investigate how this encoding scales in Mizzle. To our knowledge, Mizzle is the first under-approximate logic for concurrency that is proved complete.

Under-approximate logics exist outside of the world of incorrectness. For example, Angelic [Moine et al. 2026] is a logic allowing for proving that *one* interleaving is safe and terminates—such a property is used in combination with another property asserting that the verification of a single interleaving suffices to guarantee that they are all correct. Angelic comes with a WP-like proof mode with a run modality targeting a single thread at a time, encoded on top of a "scheduler" mode close to our `goal`-oriented approach. Some logics try to combine under-approximate reasoning with over-approximate reasoning. In particular, Outcome Logic [Zilberstein 2025] offers a framework where both demonic choice (for over-approximation) and angelic choice (for under-approximation) can be expressed. Exact separation logic [Maksimović et al. 2023] is another variant of the combination of under- and over-approximate reasoning, this time by forcing that the set of states described by the postcondition is exactly the set of states reachable from the precondition.

Automated static analyses for finding true positives. Many automated tools are designed to avoid false positives, and the incorrectness logics described above have been used to justify that these analyses have this property. For example, the Pulse analysis shipped in Infer [Calcagno et al. 2015] is grounded in ISL. On the data race side, the no-false-positives property of RacerD [Blackshear et al. 2018] was proven using CISL. Other under-approximate reasoning is also used in a deadlock detector [Brotherston et al. 2021]; it would be interesting to extend Mizzle's approach to deadlock detection. More compositional symbolic execution engines such as Gillian [Fragoso Santos et al. 2020; Ayoun 2025] and Soteria [Ayoun et al. 2026], also allow for under-approximate reasoning about programs written in a variety of languages. Recently, Bayer et al. [2026] make use of Soteria to train models on traces and improve accuracy on verification tasks.

Non-linearizability and its detection. A number of tools exist for automatically proving linearizability [Vafeiadis 2010; Zhu et al. 2015; Meyer et al. 2023] or checking for non-linearizability [Burckhardt et al. 2010; Wing and Gong 1993; Lowe 2017; Horn and Kroening 2015; Emmi and Enea 2018]. Although checking whether a trace is non-linearizable is NP-hard, these tools develop approaches that scale well to realistic use cases. Most non-linearizability checkers are designed to avoid false alarms, but their ability to detect bugs is bounded by the traces or test depths they explore.

LLMs writing machine-checked proofs. There is a vast and growing literature on using LLMs to construct machine-checked proofs with proof assistants [Hubert et al. 2025; Lin et al. 2025; Ren et al. 2025] or to produce annotations for auto-active deductive program verifiers [Aggarwal et al. 2025; Yang et al. 2025; Mugnier et al. 2025; Banerjee et al. 2026]. To the best of our knowledge, our work is the first to propose using LLMs to construct proofs of *incorrectness* of programs.

7 Discussion and Future Work

We presented Mizzle, an incorrectness separation logic proved sound and complete for a subset of OCaml 5 programs, mechanized in Rocq using the Iris separation logic framework.

For future work, we would like to extend the proof of concept we present in this paper to a full-fledged tool. In particular, we plan on improving tactics support in order to simplify the proof effort for agents. We would also like to develop a more extensive benchmark and test suite in order to evaluate the scalability of our approach, for example by targeting open source OCaml code.

Besides these improvements, we would like to explore additional scenarios where Mizzle presents some advantages compared to testing. In particular, because it is a program logic, users can give axiomatic specifications to functions or library components that they do not want to focus on, and assume this specification holds while focusing on a specific, potentially buggy, client. We plan on studying further at least three cases, as we elaborate on below: (1) when the function is under-specified, (2) when the function is too expensive or difficult to run during a test, (3) when the function is already verified functionally correct.

Under-specified functions. As we explained in Section 5.3, some functions are under-specified, in order to leave room for runtime optimizations or future changes. For example, the chunking parameter in Section 5.3 is a tunable option that could change in the future. Some programs may not be buggy with the current chunking value, but may become buggy if the default value changes. But when we test against a particular concrete implementation with a particular current choice of chunking, then testing will only cover behaviors that could be generated by the current implementation. While it is easy to alter the chunking parameter in this case, there are other kinds of under-specification that are harder to test. For example, a particular implementation of a concurrent data structure may generate fewer interleavings (because of some stronger internal ordering) than linearizability permits. If the implementation of that data structure changes, even to another linearizable version, bugs may suddenly be triggered.

Because Mizzle is compositional, the user can assume a specification for the under-specified function, and use that instead in order to find a bug. Of course, as with any axiomatization, this requires care. An "invalid" axiom in our setting can have the effect of hiding actual bugs (by ruling out realizable behaviors) or cause false alarms (by accidentally adding unrealizable behaviors).

Scenarios that are expensive or difficult to test. Other cases warrant the use of axioms in Mizzle. In many situations, programs may involve interactions that are costly to execute or that are difficult to run while testing. To deal with this, a common practice is to use *mock* functions that simulate a component or aspect of the environment. For example, when testing a program that involves network interactions, it may be too slow or costly to actually communicate over the network, so these interactions are often replaced by code that returns simulated responses.

The limitation is that these mock functions may not actually exhibit the full space of behaviors that the real environment can generate. As a result, testing with mocked environments can miss bugs. With Mizzle, we can instead axiomatize specifications for functions that interact with the environment, where these axioms can capture the full set of allowable behaviors.

Reusing a correctness proof. It is possible to connect a standard *correctness* logic specification for a function to Mizzle. In particular, if we have a weakest precondition specification for a program e of the form $\varphi \twoheadrightarrow \mathsf{wp}\, e\, \psi$, where φ is the precondition and ψ is the postcondition, then this guarantees that, starting from a state satisfying φ, *every* interleaving of e is safe (*i.e.*, has no bug), and if it terminates, it reaches a state satisfying ψ. Hence, if e terminates, this guarantees that *at least one* interleaving of e can reach a state satisfying ψ from a state satisfying φ. Thus, intuitively, if we use a *total* weakest precondition, which implies termination, this should imply a Mizzle specification. We proved a version of this as follows, for a custom definition of a total weakest precondition:

Lemma 7.1 (WP Entails Goal). *The following entailment holds:*

$$(\varphi \twoheadrightarrow \mathit{wp}\, e\, \psi) * \varphi * \mathit{thread}\, \pi\, \vec{\kappa}\, e * (\forall v.\ \psi\, v \twoheadrightarrow \mathit{thread}\, \pi\, \vec{\kappa}\, v \twoheadrightarrow \mathfrak{goal}) \vdash \mathfrak{goal}$$

This says that if the user has a WP specification for e, and if some thread π is actually executing e under evaluation context $\vec{\kappa}$, and if the precondition φ holds, then the user can use the WP specification to advance thread π, under evaluation context $\vec{\kappa}$, to a value v satisfying $\psi\, v$.

The above lemma gives us a way in which the use of Mizzle can be intertwined with formal verification: if a program is verified, then we can reuse its specification in Mizzle to find bugs in clients being developed.

References

Pranjal Aggarwal, Bryan Parno, and Sean Welleck. 2025. AlphaVerus: Bootstrapping Formally Verified Code Generation through Self-Improving Translation and Treefinement. In *Proceedings of the 42nd International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 267)*. PMLR, 587–615.

Clément Allain and Gabriel Scherer. 2026. Zoo: A framework for the verification of concurrent OCaml 5 programs using separation logic. *Proceedings of the ACM on Programming Languages* 10, POPL (Jan. 2026). doi:10.1145/3776701

Anon. 2026. Supplementary Material. Submitted together with this paper.

Sacha-Élie Ayoun, Opale Sjöstedt, and Azalea Raad. 2026. Soteria: Efficient Symbolic Execution as a Functional Library: Perhaps You Should Write Your Own Symbolic Execution Engine! *Proc. ACM Program. Lang.* 10, PLDI, Article 228 (June 2026), 26 pages. doi:10.1145/3808306

Sacha-Élie Ayoun. 2025. *Gillian: Foundations, Implementation, and Applications of Compositional Symbolic Execution.* Ph. D. Dissertation. Imperial College London. doi:10.25560/119340

Debangshu Banerjee, Olivier Bouissou, and Stefan Zetzsche. 2026. DafnyPro: LLM-Assisted Automated Verification for Dafny Programs. arXiv:2601.05385 [cs.SE] https://arxiv.org/abs/2601.05385

Jonas Bayer, Stefan Zetzsche, Olivier Bouissou, Remi Delmas, Michael Tautschnig, and Soonho Kong. 2026. Teaching LLMs Program Semantics via Symbolic Execution Traces. arXiv:2605.06184 [cs.SE] https://arxiv.org/abs/2605.06184

Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: compositional static race detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (Oct. 2018), 28 pages. doi:10.1145/3276514

James Brotherston, Paul Brunet, Nikos Gorogiannis, and Max Kanovich. 2021. A Compositional Deadlock Detector for Android Java. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 955–966. doi:10.1109/ASE51524.2021.9678572

Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: a complete and automatic linearizability checker. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. ACM, 330–340. doi:10.1145/1806596.1806634

Cristiano Calcagno, Dino Distefano, and Peter O'Hearn. 2015. Open-sourcing Facebook Infer: Identify bugs before you ship. https://code.facebook.com/posts/1648953042007882/open-sourcing-facebook-infer-identify-bugs-before-you-ship/.

Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Software Engineering and Formal Methods*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–171.

domainslib's contributors. 2024. domainslib: Parallel Programming over Domains for OCaml 5. https://github.com/ocaml-multicore/domainslib. Accessed: 2026-07-05.

Brijesh Dongol and John Derrick. 2015. Verifying Linearisability: A Comparative Survey. *ACM Comput. Surv.* 48, 2 (2015), 19:1–19:43. doi:10.1145/2796550

Michael Emmi and Constantin Enea. 2018. Sound, complete, and tractable linearizability monitoring for concurrent collections. *Proc. ACM Program. Lang.* 2, POPL (2018), 25:1–25:27. doi:10.1145/3158113

Mingdong Feng and Charles E. Leiserson. 1997. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures* (Newport, Rhode Island, USA) *(SPAA '97)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/258492.258493

José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, part i: a multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 927–942. doi:10.1145/3385412.3386014

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

Alex Horn and Daniel Kroening. 2015. Faster Linearizability Checking via P-Compositionality. In *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9039)*. Springer, 50–65. doi:10.1007/978-3-319-19195-9_4

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz, Vivek Veeriah, Lei Yu, Oliver Nash, Paul Lezeau, Salvatore Mercuri, Calle Sönne, Bhavik Mehta, Alex Davies, Daniel Zheng, Fabian Pedregosa, Yin Li, Ingrid von Glehn, Mark Rowland, Samuel Albanie, Ameya Velingker, Simon Schmitt, Edward Lockhart, Edward Hughes, Henryk Michalewski, Nicolas Sonnerat, Demis Hassabis, Pushmeet Kohli, and David Silver. 2025. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature* 651, 8106 (2025), 607–613. doi:10.1038/s41586-025-09833-y

C. B. Jones. 1983. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (Oct. 1983), 596–619. doi:10.1145/69575.69577

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf

Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–27. doi:10.1145/3527325

Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 11, 14 pages. doi:10.1145/3126908.3126958

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. 2025. Goedel-Prover-V2: Scaling Formal Theorem Proving with Scaffolded Data Synthesis and Self-Correction. arXiv:2508.03613 [cs.LG] https://arxiv.org/abs/2508.03613

Gavin Lowe. 2017. Testing for linearizability. *Concurr. Comput. Pract. Exp.* 29, 4 (2017). doi:10.1002/CPE.3928

Petar Maksimović, Caroline Cronjäger, Andreas Lööw, Julian Sutherland, and Philippa Gardner. 2023. Exact Separation Logic: Towards Bridging the Gap Between Verification and Bug-Finding. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 19:1–19:27. doi:10.4230/LIPIcs.ECOOP.2023.19

Roland Meyer, Anton Opaterny, Thomas Wies, and Sebastian Wolff. 2023. nekton: A Linearizability Proof Checker. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13964)*, Constantin Enea and Akash Lal (Eds.). Springer, 170–183. doi:10.1007/978-3-031-37706-8_9

Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (Philadelphia, Pennsylvania, USA) *(PODC '96)*. Association for Computing Machinery, New York, NY, USA, 267–275. doi:10.1145/248052.248106

Alexandre Moine, Sam Westrick, and Joseph Tassarotti. 2026. All for One and One for All: Program Logics for Exploiting Internal Determinism in Parallel Programs. *Proc. ACM Program. Lang.* 10, POPL, Article 26 (Jan. 2026), 29 pages. doi:10.1145/3776668

Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Yuanyuan Zhou. 2025. Laurel: Unblocking Automated Verification with Large Language Models. *Proc. ACM Program. Lang.* 9, OOPSLA1 (2025), 1519–1545. doi:10.1145/3720499

Peter W. O'Hearn. 2020. Incorrectness logic. In *Proc. ACM Program. Lang. (POPL)*, Vol. 4. Article 10, 32 pages. doi:10.1145/3371078

Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 12225)*. Springer, 225–252. https://plv.mpi-sws.org/ISL/

Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Concurrent incorrectness separation logic. *Proc. ACM Program. Lang.* 6, POPL, Article 34 (Jan. 2022), 29 pages. doi:10.1145/3498695

Azalea Raad, Julien Vanegue, Josh Berdine, and Peter O'Hearn. 2023. A General Approach to Under-Approximate Reasoning About Concurrent Programs. In *34th International Conference on Concurrency Theory (CONCUR 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 279)*, Guillermo A. Pérez and Jean-François Raskin (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:17. doi:10.4230/LIPIcs.CONCUR.2023.25

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. 2025. DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition. arXiv:2504.21801 [cs.CL] https://arxiv.org/abs/2504.21801

rocq-mcp's contributors. 2025. rocq-mcp: An MCP Server for the Rocq Prover. https://github.com/LLM4Rocq/rocq-mcp. Accessed: 2026-07-05.

Saturn's contributors. 2024. Saturn: A collection of parallelism-safe data structures for OCaml 5. https://github.com/ocaml-multicore/saturn. Accessed: 2026-07-06.

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 113:1–113:30. https://doi.org/10.1145/3408995

Daniel Stenberg. 2026. The end of the curl bug-bounty. https://daniel.haxx.se/blog/2026/01/26/the-end-of-the-curl-bug-bounty/.

R. Kent Treiber. 1986. *Systems programming: Coping with parallelism.* Research Report RJ 5118. IBM Thomas J. Watson Research Center. https://dominoweb.draco.res.ibm.com/reports/rj5118.pdf

Viktor Vafeiadis. 2010. Automatically Proving Linearizability. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6174)*. Springer, 450–464. doi:10.1007/978-3-642-14295-6_40

Jeannette M. Wing and Chun Gong. 1993. Testing and Verifying Concurrent Objects. *J. Parallel Distributed Comput.* 17, 1-2 (1993), 164–182. doi:10.1006/JPDC.1993.1015

Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. 2025. AutoVerus: Automated Proof Generation for Rust Code. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 3454–3482. doi:10.1145/3763174

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2015. Poling: SMT Aided Linearizability Proofs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9207)*. Springer, 3–19. doi:10.1007/978-3-319-21668-3_1

Noam Zilberstein. 2025. Outcome Logic: A Unified Approach to the Metatheory of Program Logics with Branching Effects. *ACM Trans. Program. Lang. Syst.* 47, 3, Article 14 (Sept. 2025), 71 pages. doi:10.1145/3743131