Lazy Proof Automation for Separation Logic

Valentin Mikhalchuk
School of Computing, National University of Singapore, Singapore

Vladimir Gladshtein
School of Computing, National University of Singapore, Singapore

Ilya Sergey
School of Computing, National University of Singapore, Singapore

Abstract

Separation Logic is an established formalism for deductive verification of heap-manipulating programs. Proofs of symbolic heap entailment, an analogue of the ordinary logical implication, are amongst the most common reasoning steps in Separation Logic, and many existing heap verifiers provide automation for discharging valid heap entailments.

We observe that existing techniques for automating entailment proofs in foundational Separation Logic verifiers embedded into provers such as Rocq, suffer from three main drawbacks: (a) poor performance due to metaprogramming overhead, (b) limited expressivity, and (c) restricted extensibility. To address these shortcomings, we propose *lazy proof automation*—an approach to entailment proofs inspired by translation validation. Our key idea is to implement an entailment checker as a combination of (1) an efficient but unverified prover, suitable for fast-paced interactive proofs, and (2) a proof reconstruction procedure that takes the prover's trace and produces a certificate of entailment validity that can be checked a posteriori. We implemented these ideas in Yolo—a generic and extensible heap entailment prover built in Lean. We instantiate Yolo for two Lean-embedded Separation Logics and show its practical benefits, both in terms of user experience and proof-checking speed, compared with the automation available in state-of-the-art foundational Separation Logics.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases Lean, proof engineering, meta-programming

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.7

Supplementary Material *Software (Source Code)*: `https://zenodo.org/records/20366499`

Funding This work was supported by Stellar Development Foundation Academic Research Grant.

Acknowledgements We thank the ITP'26 reviewers for their insightful comments. We are also grateful to Arthur Charguéraud for hinting the idea of delayed automation in CFML.

1 Introduction

Separation Logic (SL) is a well-adopted methodology to specify and verify the behaviour of state-manipulating programs. Many frameworks implement *foundational* embeddings of SL-based verifiers into general-purpose theorem provers (*e.g.*, Rocq, Isabelle/HOL), reducing the trusted code base to that of the prover. A central notion in such verifiers is *heap entailment*: $P \vdash Q$ states that any heap satisfying P also satisfies Q. Heap entailment underpins standard SL rules (*e.g.*, the rule of consequence) and the encoding of Hoare triples via weakest preconditions, $P \vdash wp\ C\ Q$ [4], reducing program verification to proving entailment validity [1, 16, 21]. Because entailment proofs dominate verification conditions, their automation is crucial: poor performance or limited expressivity directly degrades the user experience. In this work, we tackle the challenge of implementing automation for Separation Logic entailment proofs that is both efficient and extensible.

A common automation approach is to develop a specialised tactic that uses the prover's metaprogramming facilities to discharge entailments by applying lemma-justified rewrites (algebraic laws, cancellation rules, *etc.*) [11, 26]. This approach is simple and extensible—one adds new connectives by providing more verified rewrites—but suffers from poor performance, as it relies on the *interpreter* of the prover's meta-language (*e.g.*, Ltac in the case of Rocq used as a meta-verifier) rather than its computational evaluator (*e.g.*, Rocq's Gallina).

An alternative approach is to use *computational reflection*: (1) reify the entailment goal into a syntactic term, (2) run a verified decision procedure in the prover's computational language, and (3) apply a soundness lemma relating the result back to the original goal [20]. Reflection is much more efficient, but its reliance on deep embeddings limits both expressiveness and extensibility. For example, such approaches struggle to support specifications with terms that have polymorphic and value-dependent types [20]. They are also hard to extend: for instance, existing approaches support only predicates over non-heap-dependent expressions, and adding the magic wand would require changing the entire embedding.

We pose the question: can one combine the expressiveness and extensibility of tactic-based proofs with the efficiency of reflection? As we demonstrate, the answer is yes.

Inspired by *translation validation* in certified compilation [22], we decouple entailment simplification into two phases: (1) an efficient *unverified* simplifier for fast interactive use, and (2) a proof reconstruction procedure that consumes the simplifier's trace and produces a machine-checkable proof. Because the complex task of proof reconstruction can be *delayed* to a separate stage, run independently and in parallel with other proof activities, we call our approach *lazy proof automation*. We implement lazy proof automation in Yolo, a Lean tactic that replicates proof automation (xsimpl) of CFML Separation Logic [4], yet provides substantially better interactive experience and efficiency, thanks to a *translation-validation* approach, improving on its Rocq predecessor in the following four aspects:[1]

1. *Usability*: To reduce interactive latency, Yolo separates the fast (unverified) simplification phase from the later proof reconstruction. Compared to CFML/xsimpl, this avoids repeated generation and checking of proof terms during exploratory proof development.
2. *Expressivity*: Unlike frameworks for proof by reflection, such as Bedrock's MirrorShard [20], Yolo uses a mixed deep/shallow representation so users can write rich Lean terms (including those with *dependent types*) in SL assertions while still getting efficient simplification.
3. *Extensibility*: Yolo is implemented in the style of *Data Types à la Carte* [27], using Lean's type class machinery. This makes it modular: users can add new operators by instantiating interfaces without modifying core code. Moreover, Yolo works with *any* Separation Logic that implements axioms of the MoSeL verification framework [15].
4. *Performance*: We show that Yolo outperforms existing foundational tactics: it is on average faster than xsimpl, Bedrock (up to 1.5× speedup), HTT [21], CFML [4], and VST [1]. Furthermore, the performance of Yolo's unverified mode (without proof reconstruction) is comparable to that of Pulse [8], a state-of-the-art non-foundational SL-based verifier.

Contributions and Outline.

- *Lazy proof automation*—an approach for decoupling proofs in a foundational verifier into the fast unverified automation phase, followed by certificate reconstruction (Sec. 2).
- Yolo—a generic Lean library for lazy proof automation of SL heap entailments (Sec. 3).

[1] Yolo is available at `https://github.com/verse-lab/yolo`

$$\frac{\{H\}\,[t]\,\{Q\}}{\{H * H'\}\,[t]\,\{Q * H'\}}$$

(a) The Frame rule

$$\frac{\{H_1\}\,[t]\,\{Q_1\} \qquad H \vdash H_1 * H_2 \qquad Q_1 * H_2 \vdash Q}{\{H\}\,[t]\,\{Q\}}$$

(b) The consequence-frame rule

Figure 1 Separation Logic rules used by xapp for proof automation

- Instantiating Yolo to support two Lean SL implementations: LGTM [10] and Iris [12] (Sec. 4).
- A report on evaluating the performance of Yolo, comparing it to existing foundational SL verifiers, as well as state-of-the-art non-foundational frameworks (Sec. 5).

2 Overview

This section provides an overview of the key concepts and design principles behind Yolo. We begin by reviewing the proof automation tactic xsimpl for heap entailment in the Separation Logic framework CFML (Sec. 2.1). We then demonstrate how Yolo improves upon xsimpl in terms of both usability and performance. Finally, we discuss how Yolo can be extended to work with different Separation Logics and custom operators (Sec. 2.2).

2.1 Starting Point: Verifying Heap-Manipulating Programs in CFML

CFML (Characteristic Formulae for ML) [4] is a foundational framework for interactively verifying functional correctness of programs in an ML-style language in Rocq using Separation Logic. Consider the following procedure, written in OCaml, which takes a pointer as an argument and doubles the value stored at that location:

```
let double = fun p → let n = !p in let m = n + n in p := m
```

One can specify the behaviour of `double` using the following SL triple:

$$\{p \mapsto n\}\,[\texttt{double p}]\,\{p \mapsto 2 * n\}$$

This triple states that if the pointer `p` initially stores the value n, then after executing `double p`, the value stored at `p` will be $2 * n$. In CFML, this can be expressed as follows:

```
Lemma triple_double : ∀ (p:loc) (n:int),
  triple <{ double p }> (p ⟶ n) (fun _ ⇒ (p ⟶ (2 * n))).
```

A CFML proof follows the program structure step-by-step, resembling symbolic execution:

```
Proof.
  xwp.     (* initialization *)
  xapp.    (* read p *)
  xapp.    (* addition operation *)
  xapp.    (* update p *)
  xsimpl.  (* p ⟶ n + n implies p ⟶ 2 * n *)
Qed.
```

Each primitive operation (reading `p`, writing to `p`, *etc.*) is handled by the xapp tactic. Using xsimpl at the end discharges the remaining heap entailment, proving the postcondition. In addition, xsimpl is also implicitly invoked by each xapp call, as we will explain next.

As discussed in Sec. 1, there are two common approaches to automating heap entailment proofs: *computational reflection*, which is efficient but hard to extend, and *tactic-based* automation, which is extensible but slower. CFML follows the latter approach: its xapp tactic performs backward symbolic execution using the consequence-frame rule (Fig. 1b), which combines the standard Frame Rule (Fig. 1a) with the rule of consequence. Its first premise concerns a command on a small heap fragment; the remaining two require proving heap entailments, which xapp delegates to xsimpl, which simplifies entailments by extracting pure facts and existential quantifiers, and cancelling matching predicates on both sides.

Crucially, xsimpl is *foundational*: it is implemented as a metaprogramming procedure in Ltac that applies a sequence of Separation Logic lemmas, generating a proof term that the proof assistant checks on every invocation. Since xsimpl is called implicitly by each xapp, the overhead accumulates: the user must wait for each tactic to finish before proceeding, even during exploratory proof development. Our approach and the tactic implementing it, both dubbed Yolo, address this overhead via *translation validation*: it decouples fast (unverified) simplification of proof obligations from proof reconstruction, which can be delayed until the proof is complete. In practice, one controls the mode via a global option:

```
set_option hsimp.yolo [true|false]
```

When set to `true`, Yolo performs simplification without generating proof terms; otherwise, it reconstructs Lean proof terms. This separation is the key to Yolo's performance (Sec. 5).

2.2 Generalising Beyond CFML

Yolo is designed to be extensible at three levels: the underlying Separation Logic, the set of supported operators, and proof reconstruction for new operators.

2.2.1 Plugging in a Different Separation Logic.

Yolo works with any Separation Logic that implements the MoSeL interface [15], provided by the Iris-Lean library [14] in Lean. MoSeL is an abstraction layer that captures the common algebraic structure shared by many Separation Logics—from classical heap-based logics to more advanced frameworks such as Iris [12]—via a hierarchy of type classes. To use Yolo with a new logic, one instantiates two of these type classes: `BIBase`, which defines the core connectives (entailment, separating conjunction, empty heap, *etc.*), and `BI`, which supplies proofs of the required algebraic laws (transitivity of entailment, introduction and elimination rules for conjunction, *etc.*). For example, a classical heap-based Separation Logic in the style of CFML defines separating conjunction as the union of two disjoint sub-heaps and proves the corresponding laws by unfolding these definitions. By contrast, Iris defines its connectives over step-indexed resource algebras, where separating conjunction corresponds to composition of resources rather than disjoint union of heaps—yet the same MoSeL interface applies. Once these instances are provided, Yolo works out of the box. We demonstrate this in Sec. 4, where we instantiate Yolo for two different Separation Logics.

2.2.2 Adding New Operators.

Yolo supports the standard SL operators: separating conjunction ($*$), magic wand ($-\!\!*$), pure assertions, empty heap, existential and universal quantification. If it encounters an operator it does not recognise (*e.g.*, a user-defined modality or a custom points-to predicate for a particular structure), it leaves it unchanged. To add a new operator (Fig. 2), a user must:

```
-- (a) Define the operator type
structure TNew where
  arg₁ : Expr
  arg₂ : Name

-- (b) Simplification behaviour
instance : Eval TNew γ where
  evaluateL x h := ...
  evaluateR x h := ...
  evaluateLR x h := ...
  evaluateWand x h₁ h₂ := ...
```

```
-- (c) Equality checking
instance : TIsDefEq TNew γ where
  isDefEq x y eq :=
    match matchExpr y with
    | some (TNew.mk a b) => ...
    | _ => return false
-- HasHprop is omitted

-- (d) Register the operator
elab "hsimp" : tactic =>
  hMain (TStar ⊕ ... ⊕ TNew)
```

Figure 2 Code fragments for adding a new operator to Yolo

1. *Define a type* representing the operator and its non-SL arguments (a). For example, an existential quantifier carries a bound variable name and a body expression.
2. *Specify simplification behaviour* (b) by instantiating the `Eval` type class, whose methods handle the operator on the left-hand side, right-hand side, or both sides of an entailment, as well as within a magic wand.
3. *Provide helper instances* (c) for equality checking (`TIsDefEq`) and for translating Lean expressions into Yolo's internal representation (`HasHprop`). The equality check is used during cancellation: when the same operator appears on both sides of an entailment, Yolo attempts to match their arguments and eliminate the common subexpression.
4. *Register the operator* (d) in the tactic's operator list, which is a type-level sum in the style of Data Types à la Carte [27], so that Yolo recognises its simplification laws.

After these steps, Yolo can simplify entailments with the new operator in the unverified mode.

2.2.3 Proof Reconstruction for New Operators

The steps above suffice for fast unverified simplification. To additionally support proof reconstruction, the user has to prove a soundness lemma that justifies each simplification step for the new operator and registers it in the corresponding handler. For instance, if the new operator can be eliminated from the left-hand side, the user proves a lemma stating that the entailment with the operator reduces to one without it, and adds an `apply` of this lemma to the handler's proof script. This cleanly separates the simplification logic from its justification: one can develop and test the simplification first, then supply proofs incrementally.

2.3 Putting It All Together

Consider again the `double` example from Sec. 2.1. When proving this specification using Yolo in Lean, the user begins symbolic execution with `xwp` and steps through each operation with `xapp`. With `hsimp.yolo` set to `true`, Yolo operates in fast unverified mode: it simplifies goals immediately without generating proof terms. Once the proof structure is complete, the user sets `hsimp.yolo` to `false` and rechecks the file; Yolo then reconstructs proof terms for Lean to verify. This validation can happen in the background or during continuous integration, without blocking interactive development.

3 Implementation

We now outline the core procedures of Yolo. Since we build on CFML's xsimpl, we first provide an overview of the underlying simplification algorithm. We then discuss the translation-validation approach, and finally its generalisation via the Data Types à la Carte technique.

3.1 The xsimpl Algorithm

The xsimpl simplification process comprises four steps: initialisation, left-hand side simplification, right-hand side simplification, and simplification of both sides. To illustrate the main ideas, we focus on initialisation and left-hand side simplification only, as the remaining steps follow the same pattern. Consider the following example:

$$\mathsf{emp} * (H_1 \mathrel{-\!\!*} (H_2 * H_3)) * H_1 \vdash H_2 * H_3,$$

where H_1, H_2, H_3 are some heap predicates. A quick look at the statement makes it obvious that it is valid: emp can be stripped off the left-hand side, while H_1 provides exactly the premise of the magic wand, yielding the disjoint union of heaps satisfying H_2 and H_3—precisely the entailment's right-hand side.

Initialisation.

The procedure begins by applying the `xsimpl_start` lemma, which partitions both sides of the entailment into three components each, joined by separating conjunction:

```
Lemma xsimpl_start : ∀ H1 H2,
 Xsimpl (emp, emp, (H1 ∗ emp)) (emp, emp, (H2 ∗ emp)) → H1 ⊢ H2.
```

The six components are: Hla – processed items from the left, Hlw – magic wand items from the left, Hlt – remaining left items, Hra – processed items from the right, Hrg – $\top$ items from the right, Hrt – remaining right items. After applying `xsimpl_start`, our example becomes:

$$\overbrace{\mathsf{emp}}^{\mathsf{Hla}} * \overbrace{\mathsf{emp}}^{\mathsf{Hlw}} * \overbrace{[\mathsf{emp} * (H_1 \mathrel{-\!\!*} (H_2 * H_3)) * H_1 * \mathsf{emp}]}^{\mathsf{Hlt}} \vdash \underbrace{\mathsf{emp}}_{\mathsf{Hra}} * \underbrace{\mathsf{emp}}_{\mathsf{Hrg}} * \underbrace{[H_2 * H_3 * \mathsf{emp}]}_{\mathsf{Hrt}}.$$

Here and in what follows, we use an unfolded representation of `Xsimpl`, defined simply as a heap entailment with star-conjoined components, ending with the emp predicate.

Left-hand side simplification.

This next step performs symbolic simplification on the left-hand side by moving items out of Hlt: magic wands go to Hlw, emps are skipped, and everything else goes to Hla. The procedure pattern-matches on the structure of the left-hand side tuple (Hla, Hlw, Hlt), implemented as the following Ltac tactic in Rocq:

```
Ltac xsimpl_step_l tt :=
  match goal with ⊢ Xsimpl ?HL ?HR ⇒
  match HL with
  | (?Hla, ?Hlw, (?H ∗ ?Hlt)) ⇒
    match H with
    | \[] ⇒ eapply xsimpl_l_hempty
    | \[?P] ⇒ eapply xsimpl_l_hpure; intro
    | ... ⇒ eapply ...
  | ... ⇒  ...
```

Each branch applies a corresponding lemma, as we describe in the key cases below.

1. If the head of Hlt is emp, apply a lemma that simply drops it:

```
Lemma xsimpl_l_hempty : ∀ Hla Hlw Hlt HR,
  Xsimpl (Hla, Hlw, Hlt) HR → Xsimpl (Hla, Hlw, (emp ∗ Hlt)) HR.
```

2. If it is a pure fact [P], extract it as a hypothesis in the proof context:

```
Lemma xsimpl_l_hpure : ∀ P Hla Hlw Hlt HR,
  (P → Xsimpl (Hla, Hlw, Hlt) HR) → Xsimpl (Hla, Hlw, (\[P] ∗ Hlt)) HR.
```

3. If it is a separating conjunction, reassociate it to flatten nested stars:

```
Lemma hstar_assoc : ∀ H1 H2 H3, (H1 ∗ H2) ∗ H3 = H1 ∗ (H2 ∗ H3).
```

4. If it is a magic wand H1 −∗ H2, move it into the Hlw accumulator:

```
(* We only move it to the accumulator, so do not mention the wand explicitly *)
Lemma xsimpl_l_acc_hwand : ∀ H Hla Hlw Hlt HR,
  Xsimpl (Hla, (H ∗ Hlw), Hlt) HR → Xsimpl (Hla, Hlw, (H ∗ Hlt)) HR
```

5. Otherwise, accumulate the item in the Hla component:

```
Lemma xsimpl_l_acc_other : ∀ H Hla Hlw Hlt HR,
  Xsimpl ((H ∗  Hla), Hlw, Hlt) HR → Xsimpl (Hla, Hlw, (H ∗  Hlt)) HR.
```

After applying rules 1, 3, 4 and 5, our example becomes:

$$\underbrace{H_1}_{\mathsf{Hla}} * \underbrace{[(H_1 \mathrel{-\!\!*} (H_2 * H_3)) * \mathsf{emp}]}_{\mathsf{Hlw}} * \underbrace{\mathsf{emp}}_{\mathsf{Hlt}} \vdash \overbrace{\mathsf{emp}}^{\mathsf{Hra}} * \overbrace{\mathsf{emp}}^{\mathsf{Hrg}} * \overbrace{[H_2 * H_3 * \mathsf{emp}]}^{\mathsf{Hrt}}.$$

All items from Hlt have been processed, but we still need to handle the magic wands in Hlw via the cancellation rule 6:

6. When a magic wand's argument matches a heap assertion in Hla, cancel them using the lemma `xsimpl_l_cancel_hwand`; this removes the wand from Hlw, consumes its argument from Hla, and places the wand's conclusion into Hlt for further processing:

```
Lemma xsimpl_l_cancel_hwand : ∀ H1 H2 Hla Hlw Hlt HR,
  Xsimpl (emp, Hlw, (Hla ∗ H2 ∗  Hlt)) HR →
  Xsimpl ((H1 ∗  Hla), ((H1 −∗ H2) ∗ Hlw), Hlt) HR.
```

   If no match is found, the wand is kept via `xsimpl_l_keep_wand`:

```
Lemma xsimpl_l_keep_wand : ∀ H Hla Hlw Hlt HR,
  Xsimpl ((H ∗ Hla), Hlw, Hlt) HR → Xsimpl (Hla, (H ∗ Hlw), Hlt) HR.
```

For our example, the wand $H_1 \mathrel{-\!\!*} (H_2 * H_3)$ has its argument H_1 present in Hla. To locate it, the Ltac tactic `xsimpl_pick_unifiable` traverses Hla and rotates the matching conjunct to the front, ready for the cancellation lemma:

```
Ltac xsimpl_pick_unifiable H :=
  match goal with ⊢ Xsimpl (?Hla, ?Hlw, ?Hlt) ?HR ⇒
    hstars_search Hla ltac:(fun i H' ⇒  unify H H'; xsimpl_pick i)
  end.
```

The helper `hstars_search` recursively enumerates each conjunct of a separating conjunction and applies a test: here, Rocq's built-in `unify H H'` between the target `H` and each candidate `H'`. Once a match is found, `xsimpl_pick` applies one of several rotation lemmas to move the matched conjunct to the front:

```
Lemma hstars_pick_1 : ∀ H1 H, H1 * H = H1 * H.
Lemma hstars_pick_2 : ∀ H1 H2 H, H1 * H2 * H = H2 * H1 * H.
Lemma hstars_pick_3 : ∀ H1 H2 H3 H,
  H1 * H2 * H3 * H = H3 * H1 * H2 * H.
...
Lemma hstars_pick_11 : ∀ H1 H2 H3 H4 H5 H6 H7 H8 H9 H10 H11 H,
    H1 * H2 * H3 * H4 * H5 * H6 * H7 * H8 * H9 * H10 * H11 * H
  = H11 * H1 * H2 * H3 * H4 * H5 * H6 * H7 * H8 * H9 * H10 * H.
```

Since separating conjunction is represented explicitly, there is no general rotation lemma for an arbitrary number of conjuncts, placing an inherent bound on the entailments that xsimpl can handle. At the end, following rule 6, we remove H_1 from Hla and the magic wand from Hlw, placing $H_2 * H_3$ into Hlt, which results in the following proof obligation:

$$\underbrace{\mathsf{emp}}_{\mathsf{Hla}} * \underbrace{\mathsf{emp}}_{\mathsf{Hlw}} * \underbrace{[H_2 * H_3 * \mathsf{emp}]}_{\mathsf{Hlt}} \vdash \overbrace{\mathsf{emp}}^{\mathsf{Hra}} * \overbrace{\mathsf{emp}}^{\mathsf{Hrg}} * \overbrace{[H_2 * H_3 * \mathsf{emp}]}^{\mathsf{Hrt}}.$$

Processing Hlt once more using rule 5 yields:

$$\underbrace{[H_3 * H_2 * \mathsf{emp}]}_{\mathsf{Hla}} * \underbrace{\mathsf{emp}}_{\mathsf{Hlw}} * \underbrace{\mathsf{emp}}_{\mathsf{Hlt}} \vdash \overbrace{\mathsf{emp}}^{\mathsf{Hra}} * \overbrace{\mathsf{emp}}^{\mathsf{Hrg}} * \overbrace{[H_2 * H_3 * \mathsf{emp}]}^{\mathsf{Hrt}}.$$

This completes the left-hand side simplification. The right-hand side simplification then cancels matching predicates from both sides, which concludes the entire procedure.

3.2 Translation-Validation Approach

The Ltac-based approach described above is suboptimal: applying a rewriting lemma forces the proof assistant to both rewrite the goal *and* justify the rewrite. If our goal is to obtain the simplified form as quickly as possible, we can separate these two concerns: first, manipulate terms directly, and then replay the corresponding proof steps to obtain the proof term.

From the implementation perspective, we proceed as follows. Using Lean metaprogramming, we first extract the goal's syntax and obtain both sides of the heap entailment:

```
let target := (← getMainTarget).consumeMData
let some (Q, R) := target.app2? ``himpl | throwError
```

We then convert Lean's built-in `Expr` into a mixed representation that combines custom constructors with native Lean expressions (Fig. 3).

We encode $h_1 * h_2 * \ldots * h_n$ as the list $[h_1, h_2, \ldots, h_n]$. For example, $a * (b * c)$ becomes simply `[a, b, c]` rather than the deeply nested `Expr` encoding `.app (.app (.const ``hstar) a) (.app (.app (.const ``hstar) b) c)`. This offers two advantages:

(a) *Performance.* Operations such as `reverse` and `map` on `List` benefit from Lean's "functional but in place" optimisation [23]: for instance, `reverse` walks the linked list and reverses node pointers in place when the reference is unshared [17]. The equivalent operations on `Expr` are slower due to nested pattern matching overhead:

```
inductive hprop : Type where
  | sep : List hprop -> hprop
  | pure : Expr -> hprop
  | atomic : Expr -> hprop
  | top : hprop
  | emp : hprop
  | wand : hprop -> hprop -> hprop
  | exists : Name -> Expr -> hprop -> hprop
  | forall : Name -> Expr -> hprop -> hprop
```

Figure 3 The `hprop` mixed representation type.

```
-- list: flat pattern matching
match l with | .cons x l' => ... | .nil => ...
-- Expr: nested pattern matching
match e with
  | .app (.app (.const ``hstar _) h₁) h₂ => ...
  | .const ``hempty _ => ... | _ => ...
```

(b) *Extensibility.* Users who extend Yolo with new operators write simplification procedures over standard `List` primitives with rich built-in functions. The list representation also enables lemmas quantifying over an arbitrary number of conjuncts, needed during proof reconstruction, ensuring a consistent representation throughout the simplification.

Since simplification frequently tests whether two expressions are equal (*e.g.*, rule 6), retaining native `Expr` lets us reuse Lean's built-in `isDefEq`, which handles definitional equality efficiently via heuristics [18]. A fully deep embedding would require custom equality checking.

Once converted, the simplification follows the same algorithmic steps as in Sec. 3.1, but modifies lists directly instead of applying rewriting lemmas. The initialisation step stores the six components of the simplification algorithm's state in the following structure:

```
structure hState where
  hla   : List hprop
  hlw   : List hprop
  hlt   : List hprop
  hra   : List hprop
  hrg   : Bool
  hrt   : List hprop
  tacticScript : Array Syntax
```

The conceptual differences between the two approaches can be summarised as follows. Suppose emp appears in Hlt. In the Ltac-based approach, the tactic pattern-matches the goal and immediately applies the simplification lemma `emp_left`. In the Yolo approach, the list head `emp :: hs` is matched and the operation `hlt.set hs` performs the same simplification—skipping emp—without invoking any lemma. The application of `emp_left` is merely *recorded* in the `tacticScript` array and replayed later during the proof reconstruction phase. The lemmas in both approaches are conceptually identical; the Yolo variant is simply reformulated to use a list-based representation via the `sep` predicate, enabling more flexible manipulation:

```
def sep : List hProp → hProp
| [] => emp | [x] => x
| x :: xs => x * sep xs
```

Using lists, we can state theorems about an arbitrary number of conjuncts, because items can be addressed by index rather than requiring them at the front as in rule 6:

```
lemma hwand_process_cancel_left (i j : Nat) (l l' : List hProp) (H : hProp) :
 (l[i]? = l'[j]?) →
 (sep (l'.eraseIdx j) * sep ((sep (l.eraseIdx i) -* sep lr) :: hs) * hlt ⊢ H) →
 (sep l' * sep ((sep l -* sep lr) :: hs) * hlt  ⊢ H)
```

3.3 Extensibility via Data Types à la Carte

The mixed representation from the previous subsection uses a closed inductive type `hprop` (Figure 3). To allow users to extend Yolo with new Separation Logic operators without modifying this type, we adapt the Data Types à la Carte pattern [27], proposed originally as a solution to the famous *Expression Problem* [28]. The Expression Problem asks how to define a data type that is extensible both with new *variants* (*i.e.*, data constructors) and new *operations* over it, without modifying existing code and while retaining static type safety. In functional languages, algebraic data types make it easy to add new operations (as new functions with pattern matching) but adding a new variant requires changing the type definition and every function that matches on it. Object-oriented languages face the dual difficulty: subclassing easily adds new variants, but adding a new operation requires modifying every existing class. The key insight of Data Types à la Carte is to *decouple* the recursive structure of expressions from the operators they contain. Instead of a single monolithic data type, each operator is defined as a separate functor, and a generic expression type ties the recursion knot by parameterising over the choice of functor:

```
inductive Expr (f : Type u → Type u) : Type u
| In : f (Expr f) → Expr f
```

Individual operators are defined as separate types and composed via a sum functor:

```
inductive Val (e : Type) where | val : Int → Val e
inductive Add (e : Type) where | add : e → e → Add e
inductive SumF (f g : Type → Type) (e : Type) where
  | inl : f e → SumF f g e  | inr : g e → SumF f g e
infixr:60 " :+: " => SumF
```

An expression representing $118 + 1219$ can be, thus, encoded as follows:

```
def addExample : Expr (Val :+: Add) :=
  Expr.In (.inr (.add (Expr.In (.inl (.val 118))) (Expr.In (.inl (.val 1219)))))
```

Adding a new operator (*e.g.*, `Mul`) requires no changes to `Expr`: one simply defines a new type and composes it into the existing sum type. However, Lean's strict positivity restriction prevents this particular definition of `Expr`, because the type variable `f` occurs in a potentially non-positive position in the constructor `In`. This is a fundamental limitation of dependently-typed languages such as Lean and Rocq, which enforce strict positivity to guarantee logical consistency. As a result, the standard Data Types à la Carte pattern cannot be directly implemented in these systems. To address this challenge, prior work in Rocq has explored alternative approaches: Meta-theory à la carte [7] encodes data types using Church encodings, which rely on the impredicative-set option (not supported in Lean), while Coq-à-la-carte [9] uses a static code generation approach that depends on the external tool Autosubst 2. In contrast, our approach is implemented directly within the Lean type system and does not require any external tooling. Our alternative encoding preserves the extensibility benefits

of the original pattern while satisfying Lean's positivity checker. The key idea is to replace functor recursion with a generic tree parametrised by a plain operation type:

```
inductive opT (α : Type) where
  | op (op : α) (args : List (opT α)) : opT α
```

Operators are plain structures, composed via Lean's built-in `Sum` type:

```
structure TVal where v : Int
structure TAdd
structure TMul
```

Expressions are then built as trees: each node carries an operator tag and a list of subtrees serving as arguments. For example, $3 + 5$ and $3 + 4 \times 5$ are encoded as follows:

```
def myExpr : opT (TVal ⊕ TAdd) :=
opT.op (.inr TAdd.mk) [opT.op (.inl (TVal.mk 3)) [], opT.op (.inl (TVal.mk 5)) []]
```

```
def myExprMul : opT (TVal ⊕ TAdd ⊕ TMul) :=
  opT.op (.inr (.inl TAdd.mk)) [opT.op (.inl (TVal.mk 3)) [],
    opT.op (.inr (.inr TMul.mk)) [opT.op (.inl (TVal.mk 4)) [],
                                  opT.op (.inl (TVal.mk 5)) []]]
```

Semantics are defined via a type class, with one instance per operator. Each instance specifies how to compute the operator's result from the evaluated arguments:

```
class EvaluateT (α : Type) where
  eval (op : α) (args : List Int) : Int

instance : EvaluateT TVal where eval x _ := x.v
instance : EvaluateT TAdd where eval _ y := y.foldl (· + ·) 0
instance : EvaluateT TMul where eval _ y := y.foldl (· * ·) 1
```

Composition is handled by a generic sum instance, so each operator's semantics remains unchanged regardless of which other operators are present in the expression:

```
instance {α β : Type} [EvaluateT α] [EvaluateT β] : EvaluateT (Sum α β) where
  eval x y := match x with | .inl a => EvaluateT.eval a y | .inr b =>
    EvaluateT.eval b y
```

To evaluate an expression tree, we first evaluate its arguments recursively and then apply the operator's semantics to the results:

```
partial def opT.eval [EvaluateT α] : opT α → Int
  | .op op' args => EvaluateT.eval op' (EvaluateArguments args)
where EvaluateArguments : List (opT α) → List Int
  | [] => []
  | (.op op args) :: hs =>
      (EvaluateT.eval op (EvaluateArguments args)) :: (EvaluateArguments hs)

#eval opT.eval myExpr      -- 8
#eval opT.eval myExprMul   -- 23
```

This makes it a practical and reusable pattern for any Lean library that needs open, extensible data types. We apply the same approach to expressions containing Separation Logic operators, allowing users to extend Yolo with new connectives without modifying any of its core definitions. For example, to add a new operator `pure` for pure logic assertions, we define a

new structure and provide simplification procedures for it via the `Eval` type class. This class is analogous to the `EvaluateT` class for arithmetic operations from the examples above, but instead operates on the state of xsimpl algorithm:

```
structure TPure where e : Expr

class Eval (α : Type) (β : Type) ... where
  evaluate (op : α) (h : hprop β) : StateRefT (hstate β) TacticM Unit
  ...

instance : Eval TPure γ  where
  evaluate x h :=
  do
    customHave x.e -- add the pure fact to the proof context
    hlt.set hState.tail -- remove the pure fact from the list of heap assertions
  ...
```

4 Case Studies: Iris and LGTM

To demonstrate the generality and extensibility of Yolo, we instantiated it for two distinct foundational frameworks: Iris-Lean [14] and LGTM [10]. These two case studies represent opposite ends of the integration effort: due to its generic nature, Iris-Lean requires logic-independent automation that works out of the box, while LGTM demands deep, domain-specific extensions to Yolo's simplification algorithm to support relational reasoning patterns.

4.1 Iris-Lean

Iris-Lean [14] is a Lean port of Iris, a framework for higher-order concurrent Separation Logic. Iris-Lean supports MoSeL (an extension of the Iris Proof Mode, IPM [16]), a proof interface that can be instantiated for various Separation Logics [15]: iGPS [13], CFML [3], and CHL [5]. This interface provides generic axioms (*e.g.*, `emp_sep` for eliminating the `emp` predicate or `wand_intro` for introducing the magic wand), which formalise rules over predicates of an abstract type `PROP`. By exploiting these axioms, Yolo achieves the same level of generality for proof automation: any user who instantiates the proof interface with a custom logic in MoSeL can leverage Yolo's lazy proof mode "out of the box".

To adapt Yolo to Iris-Lean, we focused on the core Separation Logic predicates, namely star $*$, "magic wand" $-\!\!*$, pure assertions, the empty predicate emp, and Separation Logic quantifiers $\forall^h$ and $\exists^h$. Following the procedure outlined in Sec. 2.2.2, we defined these predicates as new types, provided a translation procedure to map Iris-Lean's native operators (*e.g.*, `Iris.BI.BIBase.sep`) to our internal representation, and specified their simplification behaviour. Since the simplification procedure mirrors that of CFML, we had to adapt the proof reconstruction lemmas to use the aforementioned axioms. Finally, following the steps detailed in Sec. 2.2.2, we registered the full set of operators with the tactic's main procedure:

```
elab "hMain" : tactic => hMain (TStar ⊕ TWand ⊕ TPure ⊕ TEmp ⊕ TAtomic ⊕
    TForall ⊕ TExists)
```

The extensible proof automation of Yolo can be seamlessly combined with interactive proofs in IPM to discharge complex goals involving Iris-specific logical connectives. Consider the example below involving the *later* ($\triangleright$) and *intuitionistic* ($\square$) modalities.

```
lemma yoloIPM:
    (H1 -∗ H2) ∗ H1 ∗ H3 ∗ H4 ∗ □ ▷^[5] P ∗ ▷^[3] P ∗ □Q ⊢ H3 ∗ H2 ∗ ▷^[4] P ∗ H4
:= by yolo
      iintro ⟨⟨HP1, HP2⟩, HPQ⟩
      imodintro
      iexact HP2
```

In Iris, the *later* modality ($\triangleright P$) asserts that P holds after one step of computation, while the *intuitionistic* modality ($\square P$) denotes that P is persistently true and operates independently of the spatial heap. Informally, this entailment is valid because the basic spatial resources (H_1, H_2, H_3, H_4) perfectly cancel each other out via standard Separation Logic rules. Meanwhile, the persistent propositions $\square \triangleright^5 P$ and $\square Q$ can be dropped from the left-hand side, which leaves us with the goal $\triangleright^3 P \vdash \triangleright^4 P$, which holds because $\triangleright^4 P$ is *later* than $\triangleright^3 P$. To prove it formally, we first invoke `yolo` for standard SL operators, which simplifies the goal to:

$$\square \triangleright^5 P * \triangleright^3 P * \square Q \vdash \triangleright^4 P$$

The remaining goal is discharged by using `imodintro` to simplify the *later* modality, followed by `iexact` to match the hypothesis directly from the context, which completes the proof. Right now Yolo supports only the core SL operators, but it can be extended by following the same procedure (Sec. 2.2.2).

4.2 LGTM

LGTM [10] is a Hoare-style relational Separation Logic tailored for verifying computations over structured and sparse data. Since LGTM is a relational program logic, its heap predicates operate over families of program executions. While its baseline automation is heavily inspired by CFML, LGTM requires the addition of new operators and further modifications to the simplification procedure to support common patterns of reasoning about properties of sets of programs. LGTM's state assertions are defined as predicates of type `hhProp` α, parameterised by an *index* type α, whose values serve as "addresses" of individual programs in a relational triple. Consequently, LGTM lifts standard Separation Logic operators to indexed assertion families: for instance, `hhsingle s p v` asserts a points-to relationship for every index within a set `s`, and `hharrayFun s f n p` represents an array across the index set `s`. Within this relational context, cancellation of atomic predicates in logical assertions about indexed states becomes significantly more complex, as it requires comparing not just pointer expressions but also their associated index sets.

Following the steps described in Sec. 2.2.2 we added new types: `THhSingle` for `hhsingle s p v` and `THhArray` for `hharrayFun s f n p`, and provided new simplification procedures.

```
structure THhSingle where
  s : Expr      -- set
  p : Expr      -- pointer
  v : Expr      -- value
structure THhArray where
  s : Expr      -- set
  f : Expr      -- array content function
  n : Expr      -- length
  p : Expr      -- pointer
-- The sum type for LGTM's operators
abbrev YOps := TStar ⊕ TWand ⊕ TQWand ⊕ TPure ⊕ TEmp ⊕ TAtomic ⊕ TTop ⊕
    TForall ⊕ TExists ⊕ THhSingle ⊕ THhArray
```

Table 1 Performance comparison of xsimpl vs. Yolo on CFML-style benchmarks. The columns provide the name of the benchmark (**goal**), execution time of CFML-style xsimpl, Yolo's unverified simplification, Yolo's proof reconstruction (PR), and the ratio of xsimpl's runtime *w.r.t.* that of Yolo (simplifier + proof reconstruction). All times are given in milliseconds.

#	goal	CFML xsimpl	Yolo xsimpl	PR	old / (yolo + PR)
1	`linearInterp_spec`	2205	190	483	3.276
2	`bilinearInterp_spec`	9283	517	912	6.496
3	`incr_spec`	163	24	87	1.468
4	`add_pointer_spec`	325	54	164	1.491
5	`findIdx_spec`	1700	147	1240	1.226
6	`findIdx_spec'`	102	16	77	1.097
7	`array_exists_spec`	2528	256	900	2.187
				Average:	**2.463**

Because LGTM proofs involve highly complex and heavy entailments, its baseline automation (the `ysimp` tactic) suffers from severe performance bottlenecks during interactive development. This makes LGTM an ideal target for Yolo's approach. By offloading the heavy computational burden of cancellation to the fast unverified phase, Yolo drastically reduces interactive latency, which provides a substantially more user-friendly proof development experience. For example, the time of automation for proving correctness of `intervalCountQuery`, a program that computes the number of points in a two-dimensional array-encoded region, in LGTM went down from 33 sec to 4.1 sec (*i.e.*, it became 8 × faster) with the fast unverified phase, and to 19.7 sec with the proof reconstruction enabled.

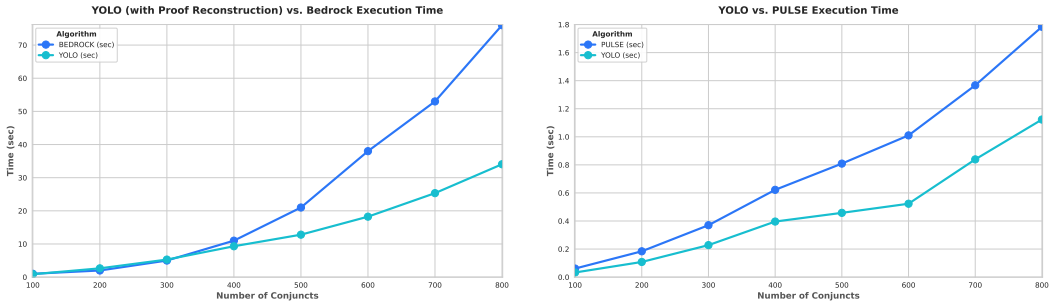
5 Evaluation and Benchmarks

We evaluated the performance of Yolo by comparing its runtimes when discharging non-trivial goals against state-of-the-art SL-based verifiers, both foundational and auto-active. Our suite of benchmarks includes both Separation Logic proof goals from real-world examples and artificially constructed large heap entailments. All experiments were conducted on a Dell workstation running Ubuntu, equipped with a 5.4GHz Intel Core i7-14700 CPU.

5.1 Comparing Yolo to xsimpl

In our first experiment, we compare the performance of Yolo on entailment-heavy proof goals against the baseline—an implementation of the xsimpl tactic in Splean, a faithful Lean port of CFML.[2] Our suite of benchmarks is presented in Tab. 1. It is assembled from proof goals extracted from programs that manipulate array-based representations of geometric objects (rows 1–2) and in-place heap-manipulating traversals of linked lists and arrays (rows 3–7). These proofs involve a combination of pure logical goals and Separation Logic goals, requiring both standard tactics and Splean's domain-specific tactics. To isolate the performance of our automation, we instrumented the proof process to record the time spent specifically on tactic execution. Tab. 1 demonstrates that our new approach achieves a 2.5× speedup on average with proof reconstruction enabled, and is about 10× faster than xsimpl in an unverified mode.

[2] `https://github.com/verse-lab/splean`



(a) Bedrock vs Yolo **(b)** Pulse vs Yolo

Figure 4 Performance trends on large SL entailment goals compared to Bedrock and Pulse.

5.2 Yolo against Bedrock and Pulse

Our second experiment compares Yolo against state-of-the-art automated Separation Logic verifiers representing two fundamentally different design points: Bedrock [6,19], a foundational Rocq-based simplifier that uses proof by reflection, and Pulse [8], an SMT-backed Separation Logic embedded in F*. To isolate the core simplification performance from framework-specific differences in proof methodology and tactic automation, we use synthetic heap entailments of the form $*_{i=1}^{N} H_i \vdash *_{i=1}^{N} H_i$, where both sides are randomly shuffled and N is the number of conjuncts. We focus exclusively on separating conjunction, as it is the most fundamental operation supported across all evaluated frameworks.

To evaluate Yolo against Bedrock, we used Rocq's `Time` command to measure the execution of the `sepLemma` tactic. For Yolo in Lean, we used a custom metaprogramming-based instrumentation to precisely record execution timing. For Pulse, we measured execution via the command-line `time` utility, excluding initialisation overhead (*e.g.*, `assume val`) to ensure a fair comparison. Figures 4a (Yolo vs Bedrock) and 4b (Yolo vs Pulse) illustrate the comparative performance as N increases. In both scenarios, Yolo's performance degrades more gracefully in comparison with its competitors.

5.3 Yolo against Iris (iFrame)

Iris's Proof Mode [16] provides iFrame, a standard tactic for closing goals by using selected hypotheses. In our benchmark, we first invoke `iIntros` to introduce the hypotheses and then run iFrame. iFrame is already slower on relatively small goals: for 100 conjuncts, it takes 5.67 seconds, while Yolo discharges them 5 × faster, and the difference becomes more pronounced as the number of conjuncts increases in the log-scale running-time comparison (Fig. 5).

5.4 Yolo against VST and HTT

Finally, we compare Yolo against two mature Rocq-based Separation Logic frameworks to understand how our approach fares against proof automation tailored for different styles of embedding a Separation Logic into a proof assistant (deep and shallow).

Verified Software Toolchain (VST) [1] is a Separation Logic designed to reason about realistic programs written in C and implemented as a deep embedding into Rocq. VST provides several automation tactics for different purposes. Our use case exactly matches the `cancel` tactic, which, according to the VST documentation [2], removes the same conjuncts

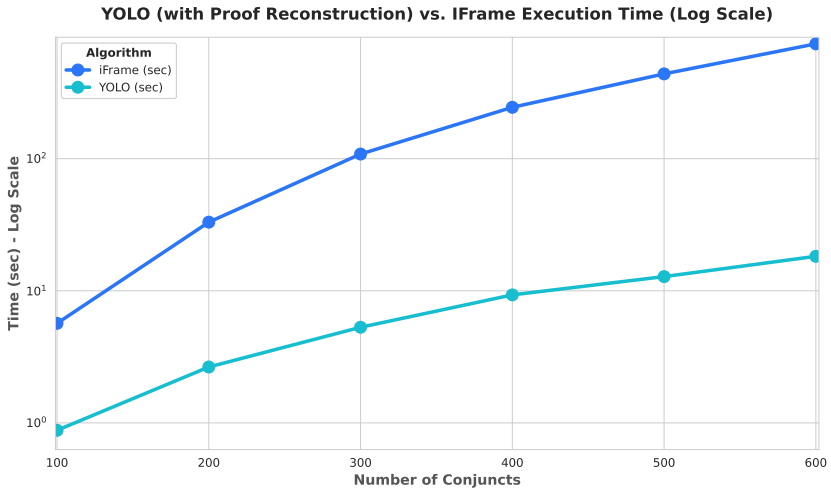


Figure 5 Performance comparison of iFrame and Yolo on Iris framing goals (logarithmic scale).

from both sides of a heap entailment. This tactic is significantly slower than simplifiers of Yolo and Bedrock, even at $N = 300$, it is 4.5 times slower than both, as shown in Tab. 2.

Hoare Type Theory (HTT) [21] is a minimalistic Separation Logic shallowly embedded into Rocq. It provides an automation tactic called `hhauto` to prove goals of the form $\uplus_{i=1}^{N} H_i = \uplus_{i=1}^{N} H_i$, where H_i is a heap part, $\uplus$ denotes disjoint union for heap parts, and N is the number of heap parts. Even at $N = 200$, it takes more than 21 minutes to complete, which is 120 times slower than VST's `cancel` tactic and 480 times slower than Yolo tactic.

6 Related Work

Several frameworks provide automation for heap entailment proofs in foundationally embedded Separation Logics. The xsimpl tactic of CFML [4] performs heap entailment simplification through a sequence of Ltac metaprogramming steps that generate foundational proof terms. Yolo directly improves upon xsimpl: by decoupling simplification from proof generation via translation validation, it achieves a 2.5× average speedup while remaining foundational. VST [2], a framework for verifying C programs, provides various automation tactics for SL goals, including `cancel`, which removes identical conjuncts from both sides of an entailment. As shown in Sec. 5, VST's `cancel` is significantly slower than both Yolo and Bedrock, even on moderately sized entailments. Bedrock [6, 19] uses proof-by-reflection with deep embeddings to achieve verified simplification procedures. While Bedrock is efficient, its

Table 2 Speedup of Yolo and Bedrock over VST on synthetic entailments.

#	Conjuncts (N)	VST/ (Yolo + PR)	VST/ Bedrock
1	100	2.887	2.533
2	200	3.54	4.687
3	300	4.46	4.721
4	400	4.833	4.09
5	500	6.723	4.1
6	600	7.533	3.615

reliance on deep embeddings limits expressiveness (*e.g.*, it cannot handle specifications with dependently-typed terms) and makes it difficult to extend with new connectives. Yolo matches Bedrock's performance while offering greater expressiveness through its mixed deep/shallow representation and greater extensibility through its Data Types à la Carte encoding. HTT [21], a Separation Logic shallowly embedded into Rocq, provides the `hhauto` tactic for heap entailment, which is orders of magnitude slower than Yolo on even average-size goals. Beyond foundational approaches, Pulse [8], a concurrent Separation Logic embedded in F*, provides a simplification procedure backed by an SMT solver. Pulse does not produce independently checkable proof certificates, whereas Yolo's lazy proof mode offers comparable interactive performance while retaining the ability to reconstruct foundational proofs.

The original Data Types à la Carte approach [27] leverages Haskell's rich type system to define extensible data types in a modular way. However, it is not directly suitable for proof assistants like Rocq and Lean due to their more restrictive type systems (strict positivity requirements). Several works have adapted this approach for proof assistants, including Meta-theory à la carte [7] and Coq-à-la-carte [9] for Rocq, and Modular Type Safety Proofs [24] for Agda. In Lean, recent work [25] uses metaprogramming capabilities to combine several inductive types into a single one. However, this implementation is at an early stage and has several limitations: functions must be in a specific form (pattern-matching on only one argument), and mutual recursion is not supported. These works are aimed primarily at supporting modularity and extensibility of proofs. Our approach is simpler: it is based on type classes and imposes no restrictions on the functions, which makes it better suited for efficient implementation of Yolo's metaprogramming and simplification procedures.

7 Conclusion

We presented Yolo, a lazy proof automation system for heap entailment in Separation Logic that addresses three key limitations of existing approaches: poor performance due to metaprogramming overhead, limited expressivity, and restricted extensibility.

Our key insight is to decouple the simplification phase from proof generation using translation validation, enabling fast interactive proof development while maintaining foundationality. Yolo is implemented in Lean and works generically with any MoSeL instance [15]. To achieve extensibility, we propose a novel adaptation of the Data Types à la Carte pattern [27] that overcomes Lean's strict positivity restriction, enabling users to add new Separation Logic connectives without modifying Yolo's core definitions. Our evaluation demonstrates performance improvements: 2.5× average speedup over the foundational CFML approach and competitive performance with state-of-the-art SL entailment simplifiers.

References

1 Andrew W. Appel. Verified Software Toolchain - (Invited Talk). In *ESOP*, volume 6602 of *LNCS*, pages 1–17. Springer, 2011. doi:10.1007/978-3-642-19718-5_1.

2 Andrew W. Appel, Lennart Beringer, Qinxiang Cao, and Josiah Dodds. *Verifiable C: Applying the Verified Software Toolchain to C Programs*. Princeton University, 2023. URL: https://vst.cs.princeton.edu/download/VC.pdf.

3 Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *ICFP*, pages 418–430. ACM, 2011. doi:10.1145/2034773.2034828.

4 Arthur Charguéraud. Separation Logic for Sequential Programs (Functional Pearl). *Proc. ACM Program. Lang.*, 4(ICFP):116:1–116:34, 2020. doi:10.1145/3408998.

5 Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare logic for certifying the FSCQ file system. In *SOSP*, pages 18–37. ACM, 2015. doi:10.1145/2815400.2815402.

6 Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245. ACM, 2011. doi:10.1145/1993498.1993526.

7 Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *POPL*, pages 207–218. ACM, 2013. doi:10.1145/2429069.2429094.

8 Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil Swamy. PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs. *Proc. ACM Program. Lang.*, 9(PLDI):1516–1539, 2025. doi:10.1145/3729311.

9 Yannick Forster and Kathrin Stark. Coq à la carte: a practical approach to modular syntax with binders. In *CPP*, pages 186–200. ACM, 2020. doi:10.1145/3372885.3373817.

10 Vladimir Gladshtein, Qiyuan Zhao, Willow Ahrens, Saman P. Amarasinghe, and Ilya Sergey. Mechanised hypersafety proofs about structured data. *Proc. ACM Program. Lang.*, 8(PLDI):647–670, 2024. doi:10.1145/3656403.

11 Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. In *ICFP*, pages 163–175. ACM, 2011. doi:10.1145/2034773.2034798.

12 The Iris Project. The Iris 4.3 Reference, 2024. URL: https://iris-project.org/.

13 Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP*, volume 74 of *LIPIcs*, pages 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.ECOOP.2017.17.

14 Lars König. An improved interface for interactive proofs in separation logic. Master's thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2022.

15 Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.*, 2(ICFP):77:1–77:30, 2018. doi:10.1145/3236772.

16 Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, pages 205–217. ACM, 2017. doi:10.1145/3009837.3009855.

17 Lean 4 documentation: Init.Data.List.Basic. https://leanprover-community.github.io/mathlib4_docs/Init/Data/List/Basic.html#List.reverse, 2025.

18 Metaprogramming in Lean 4: MetaM. https://leanprover-community.github.io/lean4-metaprogramming-book/main/04_metam.html, 2025.

19 Gregory Malecha. Building Bedrock: Verifying a program verifier. In *The Coq Workshop*, 2012.

20 Gregory Malecha, Adam Chlipala, and Thomas Braibant. Compositional computational reflection. In *ITP*, volume 8558 of *LNCS*, pages 374–389. Springer, 2014. doi:10.1007/978-3-319-08970-6_24.

21 Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *POPL*, pages 261–274. ACM, 2010. doi:10.1145/1706299.1706331.

22 George C. Necula. Translation Validation for an Optimizing Compiler. In *PLDI*, pages 83–94. ACM, 2000. doi:10.1145/349299.349314.

23 Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. Perceus: garbage free reference counting with reuse. In *PLDI*, pages 96–111. ACM, 2021. doi:10.1145/3453483.3454032.

24 Christopher Schwaab and Jeremy G. Siek. Modular type-safety proofs in Agda. In *PLPV*, pages 3–12. ACM, 2013. doi:10.1145/2428116.2428120.

25 Ramy Shahin. Towards Modular Composition of Inductive Types Using Lean Metaprogramming. In *TYPES*, June 2025. Abstract and Slides available online.

26 Gordon Stewart, Lennart Beringer, and Andrew W. Appel. Verified heap theorem prover by paramodulation. In *ICFP*, pages 3–14. ACM, 2012. doi:10.1145/2364527.2364531.

27 Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008. doi:10.1017/S0956796808006758.

28 Philip Wadler. The Expression Problem. Email to the Java-Genericity mailing list, 1998. November 12, 1998. URL: https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.