Internship Report: Investigating Decision Procedures for Fragments of the Iris Separation Logic Framework

Zhicheng Hui
École Polytechnique

Advisor: François Pottier
Cambium, Inria Paris
August 18, 2025

Contents

1 Introduction

1.1 Motivation and Problematic

Separation logic is a successful formalism on program verification[30]. It partly originates from the logic of bunched implications[31] featuring two logical connectives: $*$ and $-\!\!*$, which resemble normal conjunction and implication in the sense that:

$$A * B \vdash C \quad \text{iff} \quad A \vdash B -\!\!* C$$

The standard model in terms of program memory provides a simple way to understand them: Each proposition in separation logic specifies some property holding on a chunk of heap memory, they are often called heap predicates or spatial predicates. The separating conjunction $A * B$ specifies a heap composed of two disjoint parts, one satisfying A and the other satisfying B, while the separating implication $A -\!\!* B$ specifies a heap which would satisfy B whenever extended with a disjoint heap satisfying A. Such interpretation provides a concise representation for non-aliasing property, making separation logic especially useful for reasoning about heap-manipulating programs in a modular fashion.

Over the past years, there have been many evolutions of separation logic. Among them, Iris[21] is a higher-order concurrent separation logic framework. It provides a simple set of language-independent, primitive rules useful for reasoning about concurrent, heap-manipulating programs. The expressive power of Iris has led to many applications in verifying fine-grained concurrent programs[27, 39], secure language implementations[37, 19, 25], and certified operating systems[13, 12].

The wide application of separation logic has created a strong demand for corresponding automated deduction tools, whose development remains a challenging task. In fact, separation logic with equality theory and a two-record-field points-to predicate $l \mapsto (v_1, v_2)$ is already undecidable[11], and decidability results often rely on restricting the logic to a specialized set of predicates and excluding the separating implication $-\!\!*$. For example, most existing automated deduction tools for separation logic deal with (variants of) the symbolic heap fragment[1][8]. This fragment excludes separating implication and defines two predicates: the single-record-field points-to predicate $l \mapsto v$ and a predicate for describing singly linked list segments. Formulae in symbolic heap have the syntactic form:

$$(\phi_1 \wedge \phi_2 \wedge \cdots) \wedge (\psi_1 * \psi_2 * \cdots)$$

where ϕ_i denotes (dis)equalities, and ψ_i denotes heap predicates. This separation of a non-spatial part from a separating conjunction of spatial predicates has become a practical design choice for separation logic tools. For example, assertions in Verified Software Toolchain[5] admit the form[6]:

$$\text{PROP}(\phi_1; \phi_2; \cdots)\text{LOCAL}(v_1; v_2; \cdots)\text{SEP}(\psi_1; \psi_2; \cdots)$$

which additionally includes a LOCAL part describing the values of C program variables, irrelevant to the PROP and SEP parts. Iris proof mode[22] also follows that pattern. When reasoning in separation logic using the Rocq proof assistant[4], it provides a proof context divided into three parts: the Rocq context containing variables and non-spatial hypotheses, the spatial context containing spatial predicates, along with a persistent context containing persistent spatial predicates[2].

[1] In SL-COMP 2018[2], a competition of solvers for separation logic, only one division among eleven contains problem not based on symbolic heap, and the only participant is CVC4[7].

[2] See Section 3 for the definition of persistent predicates.

These tools have proven useful in software verification, where the required predicates are similar to those of symbolic heap. However, they fall short in supporting the development of new variants of separation logic. For example, many recent works in the Iris literature make use of custom "ghost theories"[19, 18, 16], that is, purely logical libraries where one or more logical predicates are defined, together with the laws they satisfy. It would be useful to develop a tool capable of reasoning in those ghost theories, in order to verify their consistency, or to facilitate their use in proofs, possibly via dedicated tactics as in Diaframe[29]. To that end, the tool needs to support user-defined predicates, separating implication, and common theories. In this internship report, I present the design and implementation of a prototype meeting these requirements.

Concretely, this tool attempts to prove $P \vdash \bot$, where P is a ghost theory given as input. Appendix A illustrates a ghost theory together with an inconsistency proof that the tool can generate in graphical form. The code for this tool is available on GitHub[3].

1.2 Context and Progress of the Internship

This M1 research internship was carried out in the Cambium team at INRIA Paris under the supervision of François Pottier. The internship topic is on designing and implementing a decision procedure for a fragment of the Iris separation logic. This topic combines research, algorithm design and OCaml programming, due to the foreseeable engineering effort involved in building an automated deduction tool and optimizing its performance. Mr. Pottier anticipated some difficulties in the process, but considered it both fun and instructive.

My first task was to implement an incomplete decision procedure for a simple propositional separation logic defined in Section 3, with which we are already able to write non-trivial problem instances. This decision procedure is incomplete because it may give "unknown" as answer when the exploration exceeds the specified depth limit. We then tested the algorithm on those instances, and profiled the implementation to find possible optimizations. Subsequent steps of the internship followed a similar pattern: for each extension, we tried to find interesting examples and efficient implementations. Some of these examples were adopted from the Rocq development of Iris [1], others were generated randomly. It turns out that the lack of real-world problem instances posed challenges in the validation of our design and implementation choices, as synthetic data may fail to capture the structure of realistic use cases.

Even though most other automated deduction tools do not share with us their target logic or reasoning mechanism, we nonetheless share a set of common implementation challenges, such as efficient term indexing and subsumption checking techniques. During my internship, some effort has been dedicated to identifying such shared challenges, as well as to comparing our approach with that of existing tools. This made me more familiar with the research area of automated theorem proving.

As part of my internship activities, I listened to various talks on separation logic, type theory, and mechanized mathematics. Thanks to Mr. Pottier, I also had the opportunity to attend the Iris Workshop 2025 held at INRIA Paris, which involves Iris-related research.

1.3 Structure of the Report

The remainder of this report is organized as follows: Section 2 introduces some formalism and notions necessary for understanding certain parts of the report. A first fragment of separation logic that the tool deals with is defined in Section 3, with which I present the core proof search algorithm and some implementation details. Two subsequent sections focus on extending this

[3] https://github.com/Laplace-Demon/IrisSearch

logic with first-order quantification and disjunction, both being basic components of Iris formulae. Finally, I conclude this report in Section 6 with discussions on possible directions of further development of this tool.

2 Preliminaries

We fix a universe Val of values and consider formulae in many-sorted first-order logic over a signature $\Sigma = (\mathrm{Sort}, \mathrm{Pred} \cup \{\simeq\}, \mathrm{Const}, \mathrm{Var})$, where Sort is the set of sort symbols, Pred is the set of predicate symbols, Const is the set of constant symbols, and Var is the infinite set of variables disjoint with Const. Each $v^S \in \mathrm{Val}$ is assigned a sort symbol $S \in \mathrm{Sort}$, denoting the sort of that value. Thus, for each sort symbol $S \in \mathrm{Sort}$, there exists a set $\mathrm{Val}(S) \subset \mathrm{Val}$ consisting of values of that sort. The family $\{\mathrm{Val}(S) \mid S \in \mathrm{Sort}\}$ forms a partition of Val.

We write D^* to denote the set of all finite sequences over D, i.e. $D^* = \bigcup_{n \in \mathbb{N}} D^n$, and use boldface to denote sequences. Whenever the distinction between constants and variables is not essential, we refer to both as members in the set $\mathrm{Term} = \mathrm{Const} \uplus \mathrm{Var}$. Each $P^{\mathbf{S}} \in \mathrm{Pred}$ is associated with a sequence of sort symbols $\mathbf{S} \in \mathrm{Sort}^*$, denoting the parameter sort sequence of that predicate. Each $t^S \in \mathrm{Term}$ is associated with one sort symbol $S \in \mathrm{Sort}$, denoting the sort of that term. We write $(t_1, t_2, \cdots)^{(S_1, S_2, \cdots)}$ as shorthand for the sequence $(t_1^{S_1}, t_2^{S_2}, \cdots)$. For $P^{\mathbf{S}} \in \mathrm{Pred}$ and $\mathbf{t}^{\mathbf{S}'} \in \mathrm{Term}^*$, the predicate $P^{\mathbf{S}}(\mathbf{t}^{\mathbf{S}'})$ is well-sorted if its parameter sort sequence $\mathbf{S}$ coincides with the argument sort sequence $\mathbf{S}'$. There is also a distinguished binary predicate $\simeq$. For $t^S, t'^{S'} \in \mathrm{Term}$, the predicate $t^S \simeq t'^{S'}$ is well-sorted if S coincides with S'. We assume all predicates are well-sorted and omit sort symbols whenever possible.

2.1 Separation Logic

In this subsection, we define the syntax and semantics of the separation logic fragment considered in this report. They are essential for understanding the logic.

Syntax Let t, t' range over Term, $\mathbf{t}$ range over Term^*, x range over Var, and P range over Pred. The language of separation logic is the set of formulae generated by the grammar:

$$\phi := P(\mathbf{t}) \mid \Box\,\phi_1 \mid \phi_1 * \phi_2 \mid \phi_1 \mathrel{-\!\!*} \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \forall x.\phi_1 \mid \exists x.\phi_1 \mid t \simeq t' \mid t \not\simeq t' \mid \bot \mid \top$$

An atomic formula, or atom, is $P(\mathbf{t})$, $\bot$, $\top$, or $\Box\,\phi$, where ϕ is an atom. We denote by Atom the set of atomic formulae under the fixed signature Σ. In the following, we use the standard notations

$$\exists(x_1, x_2, \cdots).\phi_1 \triangleq \exists x_1.\exists x_2.\cdots.\phi_1$$
$$\forall(x_1, x_2, \cdots).\phi_1 \triangleq \forall x_1.\forall x_2.\cdots.\phi_1$$

Semantics of the points-to fragment To ease understanding, we first give the semantics of that separation logic limited to the single-record-field points-to predicate. We fix two sorts Loc and Data such that Val(Loc) is countably infinite, and contains one element $\mathrm{null}^{\mathrm{Loc}}$. Recall the syntax of the points-to predicate:

$$\phi := \cdots \mid x^{\mathrm{Loc}} \mapsto y^{\mathrm{Data}}$$

The model (s, h) of a separation logic formula consists of a stack s and a heap h. The stack is a finite partial function $s : \mathrm{Var} \rightharpoonup_{\mathrm{fin}} \mathrm{Val}$ from variables to values, and the heap is a finite partial function $h : \mathrm{Val}(\mathrm{Loc}) \rightharpoonup_{\mathrm{fin}} \mathrm{Val}(\mathrm{Data})$ from locations to data. We say (s, h) is a model of ϕ when the relation $s, h \vDash \phi$ holds. The relation $s, h \vDash \phi$ is defined inductively as follows:

$$
\begin{array}{lll}
s, h \vDash x \mapsto y & \text{iff } h(x) = y & \\
& \text{and } x \neq \text{null}^{\text{Loc}} & \\
s, h \vDash \square\, \phi_1 & \text{iff } s, \emptyset \vDash \phi_1 & \\
s, h \vDash \phi_1 * \phi_2 & \text{iff exists } h_1, h_2 & \\
& \text{that } h = h_1 \uplus h_2 & \\
& \text{and } s, h_1 \vDash \phi_1 & \\
& \text{and } s, h_2 \vDash \phi_2 & \\
s, h \vDash \phi_1 \mathrel{-\!\!*} \phi_2 & \text{iff for every } h' & \\
& \text{if } \text{dom}(h) \cap \text{dom}(h') = \emptyset & \\
& \text{and } s, h' \vDash \phi_1 & \\
& \text{then } s, h \uplus h' \vDash \phi_2 & \\
s, h \vDash \phi_1 \wedge \phi_2 & \text{iff } s, h \vDash \phi_1 & \\
& \text{and } s, h \vDash \phi_2 & \\
s, h \vDash \phi_1 \vee \phi_2 & \text{iff } s, h \vDash \phi_1 & \\
& \text{or } s, h \vDash \phi_2 & \\
s, h \vDash \forall x. \phi_1 & \text{iff for every } x & \\
& \text{we have } s, h \vDash \phi_1(x) & \\
s, h \vDash \exists x. \phi_1 & \text{iff exists } x & \\
& \text{that } s, h \vDash \phi_1(x) & \\
s, h \vDash x \simeq y & \text{iff } s(x) = s(y) & \\
s, h \vDash x \not\simeq y & \text{iff } s(x) \neq s(y) & \\
s, h \vDash \top & \text{iff } \mathit{true} &
\end{array}
$$

The formula ϕ is consistent or satisfiable if there exists (s, h) such that $s, h \vDash \phi$ holds. it is valid if for every (s, h), the relation $s, h \vDash \phi$ holds.

We can gain some basic understanding of the semantics of separation logic. The stack s can be thought of as the stack frame in a typical programming language runtime, it contains an environment mapping each program variable to its current value. The heap h models the dynamically allocated memory of the program, it is specified by the points-to predicate $x \mapsto y$, where x is the memory address (pointer) and y the data stored in that memory location.

$\square\, \phi$ stands for the persistence modality[20] of Iris, this means ϕ holds forever and can be arbitrarily duplicated. Consequently, ϕ is independent of any particular heap.

Affine and linear separation logic Some developments of separation logic, including the symbolic heap fragment, would define certain rules as:

$$
\begin{array}{ll}
s, h \vDash x \mapsto y & \text{iff } h = \{(x, y)\} \\
& \text{and } x \neq \text{null}^{\text{Loc}} \\
s, h \vDash \square\, \phi_1 & \text{iff } h = \emptyset \\
& \text{and } s, \emptyset \vDash \phi_1 \\
s, h \vDash x \simeq y & \text{iff } s(x) = s(y) \\
& \text{and } h = \emptyset \\
s, h \vDash x \not\simeq y & \text{iff } s(x) \neq s(y) \\
& \text{and } h = \emptyset
\end{array}
$$

These definitions impose stricter constraints on the semantics by requiring the heap to match an exact shape. In other words, the model must contain precisely the heap predicates described, no additional being allowed. We call this variant linear separation logic. By contrast, under the previous semantics, the heap may satisfy additional predicates beyond those explicitly mentioned in the formula, we call this variant affine separation logic. Since Iris is based on affine separation logic, the fragment we study here also follows the affine interpretation. In particular, affine separation logic supports the weakening rule presented in Section 3.

Semantics of the propositional fragment As can be observed from the semantics above, the formulation of the disjointness property carried by separation logic connectives depends on

how the heap is defined, which itself depends on the concrete interpretation of heap predicates. Therefore, an abstraction over heap predicates requires a general model of heap to capture these properties.

We define the structure of a separation algebra[10] $(\mathcal{H}, \circ, \emptyset)$, where $\mathcal{H}$ is the set of heaps, $\circ$ is a partial binary operation representing the heap composition, and $\emptyset$ is a unit element for $\circ$ representing the empty heap. Disjointness is captured by the partiality of $\circ$, where the composition of two heaps is undefined if they overlap. We also require $\circ$ to be commutative and associative where defined. This structure is sometimes called a partial commutative monoid, and some definitions include the cancellativity property, but it is optional in our work.

Each predicate is satisfied on a (possibly empty) set of heap. We define the valuation

$$V : \text{Atom} \rightharpoonup \mathcal{P}(\mathcal{H})$$

assigning each closed atomic formula the set of heaps satisfying it. V is defined as a partial function in order to rule out atomic formulae containing free variables, or not of the form $P(\mathbf{t})$. No heap satisfies $\bot$, and every heap satisfies $\top$, so we have $V(\bot) = \emptyset$ and $V(\top) = \mathcal{H}$.

Given a separation algebra $(\mathcal{H}, \circ, \emptyset)$ and a valuation V, the semantic rules are defined as follows.

$$
\begin{array}{lr}
s, h \vDash P(\mathbf{t}) & \text{iff } V(P(\mathbf{t})) \text{ is defined} \\
& \text{and exists } h' \in V(P(\mathbf{t})) \\
& \text{that } h' \subseteq h \\
s, h \vDash \Box\, \phi_1 & \text{iff } s, \emptyset \vDash \phi_1 \\
s, h \vDash \phi_1 * \phi_2 & \text{iff exists } h_1, h_2 \\
& \text{that } h = h_1 \circ h_2 \\
& \text{and } s, h_1 \vDash \phi_1 \\
& \text{and } s, h_2 \vDash \phi_2 \\
s, h \vDash \phi_1 \mathrel{-\!\!*} \phi_2 & \text{iff for every } h' \\
& \text{if } h \circ h' \text{ is defined} \\
& \text{and } s, h' \vDash \phi_1 \\
& \text{then } s, h \circ h' \vDash \phi_2
\end{array}
$$

$$
\begin{array}{lr}
s, h \vDash \phi_1 \wedge \phi_2 & \text{iff } s, h \vDash \phi_1 \\
& \text{and } s, h \vDash \phi_2 \\
s, h \vDash \phi_1 \vee \phi_2 & \text{iff } s, h \vDash \phi_1 \\
& \text{or } s, h \vDash \phi_2 \\
s, h \vDash \forall x. \phi_1 & \text{iff for every } x \\
& \text{we have } s, h \vDash \phi_1(x) \\
s, h \vDash \exists x. \phi_1 & \text{iff exists } x \\
& \text{that } s, h \vDash \phi_1(x) \\
s, h \vDash x \simeq y & \text{iff } s(x) = s(y) \\
s, h \vDash x \not\simeq y & \text{iff } s(x) \neq s(y) \\
s, h \vDash \top & \text{iff } \mathit{true}
\end{array}
$$

2.2 Theory of Relations in Automated Deduction

In this subsection, we develop notions around relations in the context of automated deduction. The relations we study are binary relations on formulae in first-order logic. This provides a unifying framework for formulating the algorithmic problems in the remainder of the report.

Substitution A substitution $\sigma : \text{Var} \to \text{Term}$ is a function mapping each variable to a term. We denote by $\{x_1 \mapsto t_1, x_2 \mapsto t_2, \cdots\}$ the substitution σ defined as follows:

$$\sigma(x) = \begin{cases} t_i, & \text{if } x = x_i, \\ x, & \text{otherwise.} \end{cases}$$

We denote by $t\sigma$ the term obtained by replacing every variable x in t by $\sigma(x)$. We say σ is a renaming substitution when it is a permutation on Var.

Relations on atomic formulae In the context of automated deduction, the following relations between atomic formulae are often considered:

$$\begin{aligned} unif(t,t') &\quad \text{iff} \quad \exists\sigma.t\sigma = t'\sigma \\ inst(t,t') &\quad \text{iff} \quad \exists\sigma.t = t'\sigma \\ gen(t,t') &\quad \text{iff} \quad \exists\sigma.t\sigma = t' \\ var(t,t') &\quad \text{iff} \quad \exists\sigma.(t\sigma = t'\sigma \wedge \sigma \text{ is a renaming substitution}) \end{aligned}$$

There are different interpretations of variables, depending on whether they are bound by universal or existential quantifiers. The same relation, when instantiated under different interpretations, can lead to different meanings in practice. For example, let P be a unary predicate symbol, let $x \in \text{Var}$ and $a \in \text{Const}$. We have $inst(P(a), P(x))$ by considering $\sigma = \{x \mapsto a\}$. This relation can provide evidence for either of the following implications:

$$P(a) \Rightarrow \exists x.P(x) \quad \text{or} \quad \forall x.P(x) \Rightarrow P(a)$$

Multisets with infinite multiplicity Given a set S, a multiset $\mathcal{M}$ over S is defined as a function mapping each element in S to its multiplicity:

$$\mathcal{M} : S \to \mathbb{N} \cup \{\infty\}$$

Let s be an element of S, we write $\square\, s \in \mathcal{M}$ when $\mathcal{M}(s) = \infty$. We write $\mathcal{M}(S)$ for the collection of all multisets over S.

The multiplicity of an element stands for the number of occurrence of that element in the multiset. Multiplicity ranges over $\mathbb{N} \cup \{\infty\}$, where the symbol ∞ means infinity. We keep the usual order and arithmetic extended so that for all n, $n < \infty$ and $n + \infty = \infty$. We interpret ∞ as absorbing under subtraction so that we have $\infty - \infty = \infty$.

Relations on multisets of atomic formulae Multisets of atomic formulae are often used to represent more complex formulae. Depending on the context, such a multiset may be interpreted as a separating conjunction, a classical conjunction, or a disjunction of its elements. In the report, the intended interpretations of variables and multisets will be stated whenever they are used.

We denote by $\mathcal{M}\sigma$ the multiset obtained by applying the substitution σ on every element of the multiset. It is possible to lift the relations on atomic formulae to multisets, by both considering the relation itself and set inclusion. Concretely, we parameterize the relations by a binary relation R over multisets and define:

$$\begin{aligned} unif_R(\mathcal{M},\mathcal{M}') &\quad \text{iff} \quad \exists\sigma.R(\mathcal{M}\sigma,\mathcal{M}'\sigma) \\ inst_R(\mathcal{M},\mathcal{M}') &\quad \text{iff} \quad \exists\sigma.R(\mathcal{M},\mathcal{M}'\sigma) \\ gen_R(\mathcal{M},\mathcal{M}') &\quad \text{iff} \quad \exists\sigma.R(\mathcal{M}\sigma,\mathcal{M}') \\ var_R(\mathcal{M},\mathcal{M}') &\quad \text{iff} \quad \exists\sigma.(R(\mathcal{M}\sigma,\mathcal{M}'\sigma) \wedge \sigma \text{ is a renaming substitution}) \end{aligned}$$

As an illustrating example, let P and Q be unary predicate symbols, let $x, y \in \text{Var}$ and $a \in \text{Const}$. Take $\mathcal{M}_1 = \{P(a), Q(x)\}$, $\mathcal{M}_2 = \{P(a), Q(y)\}$ and $\mathcal{M}_3 = \{P(x)\}$, we have $unif_{=}(\mathcal{M}_1, \mathcal{M}_2)$ as witnessed by the substitution $\{x \mapsto y\}$, and $gen_{\supseteq}(\mathcal{M}_2, \mathcal{M}_3)$ as witnessed by the substitution $\{x \mapsto a\}$.

Many automated deduction tools also make use of some total ordering relations on terms. This ordering relation can be lifted to multisets, by sorting each multiset into a list of elements in nondecreasing order and compare them lexicographically. Let $\preceq$ be a total order, we denote by $\preceq^{\mathcal{M}}$ the corresponding total order obtained this way.

Relations under the theory of equality The theory of equality $\mathcal{E}$ is a set of (dis)equalities between terms, from which other (dis)equalities can be derived by standard logical inference rules. The precise inference rules for equality are not specified here, as they are standard in first-order logic with equality.

We denote by $\vdash_{\mathcal{E}}$ for entailments under the equality theory $\mathcal{E}$. Consequently, $\vdash_{\mathcal{E}} t \simeq t'$ says the formula $t \simeq t'$ can be entailed by $\mathcal{E}$. From $\vdash_{\mathcal{E}}$, we can define an equivalence relation $\equiv_{\mathcal{E}}$ on atomic formulae that extends syntactic equality:

$$\begin{array}{rl}
\bot \equiv_{\mathcal{E}} \bot & \text{iff } \mathit{true} \\
\top \equiv_{\mathcal{E}} \top & \text{iff } \mathit{true} \\
\Box\,\phi \equiv_{\mathcal{E}} \Box\,\phi' & \text{iff } \phi \equiv_{\mathcal{E}} \phi' \\
P(t_1, t_2, \cdots) \equiv_{\mathcal{E}} P'(t_1', t_2', \cdots) & \text{iff } P = P' \\
 & \text{and } \vdash_{\mathcal{E}} t_1 = t_1' \\
 & \text{and } \vdash_{\mathcal{E}} t_2 = t_2' \\
 & \text{and } \cdots
\end{array}$$

We can exploit $\equiv_{\mathcal{E}}$ in the relations defined above:

$$\begin{array}{rl}
\mathit{unif}_{\mathcal{E}}(t, t') & \text{iff } \exists\sigma.t\sigma \equiv_{\mathcal{E}} t'\sigma \\
\mathit{inst}_{\mathcal{E}}(t, t') & \text{iff } \exists\sigma.t \equiv_{\mathcal{E}} t'\sigma \\
\mathit{gen}_{\mathcal{E}}(t, t') & \text{iff } \exists\sigma.t\sigma \equiv_{\mathcal{E}} t' \\
\mathit{var}_{\mathcal{E}}(t, t') & \text{iff } \exists\sigma.(t\sigma \equiv_{\mathcal{E}} t'\sigma \wedge \sigma \text{ is a renaming substitution})
\end{array}$$

We extend $\equiv_{\mathcal{E}}$ to multisets of atoms as follows. Let $\mathcal{M}$ and $\mathcal{M}'$ be two multisets of atoms. We have:

$$\mathcal{M} \equiv_{\mathcal{E}} \mathcal{M}' \quad \text{iff} \quad \exists f : \mathcal{M} \xrightarrow{\sim} \mathcal{M}' \text{ such that } \forall \phi \in \mathcal{M},\ \phi \equiv_{\mathcal{E}} f(\phi)$$

where f is a bijection between equivalent elements of the two multisets.

Based on a multiset equivalence relation modulo an equality theory, we are able to define multiset inclusion relations modulo an equality theory. Let $\mathcal{M}$ and $\mathcal{M}'$ be two multisets of atoms. We have:

$$\mathcal{M} \subseteq_{\mathcal{E}} \mathcal{M}' \quad \text{iff} \quad \exists \mathcal{N} \subseteq \mathcal{M}' \text{ such that } \mathcal{M} \equiv_{\mathcal{E}} \mathcal{N}$$

It says that $\mathcal{M}$ is equivalent to a sub-multiset of $\mathcal{M}'$. Thus, $\mathcal{M}$ is included in $\mathcal{M}'$ under $\mathcal{E}$.

3 Reasoning on First Fragment of Iris

We follow the structure of Iris proof mode and consider a first fragment without quantifications, (dis)equalities and disjunctions. Let P range over Pred and $\mathbf{a}$ range over Const^*, we define the following grammar:

$$\begin{aligned}
\text{State} &:= \text{Atom} \mid \text{Atom} * \text{State} \\
\text{Atom} &:= \bot \mid \top \mid P(\mathbf{a}) \mid \Box\,\text{Atom} \\
\text{Law} &:= \text{State} \mathrel{-\!\!*} \text{State}
\end{aligned}$$

The tool reasons on an initial state under a collection of laws, where states are separating conjunctions of atoms, and laws are persistent separating implications. The initial state, the set of laws $\mathcal{I}$, the set of sorts, heap predicates and constants are given as input.

Persistence modality The assertion $\square\,\phi$ stands for persistence modality of ϕ, it can be read "forever ϕ", it means that ϕ holds forever, and can be used as many times as one wishes. During proof search, ϕ can be arbitrarily duplicated once $\square\,\phi$ is derived, Section 3.1 defines the proof rules concerning persistence modality.

In practice, each law is intended to be arbitrarily duplicable, and thus should be wrapped with $\square$. For simplicity, we omit this annotation by treating $\mathcal{I}$ as containing infinite copies of each law. This simplification does not reduce the expressive power of the logic system, since $\mathcal{L} \mathrel{-\!\!*} \mathcal{R}$ can be equivalently represented as $\square\,(\mathcal{L} * T \mathrel{-\!\!*} \mathcal{R})$ together with a trigger proposition T. Symbolically,

$$(\mathcal{L} \mathrel{-\!\!*} \mathcal{R}) \quad \equiv \quad \square\,(\mathcal{L} * T \mathrel{-\!\!*} \mathcal{R}) * T$$

3.1 Proof Search with Separating Implication

Syntactic entailment and inference rules We present a saturation-based refutational decision procedure, meaning that the decision procedure tries to reach $\bot$ by iteratively applying some inference rules. Some of these rules depend on the set of laws $\mathcal{I}$, and we write $\vdash_{\mathcal{I}}$ for the syntactic entailment relation between states under $\mathcal{I}$.

Let $\mathcal{L}, \mathcal{R}$ range over states, and let P, Q, R range over atoms. Fixing a set of laws $\mathcal{I}$, the inference rules defining $\vdash_{\mathcal{I}}$ are given below. When $\mathcal{I}$ is not essential, we simply write $\vdash$.

$$P * Q \vdash Q * P \quad *\text{-Comm}$$

$$(P * Q) * R \vdash \mathcal{P} * (Q * R) \quad *\text{-Assoc}$$

$$\frac{P \vdash Q}{P * R \vdash Q * R} \quad *\text{-Mono}$$

$$P * Q \vdash P \quad *\text{-Weakening}$$

$$\square\,P \vdash P \quad \square\text{-Elim}$$

$$\square\,P \vdash \square\,P * \square\,P \quad \square\text{-Dup}$$

$$\square\,P \vdash \square\square\,P \quad \square\text{-Idemp}$$

$$\frac{(\mathcal{L} \mathrel{-\!\!*} \mathcal{R}) \in \mathcal{I}}{\mathcal{L} \vdash_{\mathcal{I}} \mathcal{R}} \quad \text{App-Law}$$

Here, the rules can be grouped into four categories:

- $*$-Comm, $*$-Assoc, and $*$-Mono are structural rules of the separating conjunction $*$.
- $*$-Weak is the weakening rule of affine logic, allowing extra heap predicates to be arbitrarily discarded.
- $\square$-Elim, $\square$-Dup, and $\square$-Idemp ensure that persistent modalities can be freely eliminated, duplicated, and nested.
- App-Law encodes the main reasoning step with separating implication: whenever the left-hand side of a law is present in the current state, it may be replaced by the right-hand side.

Each of the above rule transforms one state into another. In practice, the only inference rule explicitly handled by the decision procedure is App-Law. All other rules are enforced implicitly by the representation of states, application of laws, and preprocessing steps.

- The commutativity and associativity rules of $*$ are handled by storing each state as a multiset of atoms.
- The monotonicity rule of $*$, also called the frame rule, is realized operationally by extracting substates when applying the rule App-Law. Given a state S and a law $\mathcal{L} \mathrel{-\!\!*} \mathcal{R}$, the decision procedure finds a decomposition $S \equiv \mathcal{M} * \mathcal{L}$ and produces the state $\mathcal{M} * \mathcal{R}$.

- The elimination and duplication rules of $\Box$ are handled by storing persistent atoms as having infinite multiplicity in the multiset representation of the state.
- The idempotence rule of $\Box$ is handled by preprocessing. Every nested occurrence of $\Box$ is flattened to a single $\Box$ before proof search.

The combination of $*$-Mono and App-Law yields the inference rule App-Law-Frame. The decision procedure operates by repeatedly applying this rule to the current state.

$$\frac{(\mathcal{L} \mathrel{-\!\!*} \mathcal{R}) \in \mathcal{I}}{\mathcal{M} * \mathcal{L} \vdash_{\mathcal{I}} \mathcal{M} * \mathcal{R}} \quad \textsc{App-Law-Frame}$$

We write $\vdash^*$ to denote the reflexive-transitive closure of $\vdash$. The decision procedure then tries to decide if $\mathcal{S} \vdash^* \bot$ holds, under a. We assume our inference system is refutationally complete: If it finally entails $\bot$, then the initial state is inconsistent. Symbolically,

$$\mathcal{S} \vdash^* \bot \Rightarrow \mathcal{S} \text{ is inconsistent}$$

Saturation Given a set of states $\mathbb{S}$, it is saturated with respect to a given set of inference rules if for every $\mathcal{S} \in \mathbb{S}$, for every $\mathcal{S}'$ such that $\mathcal{S} \vdash \mathcal{S}'$, $\mathcal{S}'$ also belongs to $\mathbb{S}$. For a state $\mathcal{S}$, we denote by $Cns(\mathcal{S})$ the set of states entailable from $\mathcal{S}$, i.e. $Cns(\mathcal{S}) = \{\mathcal{S}' \mid \mathcal{S} \vdash^* \mathcal{S}'\}$. $Cns(\mathcal{S})$ is a saturated set of states.

Starting from an initial state $\mathcal{S}$, a saturation-based proof search consists in systematically applying the inference rule to generate all states $\mathcal{S}'$ such that $\mathcal{S} \vdash^* \mathcal{S}'$, thereby constructing $Cns(\mathcal{S})$. In this process, it is able to check whether $\mathcal{S} \vdash^* \bot$ holds. Once no new states can be derived, the search terminates. However, we note that $Cns(\mathcal{S})$ may be an infinite set. In that case, termination is ensured by imposing a depth limit.

Law application Recall that states are stored as multisets of atoms. Applying a law $\mathcal{L} \mathrel{-\!\!*} \mathcal{R}$ in a state $\mathcal{S}$ involves checking whether $\mathcal{L} \subseteq \mathcal{S}$, and if so, computing the new state $(\mathcal{S} \setminus \mathcal{L}) \cup \mathcal{R}$ using multiset operations.

Balanced binary search trees are suitable for implementing multisets. In our system, we use the OCaml library `baby`[3], which provides set union, intersection and difference operations with complexity $O\big(m \log\big(\frac{n}{m} + 1\big)\big)$[9], where n and m denote the size of operand sets and $m \leq n$. Balanced binary search trees rely on a total ordering on elements. Instead of using lexicographical order, we compare atoms by the unique identifiers assigned to them. These identifiers are attributed once during their construction via hash-consing[17].

3.2 Pruning the Search Tree

During the process of proof search, a state $\mathcal{S}$ is considered redundant if it can be logically entailed by another known state $\mathcal{S}'$. In that case, we have $Cns(\mathcal{S}') \supseteq Cns(\mathcal{S})$, indicating that $\mathcal{S}$ provides no more information than $\mathcal{S}'$ and can be deleted. Detecting and eliminating redundant states helps to reduce the workload without compromising completeness.

Since redundancy detection requires solving an entailment problem, it is undecidable in separation logic. In practice, automated deduction tools introduce approximations, with the most common being subsumption checks. A state $\mathcal{S}$ is subsumed by another state $\mathcal{S}'$ if $\mathcal{S}$ is a subset of $\mathcal{S}'$. Here, multiset inclusion is used as a syntactic approximation of logical entailment. Symbolically,

$$\mathcal{S} \sqsubset \mathcal{S}' \quad \text{iff} \quad \mathcal{S} \subseteq \mathcal{S}'$$

Forward subsumption checking Checking the subsumption relation for every possible pair of existing states is computationally expensive. Instead, we restrict subsumption checks to newly derived states. Depending on the direction of $\sqsubset$, one distinguishes between forward and backward subsumption. In this work, we focus exclusively on forward subsumption checking, i.e. checking if newly derived state is subsumed by existing states.

The problem of forward subsumption checking is defined as follows: Given a set $\mathbb{S}$ of existing states and a fresh state $\mathcal{S}$, decide if there exists $\mathcal{S}' \in \mathbb{S}$ such that $\mathcal{S} \sqsubset \mathcal{S}'$ holds. Since states are multisets, we can solve this problem by storing $\mathbb{S}$ as a set-trie data structure[36].

Set-trie We impose an ordering $\preceq$ on atoms. Each multiset is stored in the set-trie as a sorted list of its elements. This ensures a canonical representation of multisets.

A set-trie is a tree with labeled nodes. The root is labeled with $\{\}$ and other nodes with arbitrary atomic formulae. A path from the root to certain internal node represents the list containing elements along that path in order. Nodes carry an additional flag to indicate the end of paths. As the name suggests, in a set-trie, common prefixes shared by sorted lists are represented by common paths the same way as in a trie.

Figure 1 shows a set-trie containing the multisets $\{P(a), Q(a)\}$, $\{P(a), Q(a), R(a)\}$, $\{P(a), Q(b)\}$, $\{P(b), Q(a), R(a)\}$, and $\{P(b), Q(b)\}$, suppose $P(a) \preceq P(b) \preceq Q(a) \preceq Q(b) \preceq R(a)$. Flagged nodes are represented with rectangles.

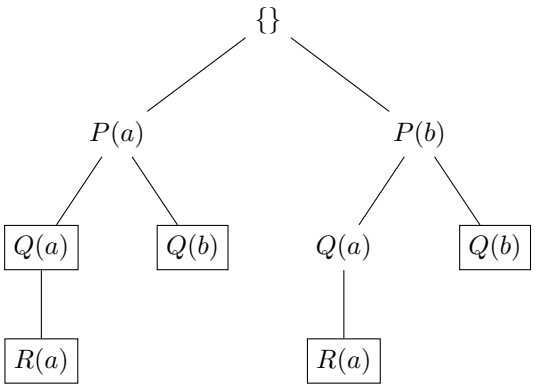

Figure 1: Example of a set-trie

Atoms of the form $\Box P$ are stored as having infinite multiplicities in the multiset representation. Since it is not possible to store lists with infinitely many copies of an element in the set-trie, we store $\Box P$ as a single element, then adjust the algorithm so that looking for any number of copies of P succeeds when $\Box P$ is present. This can be understood in terms of the compressed prefix tree (Patricia tree) representation. In this representation, a node labeled with $\Box P$ denotes an infinite sublist containing only P. We accordingly switch to the ordering $\preceq_\Box$ which satisfies:

$$\forall P \in \text{Atom}, P \preceq_\Box \Box P$$

Figure 2 shows a set-trie containing the multisets $\{P(a), \Box Q(a)\}$, $\{P(a), Q(a), R(a)\}$, $\{P(a), Q(b)\}$, $\{P(b), Q(a), R(a)\}$, and $\{P(b), Q(b)\}$. The path from the root to the leftmost leaf node stands for the sorted list: $[\{\}, P(a), Q(a), \Box Q(a)]$, which represents the multiset $\{P(a), \Box Q(a)\}$. It seems at first glance that we introduce multiple representations for one multiset. However, once a set-trie is fixed, each multiset admits a unique representation.

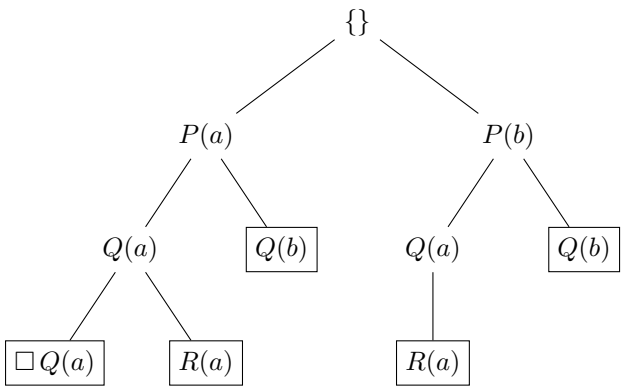

Figure 2: Example of a set-trie accounting for persistence modality

Recall we want to check the existence of superset of the query multiset. This can be done by descending simultaneously in the set-trie and the query multiset, while allowing to skip some elements in the set-trie. We implement two operations INSERT($node, list$) and QUERY($node, list$). INSERT($node, list$) inserts the multiset represented by $list$ into the set-trie rooted at $node$, and QUERY($node, list$) checks if there exists a superset of the multiset represented by $list$ in the set-trie rooted at $node$.

We denote by [] the empty list, $list.hd$ the first element of $list$, and $list.tl$ the remaining part of $list$ after removing $list.hd$. We denote by $node.label$ the label of $node$, $node.flag$ the boolean flag indicating whether a multiset terminates on $node$, and $node.children$ the set of children of $node$ in the set-trie. The two operations are presented in Algorithm 1 and Algorithm 2 respectively.

Proof search algorithm Algorithm 3 shows the proof search algorithm with forward subsumption checking. It maintains two data structures: a set of existing states implemented by set-trie, and a set of active states to be explored. At each iteration, the algorithm picks a state from the active set, and applies to it every applicable laws to generate new states. For each new state, it checks if the new state is subsumed by an existing state. The concrete proof-search strategy is unspecified, although our system uses a breadth-first search, where states are visited in order of increasing depth in the search tree.

3.3 Reasoning with Persistence Modality

So far, we have omitted another inference rule defined in Iris, saying that the persistence modality is monotone with respect to entailment.

$$\frac{P \vdash Q}{\Box P \vdash \Box Q} \quad \Box\text{-MONO}$$

In practice, we combine it with the APP-LAW-FRAME rule:

$$\frac{(\mathcal{L} \mathrel{-\!\!*} \mathcal{R}) \in \mathcal{I}}{\mathcal{M} * \Box \mathcal{L} \vdash_{\mathcal{I}} \mathcal{M} * \Box \mathcal{R}} \quad \text{APP-LAW-FRAME-}\Box$$

This can be seen as a strengthening of APP-LAW-FRAME, where persistent modality propagates through law application. This rule is enforced in the following way: Each time a law

Algorithm 1 INSERT($node, list$)

Require: A node in the set-trie *node*, a sorted list of atoms *list*.
Ensure: The multiset represented by *list* is inserted into the set-trie rooted at *node*.

if $list = []$ **then**
  $node.flag \leftarrow$ **true**
else
  for all $next\_node \in node.children$ **do**
    if $list.hd \in \{next\_node.label, \square\, next\_node.label\}$ **then** ▷ Simultaneous descent
      INSERT($next\_node$, $list.tl$)
      return
    else if $\square\, list.hd = next\_node.label$ **then** ▷ Insert $list.hd$ before $next\_node$
      $new\_node \leftarrow$ create child of *node* labeled $list.hd$
      $node.children \leftarrow node.children \setminus \{next\_node\}$
      $new\_node.children \leftarrow \{next\_node\}$
      INSERT(new_node, $list.tl$)
      return
    end if
  end for
  $next\_node \leftarrow$ create child of *node* labeled $list.hd$ ▷ No existing list share the prefix
  INSERT($next\_node$, $list.tl$)
end if

Algorithm 2 QUERY($node, list$)

Require: A node in the set-trie *node*, a sorted list of atoms *list*.
Ensure: A boolean indicating whether there exists a superset of the multiset represented by *list* in the set-trie rooted at *node*.

if $list = []$ **then**
  return true
else
  $has\_superset \leftarrow$ **false**
  for all $next\_node \in node.children$ **do**
    if $has\_superset$ **then**
      break
    else if $list.hd = next\_node.label$ **then**
      $has\_superset \leftarrow$ QUERY($next\_node$, $list.tl$) ▷ Simultaneous descent
    else if $list.hd \succeq_\square next\_node.label$ **then**
      $has\_superset \leftarrow$ QUERY($next\_node$, $list$) ▷ Skip elements in set-trie
    end if
  end for
  return $has\_superset$
end if

Algorithm 3 Saturation-based proof search with forward subsumption checking

Require: An initial state $\mathcal{S}$, a set of laws $\mathcal{I}$.
Ensure: A boolean indicating whether $\mathcal{S}$ is consistent.

$\mathbb{S} \leftarrow \{\mathcal{S}\}$ ▷ Set of existing states, implemented by set-trie
$\mathbb{A} \leftarrow \{\mathcal{S}\}$ ▷ Set of active states
while $\mathbb{A} \neq \emptyset$ **do**
  pick $\mathcal{C}$ from $\mathbb{A}$ ▷ Current state
  $\mathbb{A} \leftarrow \mathbb{A} \setminus \{\mathcal{C}\}$
  if $\bot \in \mathcal{C}$ **then**
    return false
  end if
  for all $\mathcal{L} \multimap\!\!* \mathcal{R} \in \mathcal{I}$ **do**
    if $\mathcal{L} \subseteq \mathcal{C}$ **then**
      $\mathcal{N} \leftarrow (\mathcal{C} \setminus \mathcal{L}) \cup \mathcal{R}$ ▷ New state
      if ! QUERY$(\mathcal{N}, \mathbb{S})$ **then**
        INSERT$(\mathcal{N}, \mathbb{S})$
        $\mathbb{A} \leftarrow \mathbb{A} \cup \{\mathcal{N}\}$
      end if
    end if
  end for
end while
return true

application succeeds, the decision procedure checks whether the matched premises $\mathcal{L}$ was obtained solely from $\Box$-atoms. If so, it adds $\Box\mathcal{R}$ to the state instead of $\mathcal{R}$.

We note that $\Box\,\mathcal{L}$ is never consumed, due to the inference rule $\Box$-DUP: $\Box\,\mathcal{L} \vdash \Box\,\mathcal{L} * \Box\,\mathcal{L}$. In the implementation, since states are stored in multisets, this reasoning step is realised by setting the arithmetic rule $\infty - \infty = \infty$ during multiset subtraction.

3.4 Decidability of the Fragment

In this fragment, the set of atomic formulae is fixed and finite. Consequently, it is possible to give every atomic formulae an index, and represent each state as a vector of which each element corresponds to the multiplicity of a specific atomic formula in the multiset. If we ignore persistence modality, those numbers lie in $\mathbb{N}$, and entailment problems in our system can be encoded by the reachability problem of vector addition systems[26], a well-known decidable problem.

When persistence modality is taken into account, some components of the vector may take the value ∞. Due to the non-standard substraction rule in our system $\infty - \infty = \infty$, naively treating ∞ arithmetically leads to difficulties. However, we can avoid performing $\infty - \infty = \infty$ within a vector addition system with states.

Formally, a vector addition system with states consists of a finite set of states together with a set of transitions, each transition specifying a change of state and an update on the vector. The reachability problem in this system is: given an initial state and vector, whether there exists a sequence of transitions leading to a target state and vector. This problem is also known to be decidable.

We exploit the states to distinguish between components that are finite and infinite. The transitions are defined so that whenever a state has infinite component, the corresponding coor-

dinates of the transition vector become zero. This amounts to introducing specialized transition rules that ignore substraction of infinite coordinates.

4 Handling Equality Theory and Quantification

This section presents how to extend our system to handle equality theory and quantification. In Iris, laws typically make use of universal and existential quantifiers, and often contain propositions such as (dis)equalities, which we call pure facts. Given a state $\mathcal{S}$, we denote by $Pure(\mathcal{S})$ the state consisting of pure facts in $\mathcal{S}$, and $Pred(\mathcal{S})$ the state consisting of other elements in $\mathcal{S}$. Let P range over Pred, $\mathbf{x}$ range over Var, t, t' range over Term, and $\mathbf{t}$ range over Term*. We extend the gammar definition.

$$\begin{aligned} \text{State} &:= \text{Atom} \mid \text{Atom} * \text{State} \\ \text{Atom} &:= \bot \mid \top \mid t \simeq t' \mid t \not\simeq t' \mid P(\mathbf{t}) \mid \square\,\text{Atom} \\ \text{Law} &:= \forall \mathbf{x}.(\text{State} \mathrel{-\!\!*} \text{State}) \end{aligned}$$

In this definition, a state may contain free variables. They are interpreted as existential quantifiers over the state. For example, let x, y, z range over variables, the state $x \not\simeq y * P(a, x)$ is interpreted as $\exists(x, y).\ x \not\simeq y * P(a, x)$, and the law $\forall(x, y).\ P(x, y) \mathrel{-\!\!*} (z \simeq x * Q(z, y))$ is interpreted as $\forall(x, y).\ P(x, y) \mathrel{-\!\!*} (\exists z.\ z \simeq x * Q(z, y))$.

Both equality theory and quantification have a similar impact on the proof search algorithm, i.e. they transform the multiset operations used in previous algorithms into more sophisticated operations involving the relations defined in Section 2.2.

4.1 Reasoning with (Dis)Equalities

Law application In the previous inference rule, we implicitly relied on syntactic equality by using the same symbol $\mathcal{L}$ in both the state and the premise of a law.

$$\frac{(\mathcal{L} \mathrel{-\!\!*} \mathcal{R}) \in \mathcal{I}}{\mathcal{M} * \mathcal{L} \vdash_{\mathcal{I}} \mathcal{M} * \mathcal{R}} \quad \textsc{App-Law-Frame}$$

With equality theory, there may exist pure facts in the state, and in the premise of laws. Applying a law $\mathcal{L} \mathrel{-\!\!*} \mathcal{R}$ in a state $\mathcal{S}$ requires $Pure(\mathcal{L})$ to be derivable from $Pure(\mathcal{S})$, and the existence of a sub-multiset of $Pred(\mathcal{S})$ equivalent to $Pred(\mathcal{L})$ modulo $Pure(\mathcal{S})$. The inference rule under equality theory can be written as:

$$\frac{(\mathcal{L} \mathrel{-\!\!*} \mathcal{R}) \in \mathcal{I} \qquad \vdash_{Pure(\mathcal{M})} Pure(\mathcal{L}) \qquad \mathcal{L}' \equiv_{Pure(\mathcal{M})} Pred(\mathcal{L})}{\mathcal{M} * \mathcal{L}' \vdash_{\mathcal{I}} \mathcal{M} * \mathcal{R}} \quad \textsc{App-Law-Frame-Eq}$$

Deciding the validity of $\equiv_{\mathcal{E}}$ relations reduces to solving $\vdash_{\mathcal{E}}$ entailments. In our implementation, such entailments are generated as the proof search proceeds. They are discharged using the Z3 SMT solver, which implements a sound and complete decision procedure for the theory of equality[15].

Subsumption checking The notion of subsumption must also take pure facts into account. Intuitively, $\mathcal{S}$ is subsumed by $\mathcal{S}'$ if $Pure(\mathcal{S}')$ entails $Pure(\mathcal{S})$, and $Pred(\mathcal{S})$ is contained in $Pred(\mathcal{S}')$. Formally, we have:

$$\mathcal{S} \sqsubset \mathcal{S}' \quad \text{iff} \quad \vdash_{Pure(\mathcal{S}')} Pure(\mathcal{S}) \text{ and } Pred(\mathcal{S}) \subseteq Pred(\mathcal{S}')$$

Since checking the entailment relation $\vdash_{Pure(\mathcal{S}')} Pure(\mathcal{S})$ requires invoking an external solver, in our implementation, we first check the multiset relation as a pre-filter. Only if this check succeeds do we query the solver. This can be implemented by a slight modification of the set-trie algorithms defined eariler. Specifically, we replace $node.flag$ with the set of pure facts of each state terminating at $node$. Algorithm 4 and Algorithm 5 present the modified version of *insert* and *query*.

Algorithm 4 INSERT($node, list, pure$)

Require: A node in the set-trie $node$, a sorted list of atoms $list$, a set of (dis)equalities $pure$.
Ensure: The state represented by $list$ and $pure$ is inserted into the set-trie rooted at $node$.

```
1: if list = [] then
2:     node.pure_set ← node.pure_set ∪ {pure}
3: else
4:     for all next_node ∈ node.children do
5:         if list.hd ∈ {next_node.label, □ next_node.label} then        ▷ Simultaneous descent
6:             INSERT(next_node, list.tl, pure)
7:             return
8:         else if □ list.hd = next_node.label then        ▷ Insert list.hd before next_node
9:             new_node ← create child of node labeled list.hd
10:            node.children ← node.children \ {next_node}
11:            new_node.children ← {next_node}
12:            INSERT(new_node, list.tl, pure)
13:            return
14:        end if
15:    end for
16:    next_node ← create child of node labeled list.hd        ▷ No existing list share the prefix
17:    INSERT(next_node, list.tl, pure)
18: end if
```

4.2 Instantiation Techniques

Laws in this fragment are persistent, universally quantified separating implications, whose universal quantifiers must first be instantiated before they can be applied. Taking this instantiation step into account, the inference rule could be written as:

$$\frac{\forall \mathbf{x}.\ (\mathcal{L} \mathrel{-\!\!*} \mathcal{R}) \in \mathcal{I} \qquad \vdash_{Pure(\mathcal{M})} Pure(\mathcal{L}\sigma) \qquad \mathcal{L}' \equiv_{Pure(\mathcal{M})} Pred(\mathcal{L}\sigma)}{\mathcal{M} * \mathcal{L}' \vdash_{\mathcal{I}} \mathcal{M} * \mathcal{R}\sigma} \quad \text{APP-LAW-FRAME-INS}$$

where σ instantiates variables in $\mathbf{x}$.

The unification subproblem Applying the inference rule can be reduced to computing the substitution σ. We denote by $\mathcal{S}$ the current state, and $\mathcal{L}$ the premise of a law containing variables to be instantiated. σ can be computed by unifying $Pred(\mathcal{L})$ with the subsets of $Pred(\mathcal{S})$ modulo $Pure(\mathcal{S})$. Another formulation of the problem is to decide:

$$gen_{\subseteq_{Pure(\mathcal{S})}}(Pred(\mathcal{L}), Pred(\mathcal{S}))$$

We give in Algorithm 6 the unification decision procedure. We instantiate atoms in $Pred(\mathcal{L})$ one by one. We maintain a substitution σ. Each step chooses an atom $A \in Pred(\mathcal{L})\sigma$, and tries

Algorithm 5 QUERY($node, list, pure$)

Require: A node in the set-trie $node$, a sorted list of atoms $list$, a set of (dis)equalities $pure$.
Ensure: A boolean indicating whether there exists a state subsuming the state represented by $list$ and $pure$ in the set-trie rooted at $node$.

```
1: if list = [] then
2:     for all desc_node descendants of node do
3:         for all P ∈ desc_node.pure_set do
4:             if ⊢_P pure then
5:                 return true
6:             end if
7:         end for
8:     end for
9:     return false
10: else
11:     has_superstate ← false
12:     for all next_node ∈ node.children do
13:         if has_superstate then
14:             break
15:         else if list.hd = next_node.label then
16:             has_superstate ← QUERY(next_node, list.tl, pure)        ▷ Simultaneous descent
17:         else if list.hd ≥_□ next_node.label then
18:             has_superstate ← QUERY(next_node, list, pure)           ▷ Skip elements in set-trie
19:         end if
20:     end for
21:     return has_superstate
22: end if
```

to unify A with every atom $B \in Pred(\mathcal{S})$. Once success, we compose σ with the unifier of A and B, substract B from $Pred(\mathcal{S})$, and recurse on remaining atoms of $Pred(\mathcal{L})\sigma$.

Algorithm 6 Unification modulo (dis)equalities

Require: A state $\mathcal{S}$, the premise of a law $\mathcal{L}$.
Ensure: Either FAIL, or a pair $(\sigma, \mathcal{L}')$ where σ is a substitution and $\mathcal{L}' \subseteq Pred(\mathcal{S})$ such that $\vdash_{Pure(\mathcal{S})} Pure(\mathcal{L}\sigma)$ and $\mathcal{L}' \equiv_{Pure(\mathcal{S})} Pred(\mathcal{L})\sigma$.

```
1:  x ← free variables of L
2:  E_S ← Pure(S)
3:  E_L ← Pure(L)
4:  return UNIFY({}, ∅, Pred(S), Pred(L))
5:  function UNIFY(σ, L', S, L)
6:      if L = ∅ then
7:          if ∀x ∈ x. σ(x) ≠ x and ⊢_{E_S} (E_L)σ then
8:              return (σ, L')
9:          else
10:             return FAIL
11:         end if
12:     end if
13:     A ← next(L)
14:     B ← { B ∈ S | gen_{E_S}(A, B) }
15:     for all B ∈ B do
16:         σ' ← mgu_{E_S}(A, B)
17:         σ̂ ← σ ∪ σ'
18:         r ← UNIFY(σ̂, L' ∪ {B}, S \ {B}, (L \ {A})σ')
19:         if r ≠ FAIL then return r
20:         end if
21:     end for
22:     return FAIL
23: end function
```

The worst-case complexity is exponential in $|Pred(\mathcal{L})|$. One standard heuristic to optimize the performance is "most-constrained-first", namely match atoms with the largest number of ground arguments first. We denote by *next* the function selecting such atom from a state. We also denote by *mgu* the function computing the most general unifier of two atoms.

Term indexing The algorithm needs to retrieve from a state all elements that are instances of a given query atom, as seen in line 14 of Algorithm 6. To get a candidate set effectively, we first perform a pre-filtering by indexing on predicate symbols. Concretely, in the implementation we represent the state as a two-level map: the outer map goes from predicate symbols to inner maps, and each inner map goes from argument tuples to multiplicities. This structure is illustrated below for the state: $P(a,b) * P(a,c) * P(a,c) * \square\, Q(x,b) * Q(a,y)$.

$$\begin{cases} P \mapsto \begin{cases} (a,b) \mapsto 1 \\ (a,c) \mapsto 2 \end{cases} \\ Q \mapsto \begin{cases} (x,b) \mapsto \infty \\ (a,y) \mapsto 1 \end{cases} \end{cases}$$

4.3 Decidability of the Fragment

With the addition of equality theory and quantification, this fragment of separation logic becomes undecidable, since it can axiomatise the two-record-field points-to predicate $l \mapsto (v_1, v_2)$ mentioned in (Section 1), whose presence along with (dis)equalities and first-order quantifiers results in undecidability[11]. By axiomatising, we mean this predicate can be defined entirely within the logic by a finite set of separating-implication laws, without introducing it as a primitive.

Consider the semantics of the two-record-field points-to predicate:

$$s, h \vDash l \mapsto (v_1, v_2) \;\text{ iff }\; h(l) = (v_1, v_2)$$

This is an affine predicate, and can be axiomatized with the usual law of points-to predicates:

$$\forall(l, v_1, v_2, v_3, v_4).\; l \mapsto (v_1, v_2) * l \mapsto (v_3, v_4) \mathrel{-\!\!*} \bot$$

5 Handling Disjunction

This section presents how to add support for disjunction. Disjunction specifies different possible shapes that a heap may have. Below shows a formula with disjunction that our system is intended to support:

$$P_1 * P_1 * (P_3 \vee P_4 \vee P_5) * ((P_6 * P_7 * P_8) \vee (P_9 * P_{10}))$$

Let P range over Pred, $\mathbf{x}$ range over Var, t, t' range over Term, and $\mathbf{t}$ range over Term^*. We extend the grammar definition as follows:

$$\begin{aligned}
\mathrm{Disj} &:= \mathrm{Conj} \vee \mathrm{Conj} \mid \mathrm{Conj} \vee \mathrm{Disj} \\
\mathrm{Conj} &:= \mathrm{Atom} \mid \mathrm{Atom} * \mathrm{Conj} \\
\mathrm{State} &:= \mathrm{Conj} \mid \mathrm{Conj} * \mathrm{Disj} \\
\mathrm{Atom} &:= \bot \mid \top \mid t \simeq t' \mid t \not\simeq t' \mid P(\mathbf{t}) \mid \Box\,\mathrm{Atom} \\
\mathrm{Law} &:= \forall \mathbf{x}.(\mathrm{State} \mathrel{-\!\!*} \mathrm{State})
\end{aligned}$$

As a simplification, in the following, we omit disjunctions in the premise of laws. This does not influence the expressive power of the logic, because of the following equivalence:

$$\Box\,(\mathcal{L} * (\mathcal{L}_1 \vee \mathcal{L}_2) \mathrel{-\!\!*} \mathcal{R}) \equiv \Box\,(\mathcal{L} * \mathcal{L}_1 \mathrel{-\!\!*} \mathcal{R}) * \Box\,(\mathcal{L} * \mathcal{L}_2 \mathrel{-\!\!*} \mathcal{R})$$

For simplicity, we assume there exists at most one Disj inside every state, we also take the logic fragment defined in Section 3 with disjunctions, to avoid repetitions on quantification and equality theory.

5.1 Proof Search with Disjunction

The addition of disjunction provides the following inference rule. Let $\mathcal{S}, \mathcal{T}$ range over states, and let $\mathcal{C}_1, \mathcal{C}_2$ range over Conj, we have

$$\frac{\mathcal{S} * \mathcal{C}_1 \vdash \mathcal{T} \qquad \mathcal{S} * \mathcal{C}_2 \vdash \mathcal{T}}{\mathcal{S} * (\mathcal{C}_1 \vee \mathcal{C}_2) \vdash \mathcal{T}} \;\vee\text{-Elim}$$

It says that, to derive a goal from a disjunction, we must be able to derive it from each branch.

If we consider reachability of $\bot$, a state with a disjunction $\mathcal{S} * (\mathcal{C}_1 \vee \mathcal{C}_2)$ can reach $\bot$ if and only if both branches do so. We organize proof search as an and-or graph whose nodes represent states.

$$\frac{\mathcal{S} * \mathcal{C}_1 \;\vdash^*\; \bot \qquad \mathcal{S} * \mathcal{C}_2 \;\vdash^*\; \bot}{\mathcal{S} * (\mathcal{C}_1 \vee \mathcal{C}_2) \;\vdash^*\; \bot}$$

And-or graph We consider a directed graph (V, E) where there are two kinds of nodes: $\wedge$-node and $\vee$-node. In the and-or graph used for proof search, each node is labeled by a state, and tagged with $tag \in \{\bot, \top\}$, interpreted as standard boolean values. We denote by $succ(u) = \{v \mid (u, v) \in E\}$ and $pred(u) = \{v \mid (v, u) \in E\}$ the set of immediate successors and predecessors of a node u, we denote by $u.state$ and $u.tag$ the state and the tag associated to the node u.

An and-or graph can be used to represent relations on boolean variables. Each node u in the graph satisfies:

$$u \text{ is } \wedge\text{−node} \quad \text{iff} \quad u.tag = \bigwedge_{u' \in succ(u)} u'.tag$$

$$u \text{ is } \vee\text{−node} \quad \text{iff} \quad u.tag = \bigvee_{u' \in succ(u)} u'.tag$$

In our case, the property of interest is reachability of $\bot$. For each node u, we define the value of the tag as:

$$u.tag = \bot \quad \text{iff} \quad u.state \vdash^* \bot$$

We can say that $\wedge$-node collects obligations that must all be fulfilled in order to reach $\bot$, whereas an $\vee$-node represents the proof steps where reaching $\bot$ on any successor suffices.

Graph exploration The shape of the graph depends on the proof rule being applied.

- If we apply the $\vee$-Elim rule on a state $\mathcal{S} * (\mathcal{C}_1 \vee \mathcal{C}_2)$, then the corresponding node is an $\wedge$-node labeled by $\mathcal{S} * (\mathcal{C}_1 \vee \mathcal{C}_2)$, with two successors labeled by $\mathcal{S} * \mathcal{C}_1$ and $\mathcal{S} * \mathcal{C}_2$.
- If we apply the App-Law rule on a state $\mathcal{S}$, the corresponding node is an $\vee$-node labeled by $\mathcal{S}$. Each applicable law $\Box(\mathcal{L} \mathrel{-\!\!*} \mathcal{R})$ that matches a sub-state of $\mathcal{S}$ gives rise to one successor, namely the state $(\mathcal{S} \setminus \mathcal{L}) \cup \mathcal{R}$.

Importantly, both rules may be applicable to the same state. In this case, we duplicate the node into two copies: an $\wedge$-node for $\vee$-Elim, and an $\vee$-node for App-Law. Both nodes are labeled by the same state, but connected differently to their successors. This ensures that the search graph captures the correct relation of every successor.

For example, consider the set of laws $\mathcal{I} = \{\Box(A \mathrel{-\!\!*} B),\ \Box(C \mathrel{-\!\!*} D)\}$ and the initial state $\mathcal{S} = A * C * (E \vee F)$. We can proceed in two ways:

- $\vee$-Elim: it generates an $\wedge$-node with two successors, $\mathcal{S}_1 = \{A{*}C{*}E\}$ and $\mathcal{S}_2 = \{A{*}C{*}F\}$.
- App-Law: it generates an $\vee$-node with two successors, $\mathcal{S}_3 = \{B * C * (E \vee F)\}$ using $\Box\,(A \mathrel{-\!\!*} B)$ and $\mathcal{S}_4 = \{A * D * (E \vee F)\}$ using $\Box\,(C \mathrel{-\!\!*} D)$.

The exploration of the search graph is illustrated in Figure 3.

In the implementation, these two nodes are merged together to produce a more compact graph representation. Figure 4 shows the entire search graph generated by the input above.

Tag propagation Every node has tag $\bot$ by default. Once a node derives $\bot$, its tag is assigned the value $\top$, and this information must be propagated upwards. We present this procedure in Algorithm 7.

In Algorithm 7, line 6 may be inefficient, since it recomputes the conjunction of tags over all successor nodes each time an $\wedge$-node is visited. This can be optimized using the watched-literal technique widely used in SAT solvers[28].

The complete proof search decision procedure is given in Appendix B. We fix a constant *max_depth* specifying the maximum search depth.

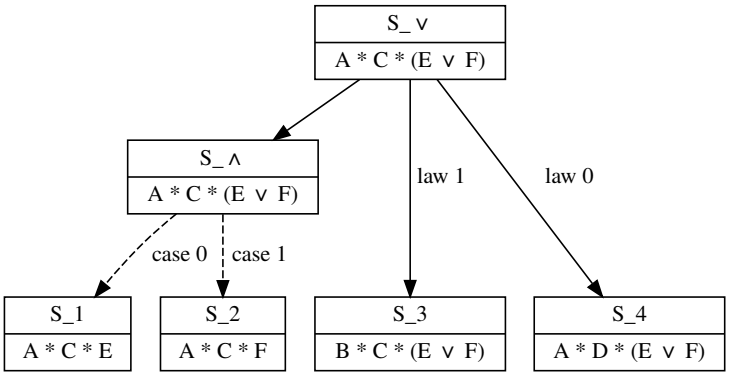

Figure 3: A local of search graph

Algorithm 7 PROPAGATE(u)

Require: A node u in the search graph that has just been tagged with $\top$.
Ensure: The search graph with updated tags propagated to predecessors of u.

```
for all v ∈ pred(u) do
    if v.tag = ⊤ then
        continue
    end if
    if v is an ∧-node then
        v.tag ← ⋀_{w∈succ(v)} w.tag
        if v.tag = ⊤ then
            PROPAGATE(v)
        end if
    else                                          ▷ v is an ∨-node
        v.tag ← ⊤
        PROPAGATE(v)
    end if
end for
```

5.2 Further Pruning the Search Graph

The existence of multiple search branches changes the notion of redundant states. Not only should we consider the entailment relation between states, but we should also consider the branches the states belong to. For example, consider the state $\mathcal{S} = A \vee (A * B)$. It generates two successor states $\mathcal{S}_1 = A$ and $\mathcal{S}_2 = A * B$ via $\vee$-ELIM. Following the original definition of subsumption, we have $\mathcal{S}_1 \sqsubset \mathcal{S}_2$. However, $\mathcal{S}_1$ cannot be discarded, since it belongs to a different disjunctive branch than $\mathcal{S}_2$.

Disjunctive branch Let $G = (V, E)$ be an and-or graph. In the setting of saturation-based proof search, there exists a unique node r in G with in-degree 0. A disjunctive branch $B \subseteq V$ is a set of nodes such that:

- B is non-empty. Either $r \in B$, or there exists an immediate successor s of an $\wedge$-node such that $s \in B$.
- If $u \in B$ and u is an $\vee$-node, then $succ(u) \subseteq B$.

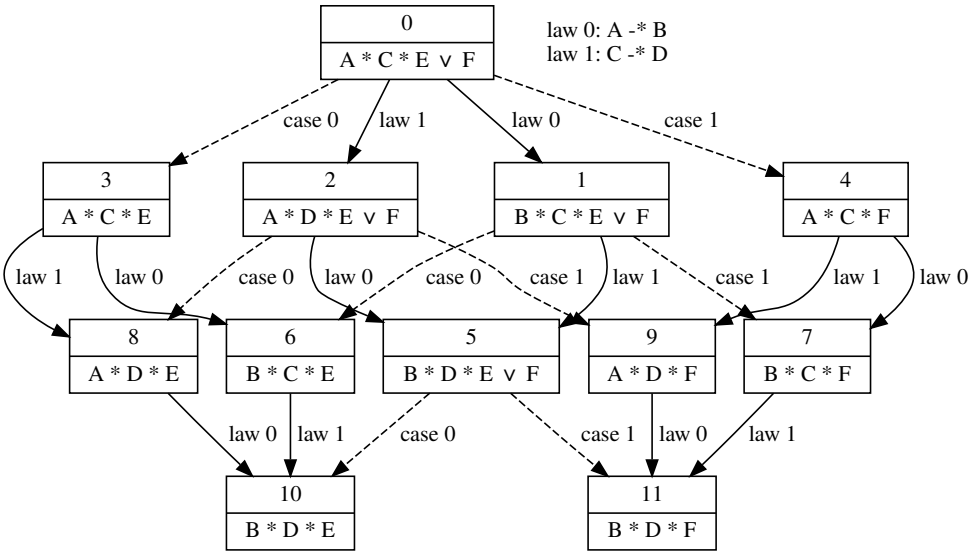

Figure 4: The entire search graph

- If $u \in B$ and u is an $\wedge$-node, $succ(u) \cap B = \emptyset$. u is the exit point of B.
- B is a connected component of G.

Following this definition, each disjunctive branch is a maximal region of the graph delimited by $\wedge$-nodes.

Figure 5 gives an illustration on the concept of disjunctive branches. There are three disjunctive branches in this search graph, namely $\{0, 1, 2\}$, $\{1, 3, 5\}$ and $\{3, 4, 5, 6, 7\}$.

Generalized forward subsumption checking With the notion of disjunctive branches, we can refine the definition of subsumption. A state $\mathcal{S}$ is subsumed by another state $\mathcal{S}'$ if $\mathcal{S}$ is a subset of $\mathcal{S}'$, and the set of disjunctive branches in which $\mathcal{S}$ occurs is contained in the set of disjunctive branches in which $\mathcal{S}'$ occurs. Formally, let $br(\mathcal{S})$ denote the set of branches containing $\mathcal{S}$, we have

$$\mathcal{S} \sqsubset \mathcal{S}' \quad \text{iff} \quad \mathcal{S} \subseteq \mathcal{S}' \text{ and } br(\mathcal{S}) \subseteq br(\mathcal{S}')$$

Due to space limitations, we do not present algorithms maintaining the set of disjunctive branches of each node.

6 Conclusion and Discussion

In this report, I have presented the design and implementation of an automated deduction tool for a fragment of the Iris separation logic framework. Due to the limited duration of the internship, the current system remains prototype. Although it is not intended to become a full-fledged automated theorem prover for separation logic, some topics remain to be further explored to improve its utility.

The first topic concerns the performance. Automated deduction tools generally handle a large and expanding body of knowledge, which inevitably slows down the program. In our system,

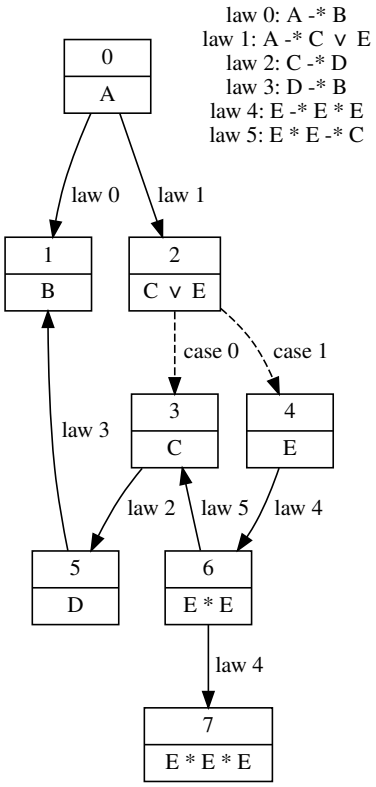

Figure 5: Disjunctive branches

forward subsumption checking accounts for the majority of the runtime. This corresponds to the remark by Larry Wos[40]:

> ⋯, after a few CPU minutes of use or the retention of a few thousand conclusions, a reasoning program typically made deductions at less than 1% of its ability at the beginning of a run.

From my perspective, there are two main directions to tackle performance degradation in our system.

Efficient data structures In the current implementation, propositions are stored in balanced binary search trees, due to their efficiency in set operations frequent in proof search. However, this choice may not be optimal for term indexing for two reasons:

- When multiple terms need to be indexed, lookups can only be performed sequentially, whereas some term-indexing data structures support efficient multi-term lookups.
- Balanced binary search trees organize terms according to a total ordering, which does not exploit the structural properties shared by terms as many term-indexing techniques do. For example, trie-based term-indexing structures allow direct access to relevant subsets of terms sharing certain structural properties. This issue gets worse when all matching terms must be retrieved: Binary search trees revert to the sequential lookups, whereas

some term-indexing data structures can stop traversal at an intermediate level and return an entire set of matching terms at once.

State-of-the-art automated deduction tools address such issues very carefully by crafting data structures specific for each task. As described in the *Handbook of Automated Reasoning*[38], the Vampire theorem prover[35] uses partially adaptive code trees[34] for forward subsumption, and path indexing trees[32] for backward subsumption. Terms in kept clauses are stored as perfectly shared terms, while in temporary clauses as flatterm representation[14]. In our case, finding a design that reconciles frequent set operations with efficient term indexing and subsumption checking remains an unresolved problem.

Clause management Modern clause-learning SAT solvers typically incorporate useful clause-forgetting mechanisms to control complexity[23, 24]. In the context of automated deduction, such selective forgetting of derived information becomes more complex and diverse. Subsumption checking can be regarded as one form of clause management, other strategies include tautology deletion, clause deletion based on age, weight or size[33], and normalization techniques such as demodulation[41], condensation, etc. Our tool only implements a basic version of forward subsumption, and determining an effective clause management strategy for our tool will require empirical evaluation on realistic datasets.

The second topic is the extension of the logical language supported by the tool. In this report, how to implement the persistence modality from Iris is explained. Together with other separation logic connectives, they can already encode parts of the Iris library such as `gen_heap.v`[4] and `token.v`[5]. However, other constructs prevalent in Iris such as invariants, fancy updates, and ghost states are not included. Adding support for these would enable the encoding of a much larger portion of the Iris library, including newly developed ones.

Another step forward would be to add support for background theories, such as uninterpreted functions and linear integer arithmetic. Iris also makes extensive use of the points-to predicate $l \overset{q}{\mapsto} v$ with a fractional permission q[20]. These rational numbers range from 0 to 1 and can be combined or split via arithmetic operators. Our tool would thus benefit from supporting bounded rational numbers. Another example is set theory, which also appears in some parts of the Iris library.

[4] https://gitlab.mpi-sws.org/iris/iris/-/blob/master/iris/base_logic/lib/gen_heap.v
[5] https://gitlab.mpi-sws.org/iris/iris/-/blob/master/iris/base_logic/lib/token.v

References

[1] The coq development for iris. `https://gitlab.mpi-sws.org/iris/iris/`, 2015.

[2] Competition of solvers for separation logic. `https://sl-comp.github.io/`, 2019.

[3] Baby: Fast sets and maps based on balanced binary search trees. `https://pauillac.inria.fr/~fpottier/baby/doc/baby/`, 2024.

[4] The rocq prover. `https://rocq-prover.org/about`, 2025.

[5] A. W. Appel. Verified software toolchain. In G. Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] A. W. Appel, L. Beringer, and C. Qinxiang. *Verifiable C*, volume 5 of *Software Foundations*. Electronic textbook, 2023. Version 1.2.2, `http://softwarefoundations.cis.upenn.edu`.

[7] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, page 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS'04, page 97–109, Berlin, Heidelberg, 2004. Springer-Verlag.

[9] G. Blelloch, D. Ferizovic, and Y. Sun. Joinable parallel balanced binary trees. *ACM Trans. Parallel Comput.*, 9(2), Apr. 2022.

[10] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, LICS '07, page 366–378, USA, 2007. IEEE Computer Society.

[11] C. Calcagno, H. Yang, and P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In R. Hariharan, V. Vinay, and M. Mukund, editors, *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, pages 108–119, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[12] T. Chajed, J. Tassarotti, M. Theng, R. Jung, M. F. Kaashoek, and N. Zeldovich. Gojournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 423–439. USENIX Association, July 2021.

[13] Y.-S. Chang, R. Jung, U. Sharma, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 871–886, Boston, MA, July 2023. USENIX Association.

[14] J. Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10(1):95–113, February 1993.

[15] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[16] P. E. de Vilhena and F. Pottier. Verifying an effect-handler-based define-by-run reverse-mode ad library. *Logical Methods in Computer Science*, Volume 19, Issue 4, Oct. 2023.

[17] J.-C. Filliâtre and S. Conchon. Type-safe modular hash-consing. In *Proceedings of the 2006 Workshop on ML*, ML '06, page 12–19, New York, NY, USA, 2006. Association for Computing Machinery.

[18] T. Hance, J. Howell, O. Padon, and B. Parno. Leaf: Modularity for temporary sharing in separation logic. *Proc. ACM Program. Lang.*, 7(OOPSLA2), Oct. 2023.

[19] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), Dec. 2017.

[20] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28(e20), 2018.

[21] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 637–650, New York, NY, USA, 2015. Association for Computing Machinery.

[22] R. Krebbers, A. Timany, and L. Birkedal. Interactive proofs in higher-order concurrent separation logic. *SIGPLAN Not.*, 52(1):205–217, Jan. 2017.

[23] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[24] T. Krüger, J.-H. Lorenz, and F. Wörz. Too much information: Why cdcl solvers need to forget learned clauses. *PLOS ONE*, 17(8):e0272967, Aug. 2022.

[25] M. Legoupil, J. Rousseau, A. L. Georges, J. Pichon-Pharabod, and L. Birkedal. Iris-mswasm: Elucidating and mechanising the security invariants of memory-safe webassembly. *Proc. ACM Program. Lang.*, 8(OOPSLA2), Oct. 2024.

[26] J. Leroux. Vector addition system reachability problem: a short self-contained proof. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 307–316, New York, NY, USA, 2011. Association for Computing Machinery.

[27] A. Moine, A. Charguéraud, and F. Pottier. Will it fit? verifying heap space bounds of concurrent programs under garbage collection. *ACM Trans. Program. Lang. Syst.*, 47(1), Apr. 2025.

[28] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, page 530–535, New York, NY, USA, 2001. Association for Computing Machinery.

[29] I. Mulder and R. Krebbers. Proof automation for linearizability in separation logic. *Proc. ACM Program. Lang.*, 7(OOPSLA1), Apr. 2023.

[30] P. O'Hearn. Separation logic. *Commun. ACM*, 62(2):86–95, Jan. 2019.

[31] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[32] R. Ramesh, I. Ramakrishnan, and D. Warren. Automata-driven indexing of prolog clauses. page 281 – 291, 1990. Cited by: 18.

[33] M. Rawson and G. Reger. Old or heavy? decaying gracefully with age/weight shapes. In *Automated Deduction – CADE 27: 27th International Conference on Automated Deduction, Natal, Brazil, August 27–30, 2019, Proceedings*, page 462–476, Berlin, Heidelberg, 2019. Springer-Verlag.

[34] A. Riazanov and A. Voronkov. Partially adaptive code trees. In *Proceedings of the European Workshop on Logics in Artificial Intelligence*, JELIA '00, page 209–223, Berlin, Heidelberg, 2000. Springer-Verlag.

[35] A. Riazanov and A. Voronkov. The design and implementation of vampire. *AI Commun.*, 15(2,3):91–110, Aug. 2002.

[36] I. Savnik, M. Akulich, M. Krnc, and R. Škrekovski. Data structure set-trie for storing and querying sets: Theoretical and empirical analysis. *PLOS ONE*, 16(2):1–38, 02 2021.

[37] R. Seassau, I. Yoon, J.-M. Madiot, and F. Pottier. Formal semantics and program logics for a fragment of ocaml. *Proc. ACM Program. Lang.*, 9(ICFP), Aug. 2025.

[38] R. Sekar, I. Ramakrishnan, and A. Voronkov. Chapter 26 - term indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 1853–1964. North-Holland, Amsterdam, 2001.

[39] S. F. Vindum, D. Frumin, and L. Birkedal. Mechanized verification of a fine-grained concurrent queue from meta's folly library. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 100–115, New York, NY, USA, 2022. Association for Computing Machinery.

[40] L. Wos. Note on mccune's article on discrimination trees. *Journal of Automated Reasoning*, 9(2):145–146, October 1992.

[41] L. Wos, G. A. Robinson, D. F. Carson, and L. Shalla. The concept of demodulation in theorem proving. *J. ACM*, 14(4):698–709, Oct. 1967.

A Ghost Theory Example

The ghost theory handled by the tool is:

A	
$* (A \twoheadrightarrow B \vee C)$	law0
$* (B \twoheadrightarrow D)$	law1
$* (C \twoheadrightarrow E)$	law2
$* (D \twoheadrightarrow \bot)$	law3
$* (E \twoheadrightarrow H * (F \vee G))$	law4
$* (F \twoheadrightarrow \bot)$	law5
$* (G \twoheadrightarrow E)$	law6
$* (H * H \twoheadrightarrow \bot)$	law7

Each node in the proof tree is labeled with an index and a formula indicating the current state. The root has the index zero, and it carries the initial heap predicate A. A node is green if the formula it carries is inconsistent. Here, every node is inconsistent.

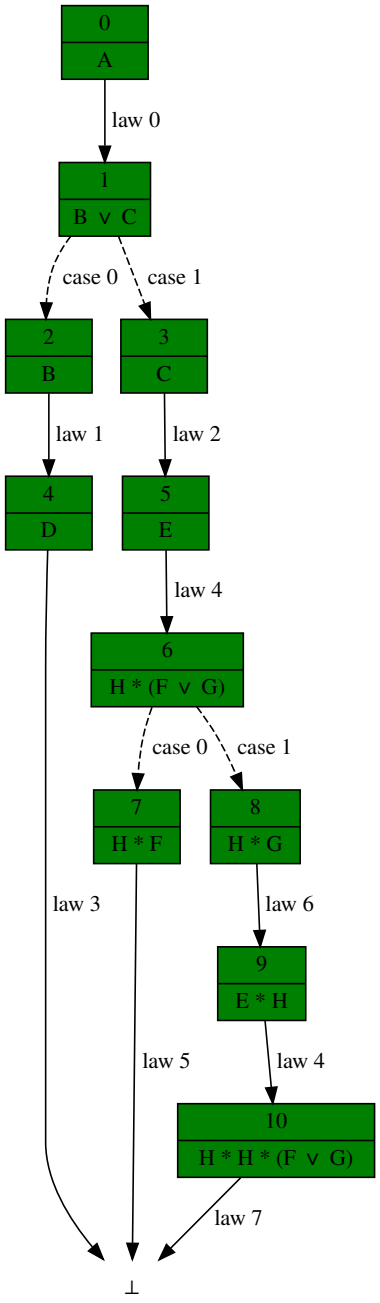

Figure 6: Inconsistency proof for a ghost theory

B Saturation-Based Proof Search with Disjunction

Algorithm 8 Saturation-based proof search with disjunction

Require: An initial state $\mathcal{S}$, a set of laws $\mathcal{I}$.
Ensure: A boolean indicating whether $\mathcal{S}$ is consistent.

$\mathcal{K} \leftarrow$ create new $\vee$-node labeled with $\mathcal{S}$
$\mathcal{K}.depth \leftarrow 0$
$\mathcal{K}.tag \leftarrow \bot$
$\mathbb{A} \leftarrow \{\mathcal{K}\}$ ▷ Set of active nodes
while $\mathcal{K}.tag = \bot$ **and** $\mathbb{A} \neq \emptyset$ **do**
  pick $\mathcal{C}$ from $\mathbb{A}$ ▷ Current node
  $\mathbb{A} \leftarrow \mathbb{A} \setminus \{\mathcal{C}\}$
  if $\mathcal{K}$.depth $\geq max\_depth$ **then**
    break
  end if
  if $\bot \in \mathcal{C}.state$ **then**
    $\mathcal{C}.tag \leftarrow \top$
    PROPAGATE($\mathcal{C}$)
  else
    for all $\mathcal{L} \nrightarrow \mathcal{R} \in \mathcal{I}$ **do**
      if $\mathcal{L} \subseteq \mathcal{C}.state$ **then**
        $\mathcal{N} \leftarrow$ create new $\vee$-node labeled with $(\mathcal{C}.state \setminus \mathcal{L}) \cup \mathcal{R}$
        $\mathcal{N}.depth \leftarrow \mathcal{C}.depth + 1$
        $\mathcal{N}.tag \leftarrow \bot$ ▷ New node
        $\mathcal{N}.pred \leftarrow \{\mathcal{C}\}$
        $\mathcal{C}.succ \leftarrow \mathcal{C}.succ \cup \{\mathcal{N}\}$
        $\mathbb{A} \leftarrow \mathbb{A} \cup \{\mathcal{N}\}$
      end if
    end for
    if a disjunctive part $D = (D_1 \vee D_2 \vee \cdots \vee D_n) \in \mathcal{C}$.state **then**
      $\mathcal{C}_\wedge \leftarrow$ create new $\wedge$-node labeled with $\mathcal{C}$.state
      $\mathcal{C}_\wedge.depth \leftarrow \mathcal{C}.depth$
      $\mathcal{C}_\wedge.tag \leftarrow \bot$
      $\mathcal{C}_\wedge.pred \leftarrow \{\mathcal{C}\}$
      $\mathcal{C}.succ \leftarrow \mathcal{C}.succ \cup \{\mathcal{C}_\wedge\}$
      for all $D_i \in D$ **do**
        $\mathcal{N} \leftarrow$ create new $\vee$-node labeled with $(\mathcal{C}.state \setminus \mathcal{D}) \cup \mathcal{D}_i$
        $\mathcal{N}.depth \leftarrow \mathcal{C}_\wedge.depth + 1$
        $\mathcal{N}.tag \leftarrow \bot$ ▷ New node
        $\mathcal{N}.pred \leftarrow \{\mathcal{C}_\wedge\}$
        $\mathcal{C}_\wedge.succ \leftarrow \mathcal{C}_\wedge.succ \cup \{\mathcal{N}\}$
        $\mathbb{A} \leftarrow \mathbb{A} \cup \{\mathcal{N}\}$
      end for
    end if
  end if
end while
return $\neg\mathcal{K}.tag$