Proof Tactics for Assertions in Separation Logic

Zhé Hóu, David Sanán, Alwen Tiu, and Yang Liu

Nanyang Technological University

Abstract. This paper presents tactics for reasoning about the assertions of separation logic. We formalise our proof methods in Isabelle/HOL based on Klein et al.'s separation algebra library. Our methods can also be used in other separation logic frameworks that are instances of the separation algebra of Calcagno et al. The first method, *separata*, is based on an embedding of a labelled sequent calculus for abstract separation logic (ASL) by Hóu et al. The second method, *starforce*, is a refinement of separata with specialised proof search strategies to deal with separating conjunction and magic wand. We also extend our tactics to handle pointers in the heap model, giving a third method *sepointer*. Our tactics can *automatically* prove many complex formulae. Finally, we give two case studies on the application of our tactics.

1 Introduction

Separation Logic (SL) is widely used to reason about programs with pointers and mutable data structures [34]. Many tools for separation logic have emerged since its inception and some of them have proven successful in real-life applications, such as the bi-abduction based techniques used in Infer [1]. Most tools for separation logic are built for small subsets of the assertion logic, notably the symbolic heap fragment [5], and applied to verify correctness and memory safety properties of computer programs. However, when verifying concurrent programs, often there is the need to use a larger fragment of the assertion language. For instance, the Separation Logic framework in Isabelle/HOL [28] and the Iris 3.0 framework [26] both use the full set of logical connectives, along with other features. Currently the frameworks that use larger fragments of the assertion language tend to focus more on the reasoning of Hoare triples than the assertions. An exception is the Iris 3.0 framework, in which the authors developed tactics for interactive proofs. Automated tools, however, are still beyond reach for larger fragments of SL and are the future work of the Iris project [26].

We are also motivated by our own project, which aims at verifying that an execution stack, including the processor architecture, micro-kernel, and applications, is correct and secure. Similar projects are NICTA's seL4 [30] and Yale's CertiKOS [19]. In particular, we are verifying the XtratuM hypervisor which runs on a multi-core LEON3 processor. Since concurrency is important in our project, it is useful to build formal models using techniques such as rely/guarantee and separation logic, and we will use the full assertion language because logical connectives such as the "magic wand" ($-\!*$) and "septraction" ($-\circledast$) are useful in rely/guarantee reasoning [38]. We aim to build a framework in Isabelle/HOL that can provide high confidence for the verification tasks. Automatic tactics in a proof assistant are therefore highly desirable because they can minimise the overhead of translating back and forth between the proof assistant and external provers, and it is easier to integrate them with other tactics.

This paper presents automatic proof tactics for reasoning about assertions in separation logic. Although frame inference is not in our scope, our tactics can be used to reason about assertions in frame inference. The tactics are independent of the separation logic framework and the choice of proof assistant, as long as the assertion logic is an instance of Calcagno et al.'s original definition of separation algebra [11]. For demonstration purposes and for the sake of our own project, we base our implementation on the work of Klein et. al. [25], which formalises Calcagno et al.'s separation algebra and uses a shallow embedding of separation logic assertions into Isabelle/HOL formulae. At the core of our tactics lies the labelled sequent calculus LS_{PASL} of Hóu et al. [22], which is one of the few proof systems that have been shown successful in reasoning about the full language of assertions of separation logics with various flavours of semantics.

We first formalise each inference rule in LS_{PASL} as a lemma in a proof assistant, we then give a basic proof search procedure *separata* which can easily solve the formulae in previous BBI and PASL benchmarks [21, 33]. To improve the performance and automation, we develop several more advanced tactics. The widely-used separating conjunction (denoted by $*$) behaves like linear conjunction in linear logic. It often creates difficulty in proof search because one has to find the correct splitting of resources to complete the proof. Effective 'resource management' in linear logic proof search is a well-known problem and it has been studied in the literature [12, 20]. Unlike the case with linear logic, where resource is a multiset, we need to deal with a more complex structure capturing relations between heaps, and it is not clear how search techniques for linear logic [20] can be employed. We propose a new formula-driven algorithm to solve the heap partitioning problem. We also present a tactic to simplify the formula when it involves a combination of $*$ and $-*$ connectives. Finally, we extend the above tactics with inference rules [22] to handle pointers in the heap semantics.

We demonstrate that our tactics are able to prove many separation logic formulae automatically. These formulae are taken from benchmarks for BBI and abstract separation logic provers and the *sep_solve* method developed in seL4. We give a case study where we formalise Feng's semantics of actions in local rely/guarantee [17] using our extension of separation algebra, and prove some properties of the semantics using our tactics. Lastly, we show that our tactics can be easily implemented in other frameworks in case the user cannot directly use our implementation.

2 Preliminaries

This section gives an overview of Klein et al.'s formalization of Calcagno et al.'s separation algebra [25]. We extend their work with additional properties which are useful in applications. Then we briefly revisit the labelled sequent calculus LS_{PASL}.

2.1 Separation Algebra

A separation algebra [11] is a partial commutative monoid $(\Sigma, +, 0)$ where Σ is a nonempty set of elements (referred to as "worlds"), $+$ is a binary operator over worlds, and

0 is the unit for $+$. In Calcagno et al.'s definition, $+$ is a partial function, whereas Klein et al. defined it as a total function. For generality we shall assume that $+$ at least satisfies (in the sequel, if not stated otherwise, variables are implicitly universally quantified)
partial-functionality : if $x+y=z$ and $x+y=w$ then $z=w$.
Some formalizations of separation algebra also include a binary relation #, called "separateness" [11], over worlds. Two properties are given to the separateness relation: (1) $x \# 0$; and (2) $x \# y$ implies $y \# x$. The first one says every element is separated from the unit 0, the second one ensures the commutativity of #. As usual, the $+$ operator enjoys the following properties:
identity : $x+0=x$.
commutativity : $x \# y$ implies $x+y=y+x$.
associativity : if $x \# y$ and $y \# z$ and $x \# z$ then $(x+y)+z=x+(y+z)$.
Klein et al. then extend the above definitions with two more properties to obtain separation algebra: (1) if $x \# (y+z)$ and $y \# z$ then $x \# y$; (2) if $x \# (y+z)$ and $y \# z$ then $(x+y) \# z$. Finally, cancellative separation algebra extends the above with
cancellativity : if $x+z=y+z$ and $x \# z$ and $y \# z$ then $x=y$.

Assertions in separation algebra include the formulae of predicate calculus which are made from $\top, \bot, \neg, \rightarrow, \wedge, \vee$, and quantifiers $\exists, \forall$. In addition, there are multiplicative constant and connectives emp, $*$, and $-\!*$. In Isabelle/HOL, assertions can be encoded as predicates of type $'h \Rightarrow bool$ where $'h$ is the type of worlds in separation algebra. We write $\lfloor A \rfloor_w$ for the boolean formula resulting from applying the world w on the assertion A. The semantics of multiplicative assertions can be defined as:
$\lfloor emp \rfloor_w$ iff $w=0$.
$\lfloor P * Q \rfloor_w$ iff there exists x, y such that $x \# y$ and $w=x+y$ and $\lfloor P \rfloor_x$ and $\lfloor Q \rfloor_y$.
$\lfloor P -\!* Q \rfloor_w$ iff for all x, if $w \# x$ and $\lfloor P \rfloor_x$ then $\lfloor Q \rfloor_{(w+x)}$.

2.2 Further Extension of Separation Algebra

We extend Klein et al.'s library with the following properties that hold in many applications such as heap model and named permissions, as discussed in [8, 9, 16]:
indivisible unit : if $x+y=0$ and $x \# y$ then $x=0$.
disjointness : $x \# x$ implies $x=0$.
cross-split : if $a+b=w$, $c+d=w$, $a \# b$ and $c \# d$, then there exist e, f, g, h such that $e+f=a$, $g+h=b$, $e+g=c$, $f+h=d$, $e \# f$, $g \# h$, $e \# g$, and $f \# h$.
We call our extension *heap-sep-algebra* because our main application is the heap model. The following tactics also work for the algebra of Calcagno et al. if we remove from the tactics these extended properties.

2.3 The Labelled Sequent Calculus LS_{PASL}

The sequent calculus LS_{PASL} [21] for abstract separation logic is given in Fig. 1, where we omit the rules for classical connectives. A distinguishing feature of LS_{PASL} is that it has "structural rules" which manipulate ternary relational atoms. We define the ternary relation as: $(a, b \triangleright c) \equiv a \# b$ and $a+b=c$.

A sequent $\mathscr{G}; \Gamma \vdash \Delta$ contains a set $\mathscr{G}$ of ternary relational atoms, and sets Γ, Δ of labelled formulae of the form $h : A$, which corresponds to $\lfloor A \rfloor_h$ in the semantics.

$$\frac{\mathcal{G};h=0;\Gamma\vdash\Delta}{\mathcal{G};\Gamma;h:emp\vdash\Delta}\,\mathit{empl}\qquad\frac{}{\mathcal{G};\Gamma\vdash 0:emp;\Delta}\,\mathit{empr}$$

$$\frac{(h_1,h_2\triangleright h_0);\mathcal{G};\Gamma;h_1:A;h_2:B\vdash\Delta}{\mathcal{G};\Gamma;h_0:A*B\vdash\Delta}\,\mathit{starl}\qquad\frac{(h_1,h_0\triangleright h_2);\mathcal{G};\Gamma;h_1:A\vdash h_2:B;\Delta}{\mathcal{G};\Gamma\vdash h_0:A\mathrel{-\!\!*}B;\Delta}\,\mathit{magicr}$$

$$\frac{(h_1,h_2\triangleright h_0);\mathcal{G};\Gamma\vdash h_1:A;h_0:A*B;\Delta\qquad(h_1,h_2\triangleright h_0);\mathcal{G};\Gamma\vdash h_2:B;h_0:A*B;\Delta}{(h_1,h_2\triangleright h_0);\mathcal{G};\Gamma\vdash h_0:A*B;\Delta}\,\mathit{starr}$$

$$\frac{(h_1,h_0\triangleright h_2);\mathcal{G};\Gamma;h_0:A\mathrel{-\!\!*}B\vdash h_1:A;\Delta\qquad(h_1,h_0\triangleright h_2);\mathcal{G};\Gamma;h_0:A\mathrel{-\!\!*}B;h_2:B\vdash\Delta}{(h_1,h_0\triangleright h_2);\mathcal{G};\Gamma;h_0:A\mathrel{-\!\!*}B\vdash\Delta}\,\mathit{magicl}$$

$$\frac{(0,h_2\triangleright h_2);\mathcal{G};h_1=h_2;\Gamma\vdash\Delta}{(0,h_1\triangleright h_2);\mathcal{G};\Gamma\vdash\Delta}\,\mathit{eq}\qquad\frac{(h_2,h_1\triangleright h_0);(h_1,h_2\triangleright h_0);\mathcal{G};\Gamma\vdash\Delta}{(h_1,h_2\triangleright h_0);\mathcal{G};\Gamma\vdash\Delta}\,e$$

$$\frac{(0,h_2\triangleright 0);h_1=0;\mathcal{G};\Gamma\vdash\Delta}{(h_1,h_2\triangleright 0);\mathcal{G};\Gamma\vdash\Delta}\,\mathit{iu}\qquad\frac{(h_1,h_1\triangleright h_2);h_1=0;\mathcal{G};\Gamma\vdash\Delta}{(h_1,h_1\triangleright h_2);\mathcal{G};\Gamma\vdash\Delta}\,d\qquad\frac{(h,0\triangleright h);\mathcal{G};\Gamma\vdash\Delta}{\mathcal{G};\Gamma\vdash\Delta}\,u$$

$$\frac{(h_3,h_5\triangleright h_0);(h_2,h_4\triangleright h_5);(h_1,h_2\triangleright h_0);(h_3,h_4\triangleright h_1);\mathcal{G};\Gamma\vdash\Delta}{(h_1,h_2\triangleright h_0);(h_3,h_4\triangleright h_1);\mathcal{G};\Gamma\vdash\Delta}\,a$$

$$\frac{(h_1,h_2\triangleright h_0);h_0=h_3;\mathcal{G};\Gamma\vdash\Delta}{(h_1,h_2\triangleright h_0);(h_1,h_2\triangleright h_3);\mathcal{G};\Gamma\vdash\Delta}\,p\qquad\frac{(h_1,h_2\triangleright h_0);h_2=h_3;\mathcal{G};\Gamma\vdash\Delta}{(h_1,h_2\triangleright h_0);(h_1,h_3\triangleright h_0);\mathcal{G};\Gamma\vdash\Delta}\,c$$

$$\frac{(h_5,h_6\triangleright h_1);(h_7,h_8\triangleright h_2);(h_5,h_7\triangleright h_3);(h_6,h_8\triangleright h_4);(h_1,h_2\triangleright h_0);(h_3,h_4\triangleright h_0);\mathcal{G};\Gamma\vdash\Delta}{(h_1,h_2\triangleright h_0);(h_3,h_4\triangleright h_0);\mathcal{G};\Gamma\vdash\Delta}\,\mathit{cs}$$

Side conditions:
In *starl* and *magicr*, the labels h_1 and h_2 do not occur in the conclusion.
In *a*, the label h_5 does not occur in the conclusion.
In *cs*, the labels h_5,h_6,h_7,h_8 do not occur in the conclusion.

Fig. 1. The inference rules for multiplicative connectives and structural rules in LS_{PASL}.

Semicolon on the left hand side of $\vdash$ means classical conjunction and on the right means classical disjunction. The sequents on the top of a rule are premises, the one below is the conclusion. These inference rules are often used *backwards* in proof search. That is, to derive the conclusion, we need to derive the premises. The structural rules *eq*, *u* capture identity; *e*, *a* are for commutativity and associativity respectively; *d* for disjointness, which suffices to derive indivisible unit *iu*; *p*, *c* are for partial-functionality and cancellativity, and *cs* for cross-split.

3 Basic Proof Search

LS_{PASL} rules as lemmas. The first step towards developing automatic tactics in proof assistants based on the proof system LS_{PASL} is to translate each inference rule in LS_{PASL} to a lemma and prove that it is sound. Suppose a sequent takes the form

$$R_1;\ldots;R_l;s_1:A_1;\ldots;s_m:A_m\vdash s'_1:B_1;\ldots;s'_n:B_n$$

where R_i are ternary relational atoms over labels/worlds, A_i and B_j are separation logic assertions, s_i and s'_j are labels denoting worlds. We translate the sequent to a formula

Type		Rules
Invertible		*lspasl-empl*, *lspasl-empr*, *lspasl-starl*, *lspasl-magicr* *lspasl-eq*, *lspasl-p*, *lspasl-c*, *lspasl-iu*, *lspasl-d*
Quasi-invertible	Logical	*lspasl-starr*, *lspasl-magicl*
	Structural	*lspasl-u*, *lspasl-e*, *lspasl-a*, *lspasl-cs*

Table 1. The types of inferences rules in LS_{PASL}.

$$(R_1 \wedge \cdots \wedge R_l \wedge \lfloor A_1 \rfloor_{s_1} \wedge \cdots \wedge \lfloor A_m \rfloor_{s_m}) \rightarrow (\lfloor B_1 \rfloor_{s'_1} \vee \cdots \vee \lfloor B_n \rfloor_{s'_n}).$$

If a rule has premises P_1, P_2 and a conclusion C, we translate it to a lemma $(P_1 \wedge P_2) \rightarrow C$. If a rule has no premises, then we simply need to prove the conclusion. For instance, the rule *starr* in Fig. 1 is translated to the following lemma:

Lemma (lspasl-starr). $(((h_1, h_2 \triangleright h_0) \wedge \Gamma \rightarrow \lfloor A \rfloor_{h_1} \vee \lfloor A{*}B \rfloor_{h_0} \vee \Delta) \wedge$
$((h_1, h_2 \triangleright h_0) \wedge \Gamma \rightarrow \lfloor B \rfloor_{h_2} \vee \lfloor A{*}B \rfloor_{h_0} \vee \Delta)) \rightarrow ((h_1, h_2 \triangleright h_0) \wedge \Gamma \rightarrow \lfloor A{*}B \rfloor_{h_0} \vee \Delta)$

Note that we combine $\mathcal{G}$ and Γ in the rule into Γ in the lemma because in proof assistants Γ is an arbitrary formula which can be used to represent both. The above lemma is thus stronger than the soundness of a direct translation of sequents.

For each inference rule r in Fig. 1, we prove a corresponding lemma *lspasl-r* to show the soundness of the rule in Calcagno et al.'s separation algebra. In the sequel we may loosely refer to an inference rule as its corresponding lemma. We have also proved the inverted versions of the those lemmas which show that all the rules in LS_{PASL} are *invertible*. That is, if the conclusion is derivable, so are the premises. Completeness for Klein et a.'s formalisation is beyond this work because the semantics that LS_{PASL} is complete for, which is also widely used in the literature [9, 16], does not consider the "separateness" relation, thus LS_{PASL} itself lacks the treatment of this relation.

Theorem (Soundness). *LS_{PASL} is sound with respect to* heap-sep-algebra*.*

Lemma (Invertibility). *The inference rules in LS_{PASL} are invertible.*

Proof search using LS_{PASL}. Proof assistants such as Isabelle/HOL can automatically deal with first-order connectives such as $\top$, $\bot$, $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\exists$ and $\forall$, so we do not have to integrate the rule applications for these connectives in proof search. We divide the other inference rules in two groups: those that are truly invertible, and those that are only invertible because we "copy" the conclusion to the premises. The intuition is as follows: "invertible" rules are those that can be applied whenever possible without increasing the search space unnecessarily. The types of inference rules are summarised in Table 1.

We analyse each rule *lspasl-r* in Table 1 and prove a lemma *lspasl-r-der* for a form of backward derivation. Such lemmas will be directly used in the tactics. Quasi-invertible rules such as lspasl-starr and lspasl-magicl need to be used with care because they may generate useless information and add unnecessary subgoals. Continuing with the example of the rule lspasl-starr, reading it backwards yields the following lemma:

Lemma (lspasl-starr-der). *If* $(h_1, h_2 \triangleright h_0)$ *and* $\neg \lfloor A{*}B \rfloor_{h_0}$*, then*
$((h_1, h_2 \triangleright h_0) \wedge \neg(\lfloor A \rfloor_{h_1} \vee \lfloor A{*}B \rfloor_{h_0}) \wedge (\mathit{starr\_applied}\ h_1\ h_2\ h_0\ (A{*}B)))$ *or*
$((h_1, h_2 \triangleright h_0) \wedge \neg(\lfloor B \rfloor_{h_2} \vee \lfloor A{*}B \rfloor_{h_0}) \wedge (\mathit{starr\_applied}\ h_1\ h_2\ h_0\ (A{*}B)))$

We include the assumptions in each disjunct so that contraction is admissible. We also include a dummy predicate "starr_applied" on each disjunct to record this rule application. This predicate is defined as $\mathit{starr\_applied}\ h_1\ h_2\ h_0\ F\ \equiv (h_1, h_2 \triangleright h_0) \wedge \neg \lfloor F \rfloor_{h_0}$.

We use three tactics to reduce search space when lspasl-starr or lspasl-magicl is applied. The first tactic is commonly used in provers for BBI and abstract separation logics [21, 33]. For example, we forbid applications of lspasl-starr on the same pair of labelled formula and ternary relational atom more than once, because repeating applications on the same pair will not advance the proof search. To realise this, we generate the predicate "starr_applied" in proof search *only* when the corresponding pair is used in a rule application. We can then check if this predicate is generated during proof search, and avoid applying the rule on the same pair again.

The second tactic applies Lemma lspasl-starr-der2, which is an alternative of the above lemma that applies lspasl-starr on $\neg \lfloor A{*}B \rfloor_{h_0}$ and $(h_2, h_1 \triangleright h_0)$:

Lemma (lspasl-starr-der2). *If* $(h_1, h_2 \triangleright h_0)$ *and* $\neg \lfloor A{*}B \rfloor_{h_0}$, *then*
$((h_1, h_2 \triangleright h_0) \wedge \neg(\lfloor A \rfloor_{h_2} \vee \lfloor A{*}B \rfloor_{h_0}) \wedge (\mathit{starr\_applied}\ h_2\ h_1\ h_0\ (A{*}B)))$ *or*
$((h_1, h_2 \triangleright h_0) \wedge \neg(\lfloor B \rfloor_{h_1} \vee \lfloor A{*}B \rfloor_{h_0}) \wedge (\mathit{starr\_applied}\ h_2\ h_1\ h_0\ (A{*}B)))$

This is a crucial step because without it we will have to wait for the lspasl-e rule application to generate the commutative variant $(h2, h1 \triangleright h)$, and this particular rule application may be very late in proof search.

The third tactic is a look-ahead in the search: analyse each pair of $\neg \lfloor A{*}B \rfloor_{h_0}$ and $(h_1, h_2 \triangleright h_0)$ in the subgoal, and look for $\lfloor A \rfloor_{h_1}$ and $\lfloor B \rfloor_{h_2}$ in assumptions. If we can find at least one of them, then we can safely apply Lemma lspasl-starr-der and solve one subgoal immediately; thus the proof search space is not increased too much. We refer to the look-ahead tactics as *lspasl-starr-der-guided* (resp. *lspasl-magicl-der-guided*). Similar tactics are also developed for the rule lspasl-magicl. We apply lspasl-starr-der-guided and lspasl-magicl-der-guided whenever possible.

The structural rule lspasl-u requires more care, because it generates a new ternary relational atom out of nothing. A natural restriction is to forbid generating an atom if it already exists in the subgoal. Moreover, we only generate $(h, 0 \triangleright h)$ when (1) h occurs in some ternary relational atom (in the subgoal), or (2) h occurs in some labelled formula. We call these two applications *lspasl-u-der-tern* and *lspasl-u-der-form* respectively.

We develop two proof methods for the associativity rule lspasl-a. In the first method, *lspasl-a-der*, when we find the two assumptions $(h_1, h_2 \triangleright h_0)$ and $(h_3, h_4 \triangleright h_1)$, we only apply the rule lspsal-a when none of the following appear in the subgoal:

$(0, h_2 \triangleright h_0)$, $(h_1, 0 \triangleright h_0)$, $(h_1, h_2 \triangleright 0)$, $(0, h_4 \triangleright h_1)$, $(h_3, 0 \triangleright h_1)$,
$(\_, h_3 \triangleright h_0)$, $(h_3, \_ \triangleright h_0)$, $(h_2, h_4 \triangleright \_)$, $(h_4, h_2 \triangleright \_)$.

In the first 5 cases, we can simplify the subgoal by unifying labels. For instance, the first case implies that $h_2 = h_0$, which can be derived by the rule lspasl-eq. The last 4 cases (_ means any label/world) indicate that one of the atoms to be generated *may* already exist in the subgoal, so we delay this rule application to the second method, *lspasl-a-der-full*, in which we generate all possible associative variants of the assumptions.

Real-world applications often involve reasoning of the form "if this assertion holds for all heaps, then ..." [30]. Hóu and Tiu's recent work included treatments for separation logic modalities with similar semantics [24]. For example, if the quantifier occurs on the left hand side of the sequent, they instantiate the quantified world to either an existing world or a fresh variable. This kind of reasoning often can be handled by existing lemmas in proof assistants, such as *meta-spec* in Isabelle/HOL. Therefore we do not detail the treatment for such quantifiers. We call the tactic for universal quantifiers on worlds *lsfasl-boxl-der* since it mimics the $\Box L$ rule in [24].

We are now ready to present the basic proof search. The first step is to "normalise" the subgoal from $P_1 \Longrightarrow P_2 \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow C$ to $P_1 \Longrightarrow P_2 \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow \neg C \Longrightarrow \bot$; otherwise, if C is some $A * B$, Lemma lspasl-starr-der will fail to apply on the subgoal. This preparation stage is called "prep".

Then we apply the "invertible" rules as much as possible, this is realised by a loop of applying the following lemmas until none are applicable: lspasl-empl-der, lspasl-starl-der, lspasl-magicr-der, lspasl-iu-der, lspasl-d-der, lspasl-eq-der, lspasl-p-der, lspasl-c-der, lspasl-starr-der-guided, lspasl-magicl-der-guided. This stage is called "invert".

The application of Lemma lsfasl-boxl-der follows, then come "quasi-invertible" rules. When applying the lemmas for structural rules lspasl-u-der-tern (identity), lspasl-e-der (commutativity), and lspasl-a-der (associativity), we apply them as much as possible based on existing ternary relational atoms in the (first) subgoal. We call this loop "non-inv-struct". We do not apply quasi-invertible logical rules as much as possible because that will produce too many subgoals. Thus in the "non-inv-logical" stage we apply one of lspasl-starr-der, lspasl-starr-der2, lspasl-magicl-der, lspasl-magicl-der2 only once.

Finally, we apply one of three rarely used lemmas in the end: lspasl-u-der-form, lspasl-cs-der, lspasl-a-der-full. We call this stage "rare".

The basic proof search procedure, named *separata*, is an infinite loop of the above stages until the subgoal is proven or none of the lemmas are applicable. We can express separata by the following regular expression, where "|" means "or" and "+" means one or more applications of the preceding element:

separata $\equiv$ (prep | (invert | lsfasl-boxl-der | non-inv-struct | non-inv-logical)+ | rare)+

4 Advanced Tactics for Proof Search

Although separata can handle all logical connectives, it is inefficient when the formula contains a complex structure with $*$ and $-\!*$. This section extends separata with specialised tactics for $*$ and $-\!*$, which pose the main difficulties in reasoning with separation logic. The former connective is pervasive in program verification, and the latter connective is important when reasoning about concurrent programs with rely/guarantee techniques [38]. We also integrate proof methods for pointers in the heap model.

4.1 Formula-driven tactics for the $*$ Connective

One of the hardest problems in reasoning about resources is to find the correct partition of resources when applying the (linear) conjunction right rule. In certain fragments of separation logic such as symbolic heap, this problem is simplified to AC-rewriting and can be solved relatively easily with existing techniques. However, in a logic with richer syntax, theorem provers often struggle to find the right partition of resources; this can be observed from the experiments of theorem provers for BBI and abstract separation logics [21, 23, 33]. To capture arbitrary interaction between additive connectives ($\wedge$, $\rightarrow$) and multiplicative connectives ($*$, $-\!*$), LS_{PASL} uses ternary relational atoms as the underlying data structure, which complicates the reasoning. This subsection proposes two-stage tactics for such situations, and gives two solutions for the second stage. Our techniques can also be adopted in other logic systems with ternary relations. Consider the following example:

Example 1. $(h_1,h_2 \triangleright h_3) \Longrightarrow (h_4,h_5 \triangleright h_1) \Longrightarrow (h_6,h_7 \triangleright h_2) \Longrightarrow (h_8,h_9 \triangleright h_6) \Longrightarrow (h_{10},h_{11} \triangleright h_8) \Longrightarrow \lfloor A \rfloor_{h_4} \Longrightarrow \lfloor B \rfloor_{h_5} \Longrightarrow \lfloor C \rfloor_{h_{10}} \Longrightarrow \lfloor D \rfloor_{h_{11}} \Longrightarrow \lfloor E \rfloor_{h_9} \Longrightarrow \lfloor F \rfloor_{h_7} \Longrightarrow \cdots \Longrightarrow \neg\lfloor (((B*E)*(A*D))*C)*F \rfloor_{h_3} \Longrightarrow \bot$

Recall Lemma lspasl-starr-der in Section 3. To apply it, we need to find an atom $(h_1,h_2 \triangleright h_0)$ which matches the labelled formula $\neg\lfloor A*B \rfloor_{h_0}$. The ternary relation represents a partition of the resource h_0, and only the correct partition will lead to a derivation. In separata, "non-inv-struct" applies identity, commutativity, and associativity without any direction. It may generate many atoms that are not needed and increase the search space. Thus the first problem is how to generate the exact set of ternary relational atoms for lspasl-starr applications. Let us take a closer look at the subgoal by viewing each ternary relational atom $(h,h' \triangleright h'')$ as a binary tree where h'' is the root and h,h' are leaf nodes. We then obtain the binary tree in Fig. 2 (left).

The first stage of the tactics is to analyse the structure of the $*$ formula and try to locate each piece of resource in the subgoal. In Example 1, it is easy to observe that A is true at world h_4 etc. Combined with the structure of the formula, we obtain the binary tree in Fig. 2 (right), which contains a few question marks that represent the worlds which are currently unknown. For instance, we do not know what is the combination of h_5 and h_9 in the subgoal; thus we should create a new ternary relational atom $\exists h.(h_5,h_9 \triangleright h)$ with a fresh symbol h, and try to find an instance of h later.

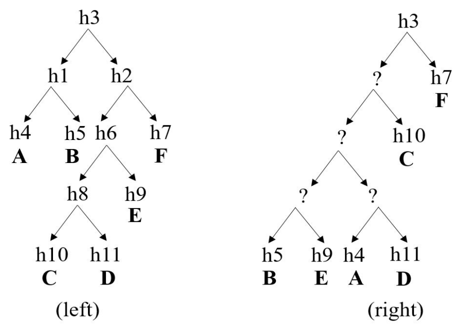


Fig. 2. Graphical representation of Example 1.

In a more general case, we first give an algorithm *findworld* (Algorithm 1) to find the world where a formula is true at and store all the new ternary relational atoms we create. We use "@" for concatenation of lists and "[]" for an empty list. The next step is to collect all the ternary relational atoms we have created to obtain Fig. 2 (right), as done in the algorithm *starstruct* (Algorithm 2).

Now that we know exactly the set of required ternary relational atoms, the second stage of the tactics is to derive Fig. 2 (right) from (left). We propose two solutions to the second stage. The first solution works for the separation algebra defined in Dockins et al.'s work [16], which is more general than the one used in this paper. A common property shared by Fig. 2 (left) and (right) is that the two binary trees have the same root and the same multiset of leaf nodes. It is easy to observe that a binary tree naturally

```
Data: subgoal, and a formula $F$
Result: a pair of a world and a list of ternary relational atoms
if $\lfloor F \rfloor_h$ is in subgoal for some $h$ then
    return $(h, [])$;
else if $F \equiv A \wedge B$ or $F \equiv A \vee B$ or $F \equiv A \to B$ then
    $(h, l) \leftarrow$ findworld(subgoal, $A$);
    if $h \equiv NULL$ then
        return findworld(subgoal, $B$);
    else
        return $(h, l)$;
else if $F \equiv \neg A$ or $F \equiv \exists x.A$ or $F = \forall x.A$ then
    return findworld(subgoal, $A$);
else if $F \equiv emp$ then
    return $(0, [])$;
else if $F \equiv A * B$ then
    $(ha, la) \leftarrow$ findworld(subgoal, $A$); $(hb, lb) \leftarrow$ findworld(subgoal, $B$);
    if $(ha, hb \triangleright h)$ occurs in subgoal for some $h$ then
        return $(h, la@lb)$;
    else
        Create a fresh variable $h'$; return $(h', la@lb@[(ha, hb \triangleright h')])$;
else
    return $(NULL, [])$;
```

Algorithm 1: The algorithm findworld.

```
Data: subgoal, and negated star formula $\neg\lfloor A*B \rfloor_h$
Result: a conjunction of ternary relational atoms
$(ha, la) \leftarrow$ findworld(subgoal, $A$); $(hb, lb) \leftarrow$ findworld(subgoal, $B$);
Make a conjunction of each ternary relational atom in $la@lb@[(ha, hb \triangleright h)]$ and
  existentially quantify over all the variables created in findworld;
```

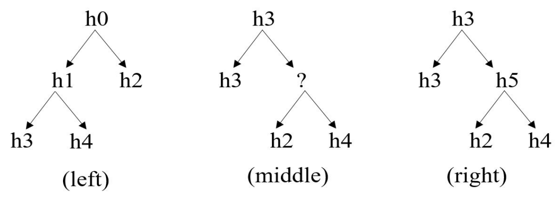
Algorithm 2: The algorithm starstruct.

corresponds to a list of ternary relational atoms. We can use the following lemma to derive Fig. 2 (right) where we say a node is *internal* if it is not the root nor a leaf node:

Lemma. *Given two binary trees t_1 and t_2 with the same root and the same multiset of leaf nodes. Suppose every internal node in t_2 is existentially quantified. There exists a sequence of lspasl-e-der and lspasl-a-der applications to derive t_2 from t_1.*

The intuition is that Fig. 2 (left) and (right) can be seen as parse trees of $*$ connected terms with different ways of bracketing. The two lemmas lspasl-e-der and lspasl-a-der correspond respectively to applications of commutativity and associativity of $*$. Those applications grant us a transformation from one bracketing to the other bracketing.

In the case that certain internal nodes in t_2 are not existentially quantified, by the construction in Algorithm 1 and 2, they must be existing worlds in the subgoal. Suppose the subgoal contains a binary tree in below (left) and also contains $(h_2, h_4 \triangleright h_5)$, and Algorithm 1 and 2 suggest to derive below (right). We can still use the above lemma to derive below (middle) from (left), where the question mark is an existentially quantified variable. We then instantiate the quantified variable to a fresh variable, e.g., h_6, and use lspasl-p-der (partial-functionality of $+$ and $\triangleright$) to unify h_6 and h_5, then derive below (right). This solution may be easy to implement in an external theorem prover, but we



faced difficulties when implementing it in Isabelle/HOL. Specifically, whenever we use Algorithm 1 and 2 to obtain the atoms we need to derive, we have to prove that those atoms correspond to a binary tree. Since ternary relation is a definition in Isabelle/HOL, the proof of the tree representation is non-trivial and slow for large instances.

The second solution is inspired by "forward reasoning" and "inverse method" [13]. This solution does not depend on the tree representation, but we shall describe it in terms of trees for simplicity. Instead of deriving Fig. 2 (right) from (left), we build (right) up from scratch using the information in (left). This can be seen as forward reasoning on ternary relational atoms. We start by choosing the bottom-most "sub-tree" $(h, h' \triangleright h'')$ in the tree to be derived. If we can prove that $h \# h'$, then there must be a world that represents the combination of h and h'. If h'' is not existentially quantified, we can use partial-functionality to derive that the combination of h and h' must be h''. Proving $h \# h'$ is the hard part. The intuition is that if h and h' are two leaf nodes in a (fragment of the) tree formed from the subgoal, then they must be "separated". For instance, from Fig. 2 (left), we should be able to derive $h_4 \# h_{10}$ since they are both leaf nodes. We should also be able to derive $h_4 \# h_6$ because they are leaf nodes of a fragment of the tree. We need to prove the following lemmas to reason about "separateness" of worlds:

disj-distri-tern : if $w \# z$ and $(x, y \triangleright z)$ then $w \# x$.
disj-distri-tern-rev : if $x \# y$ and $x \# z$ and $(y, z \triangleright w)$ then $x \# w$.
disj-comb : if $(x, y \triangleright z)$ and $x \# w$ and $y \# w$ then $z \# w$.
exist-comb : $x \# y$ implies $\exists z.(x, y \triangleright z)$.

Now we construct Fig. 2 (right) bottom-up using the algorithm *provetree* (Algorithm 3). The first step is to derive $h_5 \# h_9$ using lemmas disj-comb, disj-distri-tern, disj-distri-tern-rev, and lspasl-e-der, and unfolding the definition of the ternary relation. Next we obtain $\exists h.(h_5, h_9 \triangleright h)$ using Lemma exist-comb. The part in Algorithm 3 where we show $h''' = h''$ can be done by applying lemmas lspasl-a-der and lspasl-e-der and unfolding the definition of the ternary relation. We repeat this process until we have derived the entire tree in Fig. 2 (right). Now we can use it in lspasl-starr-der applications to solve Example 1. The whole picture is that we guess the shape of the binary tree, and guide the application of structural rules by the structure of the $*$ formula. Thus we achieve a "formula-driven" proof search. We call the above tactics "starr-smart" and we extend separata by applying starr-smart between "invert" and "lsfasl-boxl-der".

```
Data: subgoal, and a tree $t$ representing conjunctions of ternary relational atoms
Result: A proof that subgoal $\Longrightarrow t$
repeat
    Choose a lowest level ternary relational atom $(h, h' \triangleright h'')$ in $t$;
    Prove $h \# h'$ using lemmas disj-comb, disj-distri-tern and disj-distri-tern-rev;
    if $h''$ is existentially quantified then
        Derive $\exists h''.(h, h' \triangleright h'')$ using Lemma exist-comb;
    else
        Derive $\exists h'''.(h, h' \triangleright h''')$ using Lemma exist-comb; Prove that $h''' = h''$;
until All ternary relational atoms in $t$ are covered;
```

Algorithm 3: The algorithm provetree.

4.2 Tactics for Magic Wand

Although $-\!\!*$ in general is very difficult to handle and it is often deemed as a source of non-recursive-enumerability in the heap model [6], we observe that many applications of $-\!\!*$ in [30] are of the form $(A * (B -\!\!* C)) \to (A' * C)$ where A is a $*$ connected formula which can be transformed to $A' * B$. Consider the following example:

$$(D * A * ((D * C) -\!\!* B) * C) \to (A * B)$$

Instead of deriving the correct way to split the resource according to the formula on the right hand side, we can use associativity and commutativity of $*$ and transform the left hand side into $A * (D * C) * ((D * C) -\!\!* B)$, which suffices to deduce the right hand side by the following lemma:

Lemma (magic-mp). $\lfloor C * (C -\!\!* B) \rfloor_h$ *implies* $\lfloor B \rfloor_h$.

Hence the key in this tactic is to transform a formula into the form $A * C * (C -\!\!* B)$ then simplify the formula. There are many ways to implement this, for simplicity we can analyse each $*$ connected formula $F \equiv F_1 * \cdots * F_n$, and for each $F_i \equiv (C -\!\!* B)$ where $C \equiv C_1 * \cdots * C_m$, we try to match F_j, $j \neq i$ with C_k. If each C_k can be successfully matched with a (different) F_j, we can then obtain a remainder R such that $F \equiv R * C * (C -\!\!* B)$. We then apply Lemma magic-mp to remove this occurrence of $-\!\!*$. We integrate this tactic into the sub-procedure "invert". The extension of separata with the tactics in Section 4.1 and 4.2 is called *starforce*.

Compared to separata, starforce applies structural rules and the rules for $*$ and $-\!\!*$ using specialised strategies, which often lead to better performance. However, in rare cases, starforce may be too aggressive and its intermediate tactics may get stuck. Therefore we leave both options to the user.

4.3 Tactics for the Heap Model

The separation algebra in this paper can be easily extended to capture pointers and potentially other data structures in the heap model. We shall focus on pointers here. For generality, we can define a points-to predicate as $'a \Rightarrow' v \Rightarrow' h \Rightarrow bool$ where $'a$ and $'v$ can be instantiated to *address* and *value* respectively in a concrete model, and $'h$ is the type of worlds in the separation algebra. We shall write this predicate as $\lfloor (a \mapsto v) \rfloor_h$, the intended meaning is that address a in heap h has value v. We can then give this relation some properties à la Hóu et al.'s work [24] to mimic pointers in the heap model:

Injection : if $\lfloor (a \mapsto v) \rfloor_{h_1}$ and $\lfloor (a \mapsto v) \rfloor_{h_2}$ then $h_1 = h_2$.
Non-emptiness : $\neg \lfloor (a \mapsto v) \rfloor_0$.
Not-larger-than-one : if $\lfloor (a \mapsto v) \rfloor_h$ and $(h_1, h_2 \triangleright h)$ then $h_1 = 0$ or $h_2 = 0$.
Address-disjointness : $\neg \lfloor (a_1 \mapsto v_1) \rfloor_{h_1}$ or $\neg \lfloor (a_1 \mapsto v_2) \rfloor_{h_2}$ or $\neg (h_1, h_2 \triangleright h)$.
Uniqueness : if $\lfloor (a_1 \mapsto v_1) \rfloor_h$ and $\lfloor (a_2 \mapsto v_2) \rfloor_h$ then $a_1 = a_2$ and $v_1 = v_2$.
Extensibility : for any h, v, there exist a, h_1, h_2 such that $(h_1, h \triangleright h_2)$ and $\lfloor (a \mapsto v) \rfloor_{h_1}$.

It is straightforward to prove corresponding lemmas for these properties and integrate the application of the lemmas into starforce, resulting in a new method *sepointer*.

5 Examples

This section demonstrates our implementation of the above tactics in Isabelle/HOL. Our proof methods separata, starforce, and sepointer can prove formulae *automatically* without human interaction. For space reasons we only show some examples. The source code and an extensive list of tested formulae can be accessed at [2].

Benchmark examples. We show three BBI formulae from the previous benchmarks [21, 33], these formulae are also valid in separation logic. The first one is very hard for existing BBI theorem provers, but it can be solved easily *by separata*, which combines the strength of the Isabelle engine and LS_{PASL} proof system.

$$(emp \to (p0 -\!\!* (((p0 * (p0 -\!\!* p1)) * (\neg p1)) -\!\!* (p0 * (p0 * ((p0 -\!\!* p1) * (\neg p1))))))) \to$$
$$((((emp * p0) * (p0 * ((p0 -\!\!* p1) * (\neg p1)))) \to (((p0 * p0) * (p0 -\!\!* p1)) * (\neg p1))) * emp)$$

Without separata, one could rely on Isabelle's *sledgehammer*, which will spend a few seconds to find a proof. There are also examples that sledgehammer fails to find proofs in 300s, but separata can solve them instantly:

$$\neg (((A * (C -\!\!* (\neg ((\neg (A -\!\!* B)) * C)))) \wedge (\neg B)) * C)$$

In general, our Isabelle tactics can prove many complicated formulae which otherwise may be time consuming to find proofs in Isabelle.

We have tested our tactics on other benchmark formulae for BBI and PASL provers [21, 33], both separata and starforce can prove those formulae automatically.

Example 1 in Section 4 is an instance that separata struggles but starforce can solve it easily. Similarly, starforce can easily prove the following formula, which is inspired by an example in seL4, using the tactic in Section 4.2:

$$(E * F * G * ((C * Q * R) -\!\!* B) * C * ((G * H) -\!\!* I) * H * ((F * E) -\!\!* Q) * A * R) \to (A * B * I)$$

The tactics for the heap model allow us to demonstrate more concrete examples. For instance, the following formula from the benchmark in [24] says that if the current heap can be split into two parts, one is $(e_1 \mapsto e_2)$ and the other part does not contain $(e_3 \mapsto e_4)$, and the current heap contains $(e_3 \mapsto e_4)$, then we deduce that $(e_3 \mapsto e_4)$ and $(e_1 \mapsto e_2)$ must be the same heap, therefore $e_1 = e_3$. This kind of reasoning about overlaid data structures requires applications of cross-split, and it is usually non-trivial to find proofs manually (Sledgehammer fails to find a proof in 300s).

$$(((e_1 \mapsto e_2) * \neg ((e_3 \mapsto e_4) * \top)) \wedge ((e_3 \mapsto e_4) * \top)) \to (e_1 = e_3)$$

We can also prove some properties about "septraction" in separation logic with rely/guarantee where $A -\!\!\circledast B \equiv \neg (A -\!\!* \neg B)$, such as the formula below [38]:

$$((x \mapsto y) -\!\!\circledast (z \mapsto w)) \to ((x = z) \wedge (y = w) \wedge emp)$$

Examples in seL4 proofs. Klein et al. implemented separation algebra in Isabelle/HOL as a part of the renowned seL4 project. Many lemmas in their proofs related to separation algebra can now be proved with a single application of separata or starforce. The method *sep-solve* developed in the seL4 project fails to prove most of the examples we have tested for our tactics. Compared to the tactics developed in this paper, sep-solve is more ad-hoc. That is, our tactics are based on a more systematic proof theory (the labelled sequent calculus), whereas sep-solve focuses on special cases that are useful in practice. As a result, although sep-solve also have similar tactics to handle $-*$, its treatment is different because it does not consider ternary relational atoms as its underlying data structure. Below is a lemma in the development of sep-solve:

lemma $(\bigwedge s. \lfloor Q \rfloor_s \Longrightarrow \lfloor Q' \rfloor_s) \Longrightarrow \lfloor R -\!* R' \rfloor_s \Longrightarrow \lfloor (Q * R) -\!* (Q' * R') \rfloor_s$

(1) **apply** *(erule sep-curry[rotated])* *(2)* **apply** *(sep-select-asm 1 3)*

(3) **apply** *(sep-drule (direct) sep-mp-frame)* *(4)* **by** *(sep-erule-full refl-imp, clarsimp)*

This proof can now be simplified to just "**by** separata". In cases like above, separata/starforce can be used as substitutes in the correctness proof of seL4.

6 Case Study

There are two ways to use our tactics: the user can prove that a logic is an instantiation of separation algebra, and directly apply our proof methods, as demonstrated in the semantics of actions; or the user can implement our tactics in another framework, as shown in our extension of the SL framework of Lammich and Meis [28].

Semantics of actions. Our ongoing project involves integrating the semantics of actions in Feng's local rely/guarantee [17] in the CSimpl framework [35]. The assertion language of local rely/guarantee extends separation logic assertions with an additional semantic level to specify predicates over pairs (σ, σ') of states, called *actions*, which are represented by the states before and after the action. The semantics of actions redefines the separation logic operations in terms of Cartesian product of states. Additionally, the assertion language at the state level defines the separation logic operators for a state composed of three elements (l, s, i) to represent local, shared, and logical variables respectively. We represent both actions and states as products, and the core of the local rely guarantee's assertion language can be defined in CSimpl by showing that the Cartesian product of two heap-sep-algebras is a heap-sep-algebra. The instantiation of the product as a heap-sep-algebra involves proving the following properties:

zero-prod-def : $0 \equiv (0,0)$.

plus-prod-def : $p_1 + p_2 \equiv ((fst\ p_1) + (fst\ p_2), (snd\ p_1) + (snd\ p_2))$.

sep-disj-prod-def : $p_1 \# p_2 \equiv ((fst\ p_1) \# (fst\ p_2)) \wedge ((snd\ p_1) \# (snd\ p_2))$.

We then prove that the properties in Sections 2.1 and 2.2 hold for pairs of actions. For an application, we use our tactics to prove the following lemma where

$\lceil a \rceil \equiv (\lambda(\sigma,\sigma').\ (\sigma{=}\sigma') \wedge (a\ \sigma))$ and $tran\text{-}Id \equiv \lceil \lambda s.\ True \rceil$:

lemma assumes *a1:* $\lfloor A * tran\text{-}Id \rfloor_{(\sigma 1, \sigma 2)}$ **and** *a2:* $(\sigma 1, \sigma' \triangleright \sigma 1') \wedge (\sigma 2, \sigma' \triangleright \sigma 2')$

shows $\lfloor A * tran\text{-}Id \rfloor_{(\sigma 1', \sigma 2')}$

proof - **from** *a2* **have** $((\sigma 1, \sigma 2), (\sigma', \sigma') \triangleright (\sigma 1', \sigma 2'))$ **by** $(metis(full\text{-}types)\ tern\text{-}dist1)$

then show $?thesis$ **using** $a1$ $id\text{-}pair\text{-}comb$ **apply** $(simp\ add:\ tran\text{-}Id\text{-}def\ satis\text{-}def)$

by *separata* **qed**

Here $\lceil a \rceil$ represents the action with equal initial and end states that satisfies a, so $tran\_id$ represents the identity relation. Before we use separata in semantics of actions we have to provide some domain knowledge that separata does not know, such as the first step which composes two ternary relational atoms into a ternary relational atom of pairs. We then need to unfold the definitions in the semantics of actions, and separata can solve the resultant subgoal quickly.

Lammich and Meis's SL framework. In case the proof of instantiation of separation algebra is complex or Isabelle/HOL is not accessible, the user can implement our tactics in another framework (or even proof assistant). To demonstrate this we port separata to Lammich and Meis's SL framework [28] (source code at [2]). This process involves proving that the inference rules in LS_{PASL} are sound and adopting the applications of the rules in the new framework. Developing advanced tactics is feasible but time-consuming.

7 Related Work

Separation algebra was first defined as a cancellative, partial commutative monoid [11], and later formalized by Klein et al. in Isabelle/HOL [25]. Their definition includes a "separateness" relation # which is not used in other works such as [8, 9, 16]. We did, however, find this relation essential in developing tactics for automated reasoning (cf. Section 4). Later developments by Brotherston et al. [8] and Dockins et al. [16] added a few more properties in separation algebra, such as single unit, indivisible unit, disjointness, splittability, cross-split. We extend Klein et al.'s formalisation with all these properties except splittability because it does not hold in our applications. The proof theory for logics of separation algebra dates back to the Hilbert system and sequent calculus for Boolean bunched implications (BBI) [32]. The semantics of BBI is a generalised separation algebra: a non-deterministic commutative monoid [18]. The undecidability of BBI and other separation algebra induced logics [8, 29] did not stop the development of semi-decision procedures, including display calculus [7], nested sequent calculus [33], and labelled sequent calculus [23]. Among these proof systems, nested sequent calculus and labelled sequent calculus are more suitable for automated reasoning. Hóu et al. developed labelled sequent calculi for propositional abstract separation logics [21] and corresponding theorem provers. Brotherson and Villard gave an axiomatisation for separation algebras using hybrid logic [9]. As far as we know, except Klein et. al.'s work [25], none of these proof systems nor their proof search procedures have been formalised in a proof assistant.

Historically, the term "separation logic" refers to both the framework for reasoning about programs and the assertion logic in the framework. There have been numerous mechanisations of separation logic frameworks, but most of them focus on the reasoning of programs (e.g., [36]), whereas this paper focuses on the reasoning of assertions, so they are not directly comparable to this work. Moreover, most mechanisations of separation logic framework, e.g., Smallfoot [4], Holfoot [37], Myreen's rewriting tactics for SL [31], Ynot [15], Bedrock [14], and Charge! [3], only use a small subset of the assertion language, typically variants of symbolic heaps. Although some of those assertion logics are also induced from separation algebra, having a simpler syntax means that the reasoning task may be easier, and more efficient tactics, such as bi-abduction [10], can be developed for those logics. Consequently, the reasoning in those assertion logics is also not comparable to our work, which considers the full first-order assertion language. The few examples that use the full (or even higher-order) assertion language include Lammich and Meis's Isabelle/HOL formalisation of SL [28], Varming and Birkedal's formalisation of Higher-order SL (HOSL) [39], and the Iris project [26]. Lammich and Meis's SL framework includes a proof method *solve_entails* for assertions. A close inspection on the source code shows that it is mostly used to prove rather simple formulae such as $(A * emp) \to (A * \top)$ and $A * B \to B * A$ (although it can reason about some properties of lists). These formulae can be easily proved by our tactics. On the other hand, none of the examples shown in Section 5 can be proved by solve_entails. The interactive proof mode in Iris 3.0 [27] can solve many formulae in a restricted format, which is sufficient in their application. However, their tactics are not fully automatic. The formalisation of HOSL [39] also lacks automated proof methods.

This paper fills the gap of automated tactics for assertions in formalisations of SL. It is straightforward to adopt our tactics in other Isabelle/HOL formalisations; implementation in Coq should also be feasible since one can translate our tactics to Gallina and OCaml embedded code in Coq. Thus our work can be used to greatly improve the automation in SL mechanisations that involve the full language of assertions.

References

1. Facebook Infer. http://fbinfer.com/.
2. Isabelle/HOL tactics for separation algebra. http://securify.scse.ntu.edu.sg/SoftVer/Separata.
3. Jesper Bengtson, Jonas B. Jensen, and Lars Birkedal. Charge! - a framework for higher-order separation logic in coq. In *ITP'12*, pages 315–331, 2012.
4. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO'05*, pages 115–137. Springer, 2005.
5. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS'05*, volume 3780, pages 52–68, 2005.
6. Rémi Brochenin, Stphane Demri, and Etienne Lozes. On the almighty wand. *JIC*, 2012.
7. James Brotherston. A unified display proof theory for bunched logic. *ENTCS*, 2010.
8. James Brotherston and Max Kanovich. Undecidability of propositional separation logic and its neighbours. *Journal of ACM*, 2014.
9. James Brotherston and Jules Villard. Parametric completeness for separation theories. In *POPL '14*, pages 453–464, 2014.
10. Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of ACM*, 58(6):26:1–26:66, 2011.
11. Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS'07*, pages 366–378. IEEE, 2007.
12. Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theor. Comput. Sci.*, 232(1-2):133–163, 2000.
13. Kaustuv Chaudhuri and Frank Pfenning. Focusing the inverse method for linear logic. In *CSL '05*, pages 200–215, 2005.
14. Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI'11*, pages 234–245, 2011.

15. Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP'09*, 2009.
16. Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *APLAS'09*, volume 5904, pages 161–177, 2009.
17. Xinyu Feng. Local rely-guarantee reasoning. In *POPL '09*, pages 315–327. ACM, 2009.
18. D. Galmiche and D. Larchey-Wendling. Expressivity properties of Boolean BI through relational models. In *FSTTCS'06*, 2006.
19. Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *OSDI'16*, OSDI'16, pages 653–669, 2016.
20. Joshua S. Hodas, Pablo López, Jeffrey Polakow, Lubomira Stoilova, and Ernesto Pimentel. A tag-frame system of resource management for proof search in linear-logic programming. In *CSL'02*, volume 2471, pages 167–182, 2002.
21. Zhé Hóu, Ranald Clouston, Rajeev Goré, and Alwen Tiu. Proof search for propositional abstract separation logics via labelled sequents. In *POPL'14*, 2014.
22. Zhé Hóu, Rajeev Goré, and Alwen Tiu. Automated theorem proving for assertions in separation logic with all connectives. In *CADE'15*, 2015.
23. Zhé Hóu, Rajeev Goré, and Alwen Tiu. A labelled sequent calculus for BBI: proof theory and proof search. *JLC*, 2015.
24. Zhé Hóu and Alwen Tiu. Completeness for a first-order abstract separation logic. In *APLAS'16*, pages 444–463, 2016.
25. Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Separation algebra. *AFP*, 2012.
26. Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic (iris 3.0). In *ESOP*, 2017.
27. Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL'17*, pages 205–217, 2017.
28. Peter Lammich and Rene Meis. A separation logic framework for imperative HOL. *AFP'12*.
29. Dominique Larchey-Wendling and Didier Galmiche. Non-deterministic phase semantics and the undecidability of Boolean BI. *ACM TOCL*, 14(1), 2013.
30. Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *SP'13*, pages 415–429, may 2013.
31. Magnus O. Myreen. Separation logic adapted for proofs by rewriting. In *ITP'10*, 2010.
32. Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *BSL'99*, 1999.
33. Jonghyun Park, Jeongbong Seo, and Sungwoo Park. A theorem prover for Boolean BI. In *POPL'13*, pages 219–232, 2013.
34. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74, 2002.
35. David Sanán, Yongwang Zhao, Zhé Hóu, Fuyuan Zhang, Alwen Tiu, and Yang Liu. CSimpl: A rely-guarantee-based framework for verifying concurrent programs. In *TACAS '17*, 2017.
36. Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI'15*, pages 77–87, 2015.
37. Thomas Tuerk. A formalisation of smallfoot in HOL. In *TPHOLs'09*, 2009.
38. Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *Cambridge Technical Report*, volume 687, 2007.
39. Carsten Varming and Lars Birkedal. Higher-order separation logic in Isabelle/HOLCF. *ENTCS*, 218:371–389, 2008.