Separation Logic + Superposition Calculus = Heap Theorem Prover

Juan Antonio Navarro Pérez
Technische Universität München
navarrop@in.tum.de

Andrey Rybalchenko
Technische Universität München
rybal@in.tum.de

Abstract

Program analysis and verification tools crucially depend on the ability to symbolically describe and reason about sets of program behaviors. Separation logic provides a promising foundation for dealing with heap manipulating programs, while the development of practical automated deduction/satisfiability checking tools for separation logic is a challenging problem. In this paper, we present an efficient, sound and complete automated theorem prover for checking validity of entailments between separation logic formulas with list segment predicates. Our theorem prover integrates separation logic inference rules that deal with list segments and a superposition calculus to deal with equality/aliasing between memory locations. The integration follows a modular combination approach that allows one to directly incorporate existing advanced techniques for first-order reasoning with equality, as well as account for additional theories, e.g., linear arithmetic, using extensions of superposition. An experimental evaluation of our entailment prover indicates speedups of several orders of magnitude with respect to the available state-of-the-art tools.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Mechanical theorem proving

General Terms Verification, Logic, Reasoning

Keywords Separation Logic, Superposition

1. Introduction

Program analysis and verification tools crucially depend on the ability to symbolically describe and reason about program behaviors, e.g., using constraints [21], logical formulas/predicates [1, 9, 20, 22, 34], types [25, 28], and abstract domains [8, 26]. Automated deduction techniques can deliver automation support for such reasoning tasks. Today, propositional and Satisfiability Modulo Theory (SMT) solvers are ubiquitous components of software analysis, verification, and debugging tools. These solvers can efficiently deal with practically relevant logical theories of various scalar data types, e.g., fixed length bit-vectors and numbers, as well as uninterpreted functions and arrays [2, 10, 14, 19].

Dealing with programs that manipulate heap-allocated data structures using pointers imposes additional challenges on symbolic reasoning tools. Existing formalisms for tracking the shape of heap graphs, e.g., three valued structures, separation logic, Boolean heaps, and monadic second-order logic of graph types [27, 31–33, 35], demonstrate the feasibility of automated reasoning about heap manipulating programs while providing a wide spectrum of trade-offs w.r.t. automation, precision, efficiency, and applicability.

Separation logic provides a promising foundation for dealing with programs that manipulate the heap following a certain discipline [33]. This discipline can be effectively exploited for manual/tool assisted proof development [18, 28, 37], extended static checking [6, 16], and automatic inference of heap shapes [7, 12, 17, 38]. All these approaches depend on the ability to check logical entailment between separation logic formulas, e.g., for validation of loop invariant candidates or fixpoint detection.

Automation of separation logic—usually extended with recursively defined shape predicates such as lists, trees, or nested containers—relies on decidable sub-classes together with the corresponding proof systems or heuristic approaches based on folding/unfolding strategies for recursive shape definitions [6, 7, 11, 13, 23, 29, 36, 38]. Development of practical entailment checking tools for separation logic is a challenging problem. Since the existing proof systems pursue an intricate interplay between aliasing of memory locations and their occurrence in scope of the shape predicates, the proof search becomes a complex procedure exposed to the non-determinism of the inference rule applications. Heuristic approaches require sophisticated (un)folding strategies, which are difficult to get right in a predictable and robust way. As a result, entailment checkers for separation logic have not reached yet the level of applicability on par with state-of-the-art SAT and SMT solvers.

In this paper, we present an efficient, sound and complete automated theorem prover for checking validity of entailments between separation logic formulas with list segment predicates. In contrast to the existing tools, our approach puts to work the framework of paramodulation-based theorem proving [30]. Instead of representing proof rules as an axiom schema and performing a generic inference-based proof search, by applying the framework we obtain specialized inference rules together with an adequate, optimized rule application strategy, without compromising either soundness or completeness.

Our theorem prover relies on a combination of separation logic inference rules that deal with list segments and a superposition calculus to deal with equality/aliasing between memory locations. Our separation logic inference rules are obtained from an existing proof system for separation logic with list segments [4], which is a basis for various separation logic based tools [6, 11, 16, 36], by factoring out its built-in equality reasoning. We use a standard superposition

PLDI'11, June 4–8, 2011, San Jose, California, USA.

calculus for equality, and rely on its ability to generate (partial) models. The model generation ties together the equality and list segment reasoning. Given an entailment to check, our prover first constructs a model for the equality predicates and then uses this model to disambiguate the heap shape for the subsequent treatment of the list segments. Since the reasoning about the list segments can produce additional equalities, the entire process is repeated with the updated model.

By explicating the connection between the equality and separation logic inferences through the equality models, our theorem prover becomes more efficient than the original proof system. Since the equality model excludes certain heap shapes from consideration, several non-deterministic proof search steps in the original proof system turn into deterministic ones and can be efficiently implemented using a rewriting process.

An experimental evaluation of our entailment prover indicates speedups of several orders of magnitude with respect to the available state-of-the-art tools. Our implementation, which is written in Prolog to allow a declarative specification of the inference rules of the proof system, is able to check thousands of entailments of increasing complexity in seconds rather than minutes.

This paper makes the following contributions:

1. An effective factorization an important existing proof system for separation logic into equality and separation reasoning;
2. An algorithm for proving entailments that exploits this factorization;
3. Application of equality models for disambiguating heap shapes and thus eliminating the corresponding non-determinism from the proof search;
4. An efficient implementation of the theorem prover and its evaluation.

In summary, to the best of our knowledge we present to the first paramodulation-based automated theorem prover for separation logic with list segments.

Our ultimate goal is a theorem prover for an expressive fragment that combines separation logic with other theories useful for program reasoning, including arithmetic and various data structure shapes (such as trees). We believe that building upon a well-understood calculus, such as superposition, offers a viable starting point. Existing extensions of superposition with linear arithmetic [24] and SMT theories [3, 15] suggest immediate steps for extending expressiveness of our current fragment. The increasing extensibility and programmability of state-of-the-art theorem provers, where Z3 is a prominent example, suggest that the presented approach could be implemented within the existing tools.

2. Illustration

We illustrate our theorem proving algorithm using the following example. Assume we want to establish the validity of an entailment E given by

$$c \not\simeq e \wedge \mathsf{lseg}(a,b) * \mathsf{lseg}(a,c) * \mathsf{next}(c,d) * \mathsf{lseg}(d,e) \rightarrow \mathsf{lseg}(b,c) * \mathsf{lseg}(c,e)\ ,$$

where $\simeq$ is equality among program expressions, $\mathsf{next}(x,y)$ is a portion of the heap in the program state where x 'points to' y; $\mathsf{lseg}(x,y)$ is a portion of the heap (possibly empty if $x = y$) containing an acyclic path from x to y following the 'points to' relation; and '$*$' is the union of disjoint portions of the heap. The entailment itself states that any program state, composed of a stack and a heap, which satisfies the conditions on the left-hand side of the formula, should also satisfy the right-hand side.

The first step of our approach is to build a clausal representation of the *negation* of the entailment, i.e. $\neg E$. This set, denoted $\mathrm{cnf}(E)$, has the tree clauses:

$$c \simeq e \rightarrow \emptyset \tag{1}$$

$$\emptyset \rightarrow \mathsf{lseg}(a,b) * \mathsf{lseg}(a,c) * \mathsf{next}(c,d) * \mathsf{lseg}(d,e) \tag{2}$$

$$\mathsf{lseg}(b,c) * \mathsf{lseg}(c,e) \rightarrow \emptyset \tag{3}$$

where $\emptyset \rightarrow F$ asserts a formula to be *true*, while $F \rightarrow \emptyset$ asserts a formula as *false*. More generally, a clause such as $A, B \rightarrow C, D$ asserts that if *all* the facts on the left-hand side are true, then *at least one of* the facts on the right should be true.

From the clause (2), since $\mathsf{lseg}(a,b) * \mathsf{lseg}(a,c)$ represents two disjoint portions of the heap, we know that either $a \simeq b$ or $a \simeq c$, i.e. one of these two segments should be empty. Thus we derive a new clause

$$\emptyset \rightarrow a \simeq b, a \simeq c\ . \tag{4}$$

Now we ask a superposition-based theorem prover to find for us a model for the pure clauses, those which *only* contain equalities, that have been computed so far, i.e. (1) and (4). Such a prover could tell us that assuming $a \simeq c$, and the rest of the variables distinct from each other, would do. In particular $a \simeq c$ satisfies the clause (4).

We apply this information and derive a new clause by 'superposition' of (4) into (2) to obtain

$$\emptyset \rightarrow a \simeq b, \mathsf{lseg}(a,b) * \mathsf{next}(a,d) * \mathsf{lseg}(d,e)\ , \tag{5}$$

where c is replaced with a and a resulting empty list segment is removed. Note that $a \simeq b$ also appears in the clause, intuitively as a reminder that 'we have yet to try' the other alternative. But let us continue by analyzing the heap described in clause (5). Now, from $\mathsf{next}(a,d)$, we know that a is definitely pointing to something in the heap; and thus the disjoint $\mathsf{lseg}(a,b)$ should be empty. So, in either case, $a \simeq b$ and we derive a new pure clause

$$\emptyset \rightarrow a \simeq b\ . \tag{6}$$

We ask our superposition-based prover to find us a new model, and this time it tells us that, in fact, just setting $a \simeq b$ would do. This is enough to satisfy the clause (6), as well as the previous two pure clauses. So we now try a superposition of (6) into (2) to obtain

$$\emptyset \rightarrow \mathsf{lseg}(a,c) * \mathsf{next}(c,d) * \mathsf{lseg}(d,e)\ , \tag{7}$$

this time replacing b with a and removing the empty list segment. This description of a heap now looks all fine, so let us apply a superposition of (6) into clause (3), which comes from the left-hand side of the entailment, to produce

$$\mathsf{lseg}(a,c) * \mathsf{lseg}(c,e) \rightarrow \emptyset\ . \tag{8}$$

Now we see that it is possible to match and 'resolve away' these two complementary heap formulas by 'unfolding' the list segment $\mathsf{lseg}(c,e)$ into the disjoint union of $\mathsf{next}(c,d) * \mathsf{lseg}(d,e)$. This is in fact possible if $c \not\simeq e$; but if otherwise $c \simeq e$ then $\mathsf{lseg}(c,e)$ would be empty while $\mathsf{next}(c,d) * \mathsf{lseg}(d,e)$ is definitely not (at least c is pointing to something). So we have to continue and consider the alternative when

$$\emptyset \rightarrow c \simeq e\ . \tag{9}$$

But this time when we ask our superposition-based prover to find us a model for the pure clauses, we are told that there are no models anymore! In fact the clauses (1) and (9) are contradictory and derive the empty clause. This exercise proves that the set of clauses $\mathrm{cnf}(E)$, equivalent to $\neg E$, is not satisfiable and thus the original entailment *must* be valid. More than this, these inference steps actually constitute a proof of the validity of the entailment (c.f. Figure 4) which we describe later in detail in Section 5.

3. Preliminaries

This section presents preliminary definitions. We introduce notation for dealing with relations and functions, separation logic, as well as equality reasoning using superposition.

Relations We write $x \Rightarrow y$ to denote the ordered pair (x, y) and, for a binary relation R, $x \Rightarrow_R y$ to denote $x \Rightarrow y \in R$. The *domain* $\operatorname{dom} R$ of a relation is the projection of R on its first component, i.e. $\operatorname{dom} R = \{x \mid \exists y : x \Rightarrow_R y\}$. We write $\Rightarrow_R^*$ to denote the reflexive and transitive closure of R; and $\Leftrightarrow_R^*$ for the symmetric, reflexive, and transitive closure of R. An element y is *irreducible* by R if $y \notin \operatorname{dom} R$. Furthermore, y is a *normal form* of x with respect to R if $x \Rightarrow_R^* y$ and y is irreducible.

A binary relation R is *well-founded* if there is no infinite sequence $x_1 \Rightarrow_R x_2 \Rightarrow_R \cdots$; and *confluent* if having $x \Rightarrow_R^* y$ and $x \Rightarrow_R^* y'$ implies that there is a z such that $y \Rightarrow_R^* z$ and $y' \Rightarrow_R^* z$. A well-founded and confluent relation is called *convergent*. Given a convergent relation R, every x has a unique normal form denoted by x_R. Furthermore, we have that $x \Leftrightarrow_R^* y$ if and only if $x_R \equiv y_R$, i.e. two elements are equivalent with respect to R whenever their normal forms are identical.

Functions A *function* f is a relation where $x \Rightarrow_f y$ and $x \Rightarrow_f y'$ implies $y \equiv y'$. If $x \in \operatorname{dom} f$ we write $f(x)$ to denote the (unique) element y such that $x \Rightarrow_f y$. We say that f *is a function from* X *to* Y, denoted $f \colon X \to Y$, if $\operatorname{dom} f = X$ and $f(x) \in Y$ for each $x \in X$. In contrast, f *is a* partial function *form* X *to* Y, denoted $f \colon X \rightharpoonup Y$, if $\operatorname{dom} f \subseteq X$ and $f \colon \operatorname{dom} f \to Y$. We write $f[a \Rightarrow b]$ for a *function update* such that $(f[a \Rightarrow b])(x) = b$ if $x \equiv a$ and $(f[a \Rightarrow b])(x) = f(x)$ otherwise. Semicolons are used to compose function updates, i.e., $f[a \Rightarrow b; c \Rightarrow d] = (f[a \Rightarrow b])[c \Rightarrow d]$. Given a pair of functions f and g, we write $h = f * g$ when $h = f \cup g$ and $\operatorname{dom} f \cap \operatorname{dom} g = \emptyset$, i.e. h is the union of two functions f and g with disjoint domains.

Edges and paths A relation R is an *edge from* x *to* y, denoted $R \colon x \Rightarrow y$, if $R = \{x \Rightarrow y\}$; and a relation R is a *simple path from* x *to* y, denoted $R \colon x \Rightarrow^* y$, if $R = \{x_1 \Rightarrow x_2, \ldots, x_{n-1} \Rightarrow x_n\}$ where $x_1 = x$, $x_n = y$, and $x_i \neq x_j$ if $i \neq j$. Each simple path from x to y is acyclic, and $y \notin \operatorname{dom} R$. Edges and simple paths are functions. For example, $R = \{a \Rightarrow b, b \Rightarrow c, c \Rightarrow d\}$ is a simple path and a function with $\operatorname{dom} R = \{a, b, c\}$; and applying an update $R[b \Rightarrow d; d \Rightarrow c] = \{a \Rightarrow b, b \Rightarrow d, d \Rightarrow c, c \Rightarrow d\}$, the simple path property is invalidated.

3.1 Separation logic

Separation logic is used to reason about programs that manipulate pointer data structures [33]. The following is an abridged presentation of the considered fragment from Berdine et al. [5].

Syntax Let Var be a set of constant symbols together with a distinguished constant symbol nil such that $\mathsf{nil} \notin \mathsf{Var}$. We use Var to represent program variables, and nil represents the null pointer. Let x and y be constant symbols. We say that $x \simeq y$ is a *pure* atom, while $\mathsf{next}(x, y)$ and $\mathsf{lseg}(x, y)$ are *basic spatial* atoms. Then, we write $\mathsf{f}(x, y)$ for a basic spatial atom, where f is either next or f is lseg, and refer to x as the *address* of the atom. Given a multiset of basic spatial atoms that contains elements $S_1, \ldots, S_n$, we say that $S_1 * \cdots * S_n$ is a *spatial atom*; and emp denotes the empty multiset. A spatial atom Σ is *well-formed* if (1) no basic atom in Σ has a nil address, and (2) no two basic atoms in Σ share the same address.

The set of separation logic *formulas* consists of pure atoms, spacial atoms, and their boolean combinations obtained using conjunction $\wedge$, disjunction $\vee$, and negation $\neg$. A *literal* is either an atom A or its negation $\neg A$; an *entailment* $F \to G$ is a shorthand for $\neg F \vee G$; and a *logic equivalence* $F \leftrightarrow G$ abbreviates the formula $F \to G \wedge G \to F$. For a formula F, constant symbols x and y, the substitution $F[y/x]$ replaces all occurrences of x in F with y.

Semantics Let Loc be a set of *memory locations* and nil a special location such that $nil \notin \mathsf{Loc}$. We define $\mathsf{Loc}^+ = \mathsf{Loc} \cup \{nil\}$. An *interpretation* (s, h) for a separation logic formula consists of a function *stack* $s \colon \mathsf{Var} \to \mathsf{Loc}^+$ and a partial function *heap* $h \colon \mathsf{Loc} \rightharpoonup \mathsf{Loc}^+$. Given a stack s, we define the *evaluation* function $\hat{s} \colon \mathsf{Var} \cup \{\mathsf{nil}\} \to \mathsf{Loc}^+$ as $\hat{s} = s[\mathsf{nil} \Rightarrow nil]$.

Given an interpretation (s, h) and an atom A, we define the *satisfaction* relation $s, h \models A$ as follows:

$$
\begin{array}{lll}
s, h \models x \simeq y & \text{if} & \hat{s}(x) = \hat{s}(y)\ , \\
s, h \models \mathsf{next}(x, y) & \text{if} & h \colon \hat{s}(x) \Rightarrow \hat{s}(y)\ , \\
s, h \models \mathsf{lseg}(x, y) & \text{if} & h \colon \hat{s}(x) \Rightarrow^* \hat{s}(y)\ , \\
s, h \models S_1 * \cdots * S_n & \text{if} & \text{exist } h_1, \ldots, h_n \text{ such that} \\
 & & \quad h = h_1 * \cdots * h_n \text{ and} \\
 & & \quad s, h_i \models S_i \text{ for each } 1 \leq i \leq n.
\end{array}
$$

From the above definition follows that the empty spatial atom emp can only be satisfied by an interpretation $(s, \emptyset)$. We extend the satisfaction relation $\models$ to deal with logical connectives in a canonical way, i.e. $s, h \models \neg F$ if (s, h) does not satisfy F, while $s, h \models F_1 \wedge \cdots \wedge F_n$ if (s, h) satisfies every conjunct F_i, and $s, h \models F_1 \vee \cdots \vee F_n$ if (s, h) satisfies some disjunct F_i. We also write $s \models F$ when, for all heaps h, the pair (s, h) is a model of F.

An interpretation that satisfies a formula is called a *model*; a formula is *satisfiable* if it has a model; and a *valid* formula is satisfied by every interpretation. Note that an entailment $F \to G$ is valid if every model of F satisfies G or, equivalently, if $F \wedge \neg G$ is not satisfiable. Program analysis and verification tools often require checking the validity of entailments of the form

$$\Pi \wedge \Sigma \to \Pi' \wedge \Sigma'$$

where Π and Π' are conjunctions of pure literals, while Σ and Σ' are spatial atoms.

3.2 Clausal normal form

We use a clausal form to represent logic formulas. A *clause* is a disjunction of the form

$$\neg A_1 \vee \cdots \vee \neg A_n \vee A_{n+1} \vee \cdots \vee A_{n+m}$$

where each A_i is an atom and at most one of them is a spatial atom. A set of clauses $\mathcal{S}$ represents the conjunction of its elements. Thus, an interpretation satisfies the set of clauses $\mathcal{S}$ if the interpretation satisfies every clause in $\mathcal{S}$. For brevity, we write clauses in the form

$$\Gamma \to \Delta\ ,$$

where the sets $\Gamma = \{A_1, \ldots, A_n\}$ and $\Delta = \{A_{n+1}, \ldots, A_{n+m}\}$ are, respectively, the sets of negative and positive atoms in the clause. Commas are used in clauses to denote union and element inclusion, e.g. $\Gamma, \Gamma', A \to \Delta$ stands for $\Gamma \cup \Gamma' \cup \{A\} \to \Delta$. The symbol $\square$ denotes the *empty clause*, i.e. the clause $\Gamma \to \Delta$ where both $\Gamma = \Delta = \emptyset$. A clause is *pure* if it contains only pure atoms, and is *spatial* otherwise. Given a set of clauses $\mathcal{S}$ we write $\operatorname{Pure}(\mathcal{S})$ to denote the subset of pure clauses contained in $\mathcal{S}$.

Let E be an entailment $\Pi \wedge \Sigma \to \Pi' \wedge \Sigma'$ such that

$$
\begin{aligned}
\Pi &= P_1 \wedge \cdots \wedge P_n \wedge \neg N_1 \wedge \cdots \wedge \neg N_m\ , \\
\Pi' &= P'_1 \wedge \cdots \wedge P'_{n'} \wedge \neg N'_1 \wedge \cdots \wedge \neg N'_{m'}\ .
\end{aligned}
$$

We define a *clausal embedding* function cnf (stands for *clausal normal form*) that takes an entailment E and returns a representation of its negation $\neg E$ given by the set of clauses:

$$
\begin{aligned}
\operatorname{cnf}(E) = \{&\emptyset \to P_1, \quad \ldots, \quad \emptyset \to P_n, \quad \emptyset \to \Sigma \\
&N_1 \to \emptyset, \quad \ldots, \quad N_m \to \emptyset, \quad \Pi'_+, \Sigma' \to \Pi'_-\}\ ,
\end{aligned}
$$

where the sets $\Pi'_+ = \{P'_1, \ldots, P'_{n'}\}$ and $\Pi'_- = \{N'_1, \ldots, N'_{m'}\}$ are, respectively, the sets of atoms occurring positively and negatively in Π'. Because the conjunction of clauses $\mathrm{cnf}(E)$ is logically equivalent to $\neg E$, the entailment E is valid if and only if its clausal embedding $\mathrm{cnf}(E)$ is not satisfiable.

Since it is enough for our purposes, we assume that clauses have at most one spatial atom. Furthermore, in the following we assume that the symbols Γ and Δ represent sets of *pure* atoms, and a spatial atom occurring in a clause (if any) is written explicitly. Therefore a clause can either be a *pure clause*, $\Gamma \to \Delta$, a *positive spatial clause*, $\Gamma \to \Sigma, \Delta$, or a *negative spatial clause*, $\Gamma, \Sigma \to \Delta$.

3.3 Superposition calculus for equality reasoning

A key point of our approach is that it allows the use of existing and well established techniques in automated deduction in order to reason about equality in the context of separation logic. A proof system which fits our needs is that of the *superposition calculus* as presented, for example, by Nieuwenhuis and Rubio [30] (specifically the system $\mathcal{I}$ defined for general clauses in their Section 3.5). This is a proof system with inference rules such as

$$\frac{\Gamma \to x \simeq y, \Delta \qquad \Gamma' \to x \simeq y', \Delta'}{\Gamma, \Gamma' \to y \simeq y', \Delta, \Delta'}$$

where the clauses above the line are the *premises* of the rule and the clause below is its *conclusion*.

Given a set of clauses $\mathcal{S}$, if clauses matching the premises of an inference rule are found in $\mathcal{S}$, then we say that $\mathcal{S}$ *derives* the conclusion of the inference rule. In general we write $\mathcal{S} \vdash_{\mathcal{X}} C$ if it is possible to derive a clause C from a set $\mathcal{S}$ by a successive application of inference rules in $\mathcal{X}$; and write $\mathrm{Cns}_{\mathcal{X}}(\mathcal{S}) = \{C \mid \mathcal{S} \vdash_{\mathcal{X}} C\}$ to denote the set of all *consequences* derived from $\mathcal{S}$ by inferences in the set $\mathcal{X}$. A set of clauses $\mathcal{S}^*$ is *saturated*, with respect to the inferences $\mathcal{X}$, if $\mathcal{S}^* = \mathrm{Cns}_{\mathcal{X}}(\mathcal{S}^*)$.

In the particular case of the proof system $\mathcal{I}$, a number of *superposition* inference rules are defined which derive clauses that logically follow from $\mathcal{S}$ by interpreting '$\simeq$' as an equality relation. To make the system useful in practice, inferences are constrained with respect to an appropriate order $\succ$ over the terms of the language, and only 'maximal' atoms are allowed to participate in inferences. For further details we refer the reader to [30]; in the following we only give a brief summary of the concepts and definitions that are required to develop the work in this paper.

The proof system $\mathcal{I}$ is refutation complete for pure clauses. This means that if $\mathcal{S}^* = \mathrm{Cns}_{\mathcal{I}}(\mathcal{S})$ is the saturation of a set $\mathcal{S}$ of pure clauses, then the empty clause $\square \in \mathcal{S}^*$ if, and only if, the set $\mathcal{S}$ is not satisfiable. Completeness of the proof system is proved showing that if $\square \notin \mathcal{S}^*$ then it is possible to build a model, given by a relation R, as a witness of the satisfiability of $\mathcal{S}$. In this case R satisfies a pure atom $x \simeq y$, denoted $R^* \models x \simeq y$, if $x \Leftrightarrow_R^* y$. Moreover, if R is a convergent relation, then R satisfies an atom $x \simeq y$ if, and only if, their normal forms $x_R \equiv y_R$ are identical.

The relation $R = \mathrm{Gen}(\mathcal{S}^*)$ is defined in terms of a *generating* function Gen (see Definition 3.8 in [30]) which selects some clauses $C \in \mathcal{S}^*$ and *generates* edges $x \Rightarrow_R y$. For our purposes it is not important exactly how are clauses are selected from $\mathcal{S}^*$; as we only rely on the following properties of such relation R.

Lemma 3.1. *Let $\mathcal{S}^*$ be a set of pure clauses saturated with respect to $\mathcal{I}$, and let $R = \mathrm{Gen}(\mathcal{S}^*)$. The relation R is (1) convergent; and (2) if there is an edge $x \Rightarrow_R y$ then $x \succ y$, there is a clause $\Gamma \to x \simeq y, \Delta \in \mathcal{S}^*$—which generated the edge—and $R^* \not\models \Gamma \to \Delta$ (c.f. Lemma 3.9 in [30]).*

We will sometimes write $\langle R, g \rangle = \mathrm{Gen}(\mathcal{S}^*)$ to denote the fact that the generating function provides both the relation R and a mapping g from each edge in R to its corresponding *generating* clause in $\mathcal{S}^*$ satisfying the conditions of item 2 in previous lemma.

Completeness is established showing that, if the proof system $\mathcal{I}$ is unable to derive the empty clause, then R is indeed a model of the set of pure clauses.

Theorem 3.1. *Let $\mathcal{S}$ be a set of pure clauses, let $\mathcal{S}^* = \mathrm{Cns}_{\mathcal{I}}(\mathcal{S})$ and let $R = \mathrm{Gen}(\mathcal{S}^*)$. If $\square \notin \mathcal{S}^*$ then $R^* \models \mathcal{S}$.*

Observe that, in this context, models and interpretations for pure clauses are defined in terms of a relation (w.r.t. $\models$) while, in the previous section, we used a stack and a heap (w.r.t. $\models$). A simple result shows that, in fact, for a pure formula there is a one to one correspondence between its relation and stack models. For this we only need to make the assumption that nil is a minimal element with respect to the order $\succ$ used to generate the relation, i.e. $x \succ \mathsf{nil}$ for every other constant symbol x in the logic.

Definition 3.1. Given a convergent relation R, the induced stack $s_R \colon \mathsf{Var} \to \mathsf{Loc}^+$ is defined as $s_R(x) = \mathit{nil}$ if $x_R \equiv \mathsf{nil}$, and $s_R(x) = \iota(x_R)$ otherwise; where $\iota \colon \mathsf{Var} \to \mathsf{Loc}$ is an arbitrary, but fixed, injection that maps different program variables into different non-nil memory locations.

The condition asking nil to be a minimal element makes sure that if $x \Leftrightarrow_R^* \mathsf{nil}$, then $x_R \equiv \mathsf{nil}$ and s_R maps x to nil. Moreover, by definition $\hat{s}_R(x) = \hat{s}_R(x_R)$. In the following, to ease the notation and if a suitable relation R clear by context, we simply write $\hat{x}$ as a shorthand for the memory location $\hat{s}_R(x)$.

Theorem 3.2. *Let R be a convergent relation and let F be a pure formula. $R^* \models F$ if, and only if, $s_R \models F$.*

Proof. Since the formula F is pure it is enough to consider the case of a pure atom $x \simeq y$ and an arbitrary heap h. The result follows since, from the definition of s_R, the normal forms are identical, i.e. $x_R \equiv y_R$, if and only if the two symbols evaluate to the same memory location, i.e. $\hat{x} = \hat{y}$. □

4. Proof system for separation logic entailments

In the previous section we described the superposition calculus, the proof system $\mathcal{I}$, which allows us to reason about equalities in the pure fragment of the logic. We now focus on introducing the necessary inference rules to reason about the spatial component of our separation logic formulas.

4.1 Spatial reasoning

The proof system $\mathcal{SI}$ is obtained by augmenting $\mathcal{I}$ with a number of additional inference rules shown in Figure 1. These rules are grouped into three main groups of normalization, well-formedness, and unfolding inferences.

- $\mathcal{N}$: The normalization inference N1 allows to use the information in a pure clause to 'rewrite' the spatial atom occurring on a positive spatial clause; while the rule N2 discards trivial basic atoms of the form $\mathsf{lseg}(x, x)$. Rules N3 and N4 are their respective counterparts for negative spatial clauses.
- $\mathcal{W}$: The second group of well-formedness rules, W1 to W5, check for possible inconsistencies that could occur in positive spatial clauses. In particular they make sure that nil is not used as an address in the heap, and that two disjoint parts of the heap do not share a common address.
- $\mathcal{U}$: Finally the unfolding inferences include the rules U1 to U5, which perform a one-step 'unfold' of a basic atom $\mathsf{lseg}(x, z)$ in a negative spatial clause by using the information contained in a positive spatial clause; and the rule SR of *spatial resolu-*

Normalization ($\mathcal{N}$):

$$\mathsf{N1}\ \frac{\Gamma \to x \simeq y, \Delta \qquad \Gamma' \to \Delta', \Sigma}{\Gamma, \Gamma' \to \Delta, \Delta', \Sigma[y/x]} \qquad \mathsf{N2}\ \frac{\Gamma \to \Delta, \mathsf{lseg}(x, x) * \Sigma}{\Gamma \to \Delta, \Sigma}$$

$$\mathsf{N3}\ \frac{\Gamma \to x \simeq y, \Delta \qquad \Gamma', \Sigma \to \Delta'}{\Gamma, \Gamma', \Sigma[y/x] \to \Delta, \Delta'} \qquad \mathsf{N4}\ \frac{\Gamma, \mathsf{lseg}(x, x) * \Sigma \to \Delta}{\Gamma, \Sigma \to \Delta}$$

Well-formedness ($\mathcal{W}$):

$$\mathsf{W1}\ \frac{\Gamma \to \Delta, \mathsf{next}(\mathsf{nil}, y) * \Sigma}{\Gamma \to \Delta} \qquad \mathsf{W2}\ \frac{\Gamma \to \Delta, \mathsf{lseg}(\mathsf{nil}, y) * \Sigma}{\Gamma \to y \simeq \mathsf{nil}, \Delta} \qquad \mathsf{W3}\ \frac{\Gamma \to \Delta, \mathsf{next}(x, y) * \mathsf{next}(x, z) * \Sigma}{\Gamma \to \Delta}$$

$$\mathsf{W4}\ \frac{\Gamma \to \Delta, \mathsf{next}(x, y) * \mathsf{lseg}(x, z) * \Sigma}{\Gamma \to x \simeq z, \Delta} \qquad \mathsf{W5}\ \frac{\Gamma \to \Delta, \mathsf{lseg}(x, y) * \mathsf{lseg}(x, z) * \Sigma}{\Gamma \to x \simeq y, x \simeq z, \Delta}$$

Unfolding ($\mathcal{U}$):

$$\mathsf{U1}\ \frac{\Gamma \to \Delta, \mathsf{next}(x, z) * \Sigma \qquad \Gamma', \mathsf{lseg}(x, z) * \Sigma' \to \Delta'}{\Gamma', \mathsf{next}(x, z) * \Sigma' \to x \simeq z, \Delta'} \qquad \mathsf{U2}\ \frac{\Gamma \to \Delta, \mathsf{next}(x, y) * \Sigma \qquad \Gamma', \mathsf{lseg}(x, z) * \Sigma' \to \Delta'}{\Gamma', \mathsf{next}(x, y) * \mathsf{lseg}(y, z) * \Sigma' \to x \simeq z, \Delta'}\ y \not\equiv z$$

$$\mathsf{U3}\ \frac{\Gamma \to \Delta, \mathsf{lseg}(x, y) * \Sigma \qquad \Gamma', \mathsf{lseg}(x, \mathsf{nil}) * \Sigma' \to \Delta'}{\Gamma', \mathsf{lseg}(x, y) * \mathsf{lseg}(y, \mathsf{nil}) * \Sigma' \to \Delta'}\ y \not\equiv \mathsf{nil}$$

$$\mathsf{U4}\ \frac{\Gamma \to \Delta, \mathsf{lseg}(x, y) * \mathsf{next}(z, w) * \Sigma \qquad \Gamma', \mathsf{lseg}(x, z) * \Sigma' \to \Delta'}{\Gamma', \mathsf{lseg}(x, y) * \mathsf{lseg}(y, z) * \Sigma' \to \Delta'}\ y \not\equiv z$$

$$\mathsf{U5}\ \frac{\Gamma \to \Delta, \mathsf{lseg}(x, y) * \mathsf{lseg}(z, w) * \Sigma \qquad \Gamma', \mathsf{lseg}(x, z) * \Sigma' \to \Delta'}{\Gamma', \mathsf{lseg}(x, y) * \mathsf{lseg}(y, z) * \Sigma' \to z \simeq w, \Delta'}\ y \not\equiv z \qquad \mathsf{SR}\ \frac{\Gamma, \Sigma \to \Delta \qquad \Gamma' \to \Delta', \Sigma}{\Gamma, \Gamma' \to \Delta, \Delta'}$$

Figure 1. Spatial inference rules of the $\mathcal{SI}$ proof system

Well-formedness	*Unfolding*
W1: $\mathsf{next}(\mathsf{nil}, y) * \Sigma \to \emptyset$	U1: $\mathsf{next}(x, z) * \Sigma \to x \simeq z, \mathsf{lseg}(x, z) * \Sigma$
W2: $\mathsf{lseg}(\mathsf{nil}, y) * \Sigma \to y \simeq \mathsf{nil}$	U2: $\mathsf{next}(x, y) * \mathsf{lseg}(y, z) * \Sigma \to x \simeq z, \mathsf{lseg}(x, z) * \Sigma$
W3: $\mathsf{next}(x, y) * \mathsf{next}(x, z) * \Sigma \to \emptyset$	U3: $\mathsf{lseg}(x, y) * \mathsf{lseg}(y, \mathsf{nil}) * \Sigma \to \mathsf{lseg}(x, \mathsf{nil}) * \Sigma$
W4: $\mathsf{next}(x, y) * \mathsf{lseg}(x, z) * \Sigma \to x \simeq z$	U4: $\mathsf{lseg}(x, y) * \mathsf{lseg}(y, z) * \mathsf{next}(z, w) * \Sigma \to \mathsf{lseg}(x, z) * \mathsf{next}(z, w) * \Sigma$
W5: $\mathsf{lseg}(x, y) * \mathsf{lseg}(x, z) * \Sigma \to x \simeq y, x \simeq z$	U5: $\mathsf{lseg}(x, y) * \mathsf{lseg}(y, z) * \mathsf{lseg}(z, w) * \Sigma \to z \simeq w, \mathsf{lseg}(x, z) * \mathsf{lseg}(z, w) * \Sigma$

Figure 2. Separation logic axiom schemas

tion, which resolves away a common spatial atom occurring in complementary spatial clauses.

These inference rules, which can be thought of as coming from the separation logic axiom schemas in Figure 2, are a variation of the proof system for separation logic entailments originally proposed by Berdine et al. [5]. The main difference between the two systems is that, while the proof system from Berdine et al. operates at the level of entailments; our inferences have been rearranged in the form of clauses. This is key to the results presented in this paper, since it allows us to clearly separate the stages of pure equality and spatial reasoning, thus enabling the direct application of a wide array of techniques from the theorem proving community into the context of separation logic.

4.2 Soundness

It is easy to check that all inferences of the $\mathcal{SI}$ proof system are sound; namely if a pair (s, h) of a stack and a heap is a model of the premises of an inference rule, then (s, h) is also a model of the conclusion of the rule.

Theorem 4.1. *Let $E \equiv \Pi \wedge \Sigma \to \Pi' \wedge \Sigma'$ be an entailment, and let $\mathcal{S}^* = \mathrm{Cns}_{\mathcal{SI}}(\mathrm{cnf}(E))$. If $\square \in \mathcal{S}^*$ then the entailment is valid.*

Proof. As all inferences in $\mathcal{SI}$ preserve models, any model of the clausal embedding $\mathrm{cnf}(E)$ would also be a model of the empty clause $\square$. However, since the empty clause doesn't have any models, the set of clauses $\mathrm{cnf}(E)$ is unsatisfiable and, by construction of the embedding, E is valid. $\square$

4.3 Completeness by model generation

The completeness of the $\mathcal{SI}$ proof system is proved by showing that, if the system is unable to derive the empty clause from the clausal embedding of an entailment, then it is possible to build a counterexample for the entailment.

The sketch of the proof is as follows: In section 3.3 we described a construction that generates a convergent relation R as a model of

a satisfiable set of pure clauses and, furthermore, how this relation induces a corresponding stack s_R. What remains to be done is, after a suitable relation R is fixed, to build a corresponding heap that is used to refute a given entailment. For this we introduce the notion of the *graph* of a spatial atom Σ; in particular we show that if Σ is well-formed, then its graph $\mathrm{gr}_R\,\Sigma$ is actually a heap.

Given a spatial atom Σ, we also define its normalization with respect to the relation R and prove that normalization inferences are able to 'normalize' the spatial atom. Then we prove that well-formedness inferences make sure that the normalized Σ_R is indeed well-formed. Finally we show that if unfolding inferences are able to successfully rewrite a spatial atom Σ'_R into another Σ_R, then $s_R \models \Sigma \to \Sigma'$. Otherwise, if the unfolding fails, then it is possible to tweak the graph $\mathrm{gr}_R\,\Sigma$ into a counterexample showing that $s_R, h \not\models \Sigma \to \Sigma'$; and thus allows us to refute entailments.

Now that we have sketched the main idea of the proof, we are ready to delve into its details. We start by defining, as promised, the graph and the normalization of a spatial atom with respect to a given relation R.

Definition 4.1. The *graph* of a basic spatial atom S with respect to a convergent relation R is $\mathrm{gr}_R\,S = \{\hat{x} \Rightarrow \hat{y}\}$ if either

- $S \equiv \mathsf{next}(x, y)$, or
- $S \equiv \mathsf{lseg}(x, y)$ and $\hat{x} \neq \hat{y}$;

and $\mathrm{gr}_R\,S = \emptyset$ otherwise. The *graph* of $\Sigma \equiv S_1 * \cdots * S_n$ is defined as $\mathrm{gr}_R\,\Sigma = \mathrm{gr}_R\,S_1 \cup \cdots \cup \mathrm{gr}_R\,S_n$.

Definition 4.2. Given a spatial atom Σ and a convergent relation R, the *normal form* of Σ with respect to R is the atom Σ_R obtained by (1) replacing each constant symbol x occurring in Σ with its normal form x_R, and then (2) removing any trivial basic atoms of the form $\mathsf{lseg}(x, x)$.

For example, given a relation $R = \{w \Rightarrow y\}$ and an spatial atom $\Sigma \equiv \mathsf{lseg}(x, y) * \mathsf{lseg}(y, w) * \mathsf{lseg}(y, z) * \mathsf{next}(w, y)$, the normalized $\Sigma_R \equiv \mathsf{lseg}(x, y) * \mathsf{lseg}(y, z) * \mathsf{next}(y, y)$, and its graph is given by $\mathrm{gr}_R\,\Sigma = \{\hat{x} \Rightarrow \hat{y}, \hat{y} \Rightarrow \hat{z}, \hat{y} \Rightarrow \hat{y}\}$. As the following lemma shows, this normal form is useful since it allows to quickly determine the relationships between an atom Σ, a relation R and its graph $\mathrm{gr}_R\,\Sigma$.

Lemma 4.1. *Given a spatial atom Σ and a convergent relation R:*

1. $\mathrm{gr}_R\,\Sigma = \mathrm{gr}_R\,\Sigma_R$ *and* $s_R \models \Sigma \leftrightarrow \Sigma_R$*;*
2. *if two constants $x \not\equiv y$ occur in Σ_R, then $R^* \not\models x \simeq y$;*
3. *if Σ_R is well-formed, then $h = \mathrm{gr}_R\,\Sigma$ is a heap and $s_R, h \models \Sigma$;*

Proof. Both claims in the first item follow from the observation that the evaluation $\hat{x} = \hat{s}_R(x) = \hat{s}_R(x_R)$ or, in other words, the stack s_R cannot distinguish between a constant x and its normal form x_R. Furthermore, since trivial atoms the form $\mathsf{lseg}(x, x)$ have an empty graph and are only satisfied by the empty heap, they can be safely removed from Σ_R.

The second item trivially holds since the constants x, y in Σ_R are already in their normal forms and, if $x \not\equiv y$, then $R^* \not\models x \simeq y$.

For the last item let $\Sigma_R \equiv S_1 * \cdots * S_n$, and take $h_i = \mathrm{gr}_R\,S_i$ for each basic spatial atom in Σ_R. Note that if x_i is the address of the basic atom S_i, and since there are no trivial atoms in Σ_R, then necessarily $\mathrm{dom}\,h_i = \{\hat{x}_i\}$. Because Σ_R is well-formed it follows that $h = h_1 * \cdots * h_n$ is indeed a heap (i.e. $\mathsf{nil} \notin \mathrm{dom}\,h$ and any two heaps h_i, h_j with $i \neq j$ have disjoint domains). Moreover, by construction $s_R, h_i \models S_i$ and, therefore, $s_R, h \models \Sigma_R$. But, by definition, $h = \mathrm{gr}_R\,\Sigma_R$ and, from the first item, it follows that both $h = \mathrm{gr}_R\,\Sigma$ and $s_R, h \models \Sigma$ as well. □

Definition 4.3. Given a spatial clause $C \equiv \Gamma \to \Delta, \Sigma$, a relation R is said to *force* the spatial atom Σ, denoted $R, C \rightsquigarrow \Sigma$, if R is not a model of the pure part of the clause, i.e. $R^* \not\models \Gamma \to \Delta$. Similarly, for a clause $C \equiv \Gamma, \Sigma \to \Delta$, we write $R, C \rightsquigarrow \neg\Sigma$ if $R^* \not\models \Gamma \to \Delta$.

Intuitively, if $R, C \rightsquigarrow \Sigma$ (resp. $\neg\Sigma$) then the model induced by R forces the spatial atom Σ to be *true* (resp. *false*) in order to satisfy the clause C. The following three lemmas are at the core of our main result, as they explicate the role of normalization, well-formedness and unfolding inference rules in the $\mathcal{SI}$ proof system.

Lemma 4.2 (Normalization)**.** *Let $\mathcal{S}^*$ be a set of pure clauses saturated with respect to $\mathcal{I}$, let $R = \mathrm{Gen}(\mathcal{S}^*)$, and let C be a spatial clause. If $R, C \rightsquigarrow \Sigma$ (resp. $\neg\Sigma$) then there is a clause C' such that $\mathcal{S}^*, C \vdash_{\mathcal{N}} C'$ and $R, C' \rightsquigarrow \Sigma_R$ (resp. $\neg\Sigma_R$).*

Proof. More generally let $\langle R, g\rangle = \mathrm{Gen}(\mathcal{S}^*)$ be the generated relation and its selection function. By definition, if $R, C \rightsquigarrow \Sigma$ for a clause $C \equiv \Gamma \to \Delta, \Sigma$, then the relation $R \not\models \Gamma \to \Delta$. Now, if there is a constant symbol x occurring in Σ such that $x \not\equiv x_R$, there should be a rule $x \Rightarrow_R y$ for some y and, from Lemma 3.1, a clause $D \equiv g(x \Rightarrow_R y) = \Gamma' \to x \simeq y, \Delta' \in \mathcal{S}^*$ such that $R^* \not\models \Gamma' \to \Delta'$. An application of the normalization rule $\mathsf{N1}$ between C and D yields a clause C_1 such that $\mathcal{S}^*, C \vdash_{\mathcal{N}} C_1$, also $R, C_1 \rightsquigarrow \Sigma[y/x]$ and the spatial atom is *closer* to its normal form. Iterating this process all constant symbols are rewritten to their normal form. Finally, trivial atoms are removed with the rule $\mathsf{N2}$ to yield a clause C' such that $S, C \vdash_{\mathcal{N}} C'$ and $R, C' \rightsquigarrow \Sigma_R$.

An analogous argument using normalization rules $\mathsf{N3}$ and $\mathsf{N4}$ shows that if $R, C \rightsquigarrow \neg\Sigma$ then a clause C' is derived such that $S, C \vdash_{\mathcal{N}} C'$ and $R, C' \rightsquigarrow \neg\Sigma_R$. □

As a result of this lemma, for a spatial clause C we allow ourselves to write $\mathit{Norm}(\langle R, g\rangle;\, C)$ to denote any such clause C' where the spatial atom Σ has been normalized to Σ_R. Furthermore, from the proof it follows that given the relation R and the clause selection function g, it is possible to compute a normalized C' in a straightforward way without requiring any search.

The following lemma deals with rules for well-formedness, and shows how they ensure that a normalized atom Σ_R is indeed well-formed. To simplify the notation, we use $\mathrm{PCns}_{\mathcal{X}}(\mathcal{S})$ as a shorthand for the set $\mathrm{Pure}(\mathrm{Cns}_{\mathcal{X}}(\mathcal{S}))$ of *pure* clauses that are derived from $\mathcal{S}$ with inferences in $\mathcal{X}$.

Lemma 4.3 (Well-formedness)**.** *Let $\mathcal{S}^*$ be a set of pure clauses saturated with respect to $\mathcal{I}$, let $R = \mathrm{Gen}(\mathcal{S}^*)$, and let C be a spatial clause such that $R, C \rightsquigarrow \Sigma_R$. If $\mathrm{PCns}_{\mathcal{W}}(\{C\}) \subseteq \mathcal{S}^*$ and $\square \notin \mathcal{S}^*$ then Σ_R is well-formed.*

Proof. Since $R, C \rightsquigarrow \Sigma_R$, the clause C is of the form $\Gamma \to \Delta, \Sigma_R$ and $R^* \not\models \Gamma \to \Delta$. Moreover, since $\square \notin \mathcal{S}^*$, by Theorem 3.1 the relation $R^* \succeq \mathcal{S}^*$ and, by hypothesis, $R^* \succeq \mathrm{PCns}_{\mathcal{W}}(\{C\})$.

Now, if Σ_R were not well-formed then one of the inference rules in $\mathcal{W}$ would apply—either $\mathsf{W1}$ or $\mathsf{W2}$ if there is a nil address in Σ_R, or one of $\mathsf{W3}$, $\mathsf{W4}$ or $\mathsf{W5}$ if there are two basic spatial atoms with the same address—to derive a pure clause D such that $C \vdash_{\mathcal{W}} D$ but $R^* \not\models D$, and thus contradicting the fact that the relation $R^* \succeq \mathrm{PCns}_{\mathcal{W}}(\{C\})$.

For example, if there is a basic atom $\mathsf{lseg}(\mathsf{nil}, y)$ occurring in Σ_R then $\mathsf{nil} \not\equiv y$, because the atom is not trivial, and by Lemma 4.1 it follows that $R^* \not\models \mathsf{nil} \simeq y$. Applying the rule $\mathsf{W2}$ to C, a pure clause is derived such that $R^* \not\models \Gamma \to \mathsf{nil} \simeq y, \Delta$. □

Observe that there is no search involved in the computation of $\mathrm{PCns}_{\mathcal{W}}(\{C\})$, as it is enough to match C against the premises of rules in $\mathcal{W}$ to immediately compute its consequences. Finally, the last of these lemmas formalzes the application of unfolding inferences in the proof system $\mathcal{SI}$.

Lemma 4.4 (Unfolding). *Let $\mathcal{S}^*$ be a set of pure clauses saturated with respect to $\mathcal{I}$, let $R = \mathrm{Gen}(\mathcal{S}^*)$, and let C, C' be spatial clauses such that $R, C \rightsquigarrow \Sigma_R$ and $R, C' \rightsquigarrow \neg\Sigma'_R$. If the set $\mathrm{PCns}_{\mathcal{U}}(\{C, C'\}) \subseteq \mathcal{S}^*$, $\square \notin \mathcal{S}^*$, and Σ_R is well-formed, then there is an h such that $s_R, h \not\models \Sigma \to \Sigma'$.*

Proof. Since the atom Σ_R is well-formed, from Lemma 4.1, it follows that $s_R, h \models \Sigma_R$ where $h = \mathrm{gr}_R\, \Sigma_R$. Now, if $s_R, h \not\models \Sigma'_R$ we are done, as our h satisfies the conditions of the lemma.

Assume that otherwise, $s_R, h \models \Sigma'_R$. Now, take the normalized atom $\Sigma'_R \equiv S'_1 * \cdots * S'_n$ with no trivial basic atoms, it follows that the graph $h = \mathrm{gr}_R\, \Sigma_R = h'_1 * \cdots * h'_n$ for some non-empty heaps h'_i such that $s_R, h'_i \models S'_i$. In particular, since $\mathrm{gr}_R\, \Sigma_R$ is the separated union of graphs for basic atoms in Σ_R, for each h'_i there is an atom T_i occurring in Σ_R for which $h'_i = \mathrm{gr}_R\, T_i$ and, moreover, the spatial atom $\Sigma_R \equiv T_1 * \cdots * T_n$.

We now show that either: (a) unfolding inference rules are able to derive from $\{C, C'\}$ a clause C'' such that $R, C'' \rightsquigarrow \neg\Sigma_R$ by 'rewriting'each atom S'_i in Σ'_R into the corresponding T_i in Σ_R or (b) h can be fixed to satisfy the statement of the theorem. Proceed for each S'_i as follows:

- $S'_i \equiv \mathsf{next}(x, y)$. Then the graph $\mathrm{gr}_R\, T_i \colon \hat{x} \Rightarrow \hat{y}$ is an edge and the spatial atom $T_i \equiv \mathsf{f}(x, y)$. If the symbol $\mathsf{f} \equiv \mathsf{next}$ then, since $S'_i \equiv T_i$, the atom S_i has been rewritten into T_i. For the alternative, $\mathsf{f} \equiv \mathsf{lseg}$, the case (b) holds since then the pair $(s_R, h[\hat{x} \Rightarrow w; w \Rightarrow \hat{y}])$, where $w \notin \mathrm{dom}\, h$, would be a model of Σ but not of Σ'.
- $S'_i \equiv \mathsf{lseg}(x, z)$. Then the graph $\mathrm{gr}_R\, T_i \colon \hat{x} \Rightarrow^* \hat{z}$ is a simple path and the spatial atom $T_i \equiv \mathsf{f}_1(x_1, x_2) * \cdots * \mathsf{f}_k(x_k, z)$ where $x_1 \equiv x$. For each $1 \leq j \leq k$, let

  $$T_{i,j} = \mathsf{f}_1(x_1, x_2) * \cdots * \mathsf{f}_{j-1}(x_{j-1}, x_j) * \mathsf{lseg}(x_j, z)\ ,$$

  i.e. the first $j-1$ basic spatial atoms from T_i followed by the atom $\mathsf{lseg}(x_j, z)$. Note that $T_{i,1} \equiv S'_i$, while $T_{i,k}$ is identical to T_i except possibly for f_k. For each $1 \leq j < k$ we will prove that if $\{C, C'\} \vdash_{\mathcal{U}} C_j$ for a clause such that $R, C_j \rightsquigarrow \neg\Sigma_R[T_{i,j}/S'_i]$; then a clause C_{j+1} such that $R, C_{j+1} \rightsquigarrow \neg\Sigma_R[T_{i,j+1}/S'_i]$ can be derived by unfolding S'_i and 'walking' over T_i according to the following two cases:
  - $\mathsf{f}_j \equiv \mathsf{next}$. Since $x_j \not\equiv z$ because the graph of T_i is a simple path, $R^* \not\models x_j \simeq z$ and the rule U2 rewrites $T_{i,j}$ into $T_{i,j+1}$.
  - $\mathsf{f}_j \equiv \mathsf{lseg}$. If $z \not\equiv \mathsf{nil}$ then the inference rule U3 does the rewrite, and if z is the address of an atom in Σ_R then either U4 or U5 should apply. In the remaining case—when the symbol $y \not\equiv \mathsf{nil}$, z is not the address of any basic atom in Σ_R and therefore $\hat{z} \notin \mathrm{dom}\, h$—again (b) holds since then the interpretation $(s_R, h[\hat{y}_{k_1} \Rightarrow \hat{z}; \hat{z} \Rightarrow \hat{y}_k])$ is a model of Σ and not of Σ'.

  So we have shown that it is possible to rewrite S'_i into $T_{i,k}$, which is identical to T_i except if $\mathsf{f}_k \equiv \mathsf{next}$. But in such case, since $x_k \not\equiv z$, the unfolding rule U1 rewrites $T_{i,k}$ into T_i.

From the previous argument if (b) is proved then again we are done, otherwise the rewrite in (a) is successful and we derive a clause C'' such that $R, C'' \rightsquigarrow \neg\Sigma_R$. But from hypothesis we also had $R, C \rightsquigarrow \Sigma_R$ and an application of the spatial resolution rule SR derives a pure clause D such that $R^* \not\models D$. However this is a contradiction, since by construction $D \in \mathrm{PCns}_{\mathcal{U}}(\{C, C'\}) \subseteq \mathcal{S}^*$, and the relation R was supposed to be a model of D. □

One more time, from the argument of the proof it follows that there is no search required in order to compute the clauses, if any, in the set $\mathrm{PCns}_{\mathcal{U}}(\{C, C'\})$. This is a key property of our proof system: since $R, C \rightsquigarrow \Sigma_R$ and Σ_R is normalized, the basic atoms in Σ_R guide the application of unfolding inferences in a deterministic manner; *thus eliminating the nondeterminism due to the associativity-commutativity of the separating conjunction*. After having finished the proofs of these lemmas, we are ready to show the completeness of our proof system $\mathcal{SI}$.

```
1:  function prove(Π ∧ Σ → Π′ ∧ Σ′)
2:      S := Pure(cnf(Π ∧ Σ → Π′ ∧ Σ′))
3:      do
4:          repeat
5:              S* := Cns_I(S)
6:              if □ ∈ S* return valid
7:              ⟨R, g⟩ := Gen(S*)
8:              C := Norm(⟨R, g⟩; ∅ → Σ)
9:              S := S* ∪ PCns_W({C})
10:         until S = S*
11:         if R ⋡ Π′ return c-example(S*, C)
12:         C′ := Norm(⟨R, g⟩; Π′+, Σ′ → Π′−)
13:         S := S* ∪ PCns_U({C, C′})
14:         if S = S* return c-example(S*, C, C′)
15:     loop
```

Figure 3. Algorithm for entailment checking

Theorem 4.2 (Completeness). *Let $E \equiv \Pi \wedge \Sigma \to \Pi' \wedge \Sigma'$ be an entailment, and let $\mathcal{S}^* = \mathrm{Cns}_{\mathcal{SI}}(\mathrm{cnf}(E))$. If $\square \notin \mathcal{S}^*$ then the entailment is not valid (i.e. there is a counterexample).*

Proof. Let $\mathcal{S}^*_{\simeq} = \mathrm{Pure}(\mathcal{S}^*)$ be the subset of the pure clauses in the saturated set $\mathcal{S}^*$. Since $\square \notin \mathcal{S}^*$ then neither $\square \notin \mathcal{S}^*_{\simeq}$ and, by Theorem 3.1, the relation R is a model of $\mathcal{S}^*_{\simeq}$. Moreover, from the definition of $\mathrm{cnf}(E)$, the relation $R^* \not\models \Pi$.

Since the clause $C \equiv \emptyset \to \Sigma \in \mathrm{cnf}(E)$, and $R^* \not\models \square$ we have that $R, C \rightsquigarrow \Sigma$. From Lemma 4.2, it follows that there is a spatial clause C_{n}, derived by normalization inferences, such that $R, C_{\mathsf{n}} \rightsquigarrow \Sigma_R$. Furthermore, since $\square \notin \mathcal{S}^*_{\simeq}$ and $\mathcal{S}^*$ is already saturated with respect to $\mathcal{W}$ rules, by Lemma 4.3 it follows that the spatial atom Σ_R is well-formed. Then, from Lemma 4.1, there is a heap h such that $s_R, h \models \Sigma$. Also, since $R^* \not\models \Pi$, from Theorem 3.2 we get that $s_R, h \models \Pi$. Indeed the pair (s_R, h) is a model of $\Pi \wedge \Sigma$.

Now, if $R^* \not\models \Pi'$ we are done since, also from Theorem 3.2, $s_R, h \not\models \Pi'$ and we have found a counterexample. Otherwise, if $R^* \not\models \Pi'$ then, equivalently, $R^* \not\models \Pi'_+ \to \Pi'_-$ and, since the clause $C' \equiv \Pi'_+, \Sigma \to \Pi'_- \in \mathrm{cnf}(E)$, we have that $R, C' \rightsquigarrow \neg\Sigma$. Again by Lemma 4.2 there is a clause C'_{n} derived from C' by normalization rules such that $R, C'_{\mathsf{n}} \rightsquigarrow \neg\Sigma'_R$. In this case all the assumptions of Lemma 4.4 are satisfied and a suitable h' such that the pair (s_R, h') invalidates the entailment should therefore exist. □

5. Algorithm for entailment checking

In the previous section we established the soundness and completeness of our proof system for the fragment of separation logic under consideration. We now turn our attention to use the insight gained in the construction of such a proof, in order to build an efficient algorithm for proving the validity of entailments.

Along the lines of the proof of completeness, the algorithm for checking an entailment $E \equiv \Pi \wedge \Sigma \to \Pi' \wedge \Sigma'$, which is given as pseudocode in Figure 3, emerges from a careful interleaving of inferences using: superposition reasoning for pure clauses (line 5), normalization inferences (lines 8 and 12), well-formedness inferences (line 9), and unfolding inferences (line 13).

The algorithm works by incrementally building a set $\mathcal{S}$ with all the *pure* clauses that are derived from $\mathrm{cnf}(E)$. Initially $\mathcal{S}$ is set

Figure 4. Proof tree for the entailment: $c \not\simeq e \wedge \mathsf{lseg}(a,b) * \mathsf{lseg}(a,c) * \mathsf{next}(c,d) * \mathsf{lseg}(d,e) \to \mathsf{lseg}(b,c) * \mathsf{lseg}(c,e)$

to the subset of pure clauses in the clausal embedding of E, i.e. $\mathcal{S} := \{\emptyset \to A \mid A \in \Pi_+\} \cup \{A \to \emptyset \mid A \in \Pi_-\}$.

The inner loop in lines 4–10 will first saturate the set $\mathcal{S}$ with superposition inferences using the proof system $\mathcal{I}$ and, if the empty clause is derived in such process, the entailment is proved as valid. Otherwise the algorithm continues using the pair $\langle R, g\rangle$ generated from the saturated set $\mathcal{S}^*$ to normalize the clause $\emptyset \to \Sigma \in \mathrm{cnf}(E)$ into a clause of the form $C := \Gamma \to \Sigma_R, \Delta$. Finally $\mathcal{S}$ is updated with the set of pure clauses that are derived from C using well-formedness inference rules in $\mathcal{W}$. The loop repeats until either the entailment is proved valid, or a fixpoint is reached when $\mathcal{S} = \mathcal{S}^*$. Moreover, in the later case, we know that the spatial atom Σ_R in C must be well-formed.

If we exit the inner loop, the clause $\Pi'_+, \Sigma' \to \Pi'_- \in \mathrm{cnf}(E)$ is then normalized to a clause of the form $C' := \Gamma', \Sigma'_R \to \Delta'$, and pure clauses derived from $\{C, C'\}$ using unfolding inference rules in $\mathcal{U}$ are added to $\mathcal{S}$. If no new clauses are derived, then from Lemma 4.4 it is possible to build a counterexample for the entailment. Otherwise a new pure clause has been discovered and the main loop of the function iterates.

Theorem 5.1. *The function* $\mathrm{prove}(E)$ *in Figure 3 terminates, is sound and complete.*

Proof. Soundness of the algorithm immediately follows from the soundness of the $\mathcal{SI}$ proof system. And the algorithm terminates since the growing set $\mathcal{S}$ is bounded by $\mathrm{PCns}_{\mathcal{SI}}(E)$, which is itself bounded by the finite number of distinct pure clauses which can be written with the constant symbols occurring in E.

Completeness follows from the following invariants: The normalized clause C computed in line 8 satisfies $R, C \rightsquigarrow \Sigma_R$ (c.f. Lemma 4.2); after exiting the inner loop in line 10 also $R, C \rightsquigarrow \Sigma_R$ and Σ_R is well-formed (c.f. Lemma 4.3). In particular, upon exiting the loop, it is proved that $\Pi \wedge \Sigma$ is satisfiable by a model (s_R, h); if at this point $R^* \not\models \Pi'$ we have found a counterexample. Otherwise the C' computed in line 12 satisfies $R, C \rightsquigarrow \Sigma'_R$ (c.f. Lemma 4.2); and if after line 13 a fix-point is detected, we have also found a counterexample (c.f. Lemma 4.4). □

In order to get a more concrete feeling of how the algorithm works, let us run again through the example from Section 2 to establish the validity of the entailment E given by

$$c \not\simeq e \wedge \mathsf{lseg}(a,b) * \mathsf{lseg}(a,c) * \mathsf{next}(c,d) * \mathsf{lseg}(d,e) \to \mathsf{lseg}(b,c) * \mathsf{lseg}(c,e)\ .$$

The algorithm begins with $\mathcal{S} = \{D_1\}$, where $D_1 \equiv c \simeq e \to \emptyset$ is the only pure clause in $\mathrm{cnf}(E)$. Then $\mathcal{S}^* = \mathrm{Cns}_{\mathcal{I}}(\mathcal{S}) = \mathcal{S}$ is trivially computed. No clauses are selected since $R = \mathrm{Gen}(\mathcal{S}^*) = \emptyset$ is a model of $\mathcal{S}^*$; and normalization leaves the clause $C := \emptyset \to \Sigma$, where $\Sigma \equiv \mathsf{lseg}(a,b) * \mathsf{lseg}(a,c) * \mathsf{next}(c,d) * \mathsf{lseg}(d,e)$, intact.

Now the well-formedness rule W5 derives from C a new pure clause $D_2 \equiv \emptyset \to a \simeq b, a \simeq c$. The inner loop iterates and the saturated set $\mathcal{S}^* := \mathrm{Cns}_{\mathcal{I}}(\{D_1, D_2\})$ is computed. Assume an order $a \prec b \prec c$ and thus, D_2 is selected to generate the relation $R = \mathrm{Gen}(\mathcal{S}^*) = \{c \Rightarrow a\}$. Now normalization rewrites the input clause $\emptyset \to \Sigma$ into $C := \emptyset \to a \simeq b, \Sigma_R$, where the spatial atom $\Sigma_R \equiv \mathsf{lseg}(a,b) * \mathsf{next}(a,d) * \mathsf{lseg}(d,e)$.

This time, the inference rule W4 derives from C a third pure clause $D_3 \equiv \emptyset \to a \simeq b$. Another iteration of the inner loop now updates $\mathcal{S}^* := \mathrm{Cns}_{\mathcal{I}}(\{D_1, D_2, D_3\})$, which are satisfied by selecting D_3 and producing $R = \mathrm{Gen}(\mathcal{S}^*) = \{b \Rightarrow a\}$; while normalization rewrites the input clause $\emptyset \to \Sigma$ into $C := \emptyset \to \Sigma_R$ where $\Sigma_R \equiv \mathsf{lseg}(a,c) * \mathsf{next}(c,d) * \mathsf{lseg}(d,e)$.

At this point no more well-formedness rules in $\mathcal{W}$ apply, we have reached a fixpoint. Note that, indeed, Σ_R is finally a well-formed atom. We exit the inner loop and proceed to normalize the

Vars.	P_{lseg}	$P_{\not\simeq}$	% Valid	Time (secs.) jStar	Smallfoot	SLP
10	0.10	0.20	54	300.08	13.46	1.02
11	0.09	0.15	50	(16%)	27.02	1.06
12	0.09	0.11	54	(3%)	86.75	1.46
13	0.08	0.11	55	(1%)	119.66	1.48
14	0.07	0.11	53	(10%)	153.93	1.74
15	0.06	0.12	52	(8%)	155.72	1.73
16	0.05	0.17	50	(9%)	140.95	1.59
17	0.05	0.13	54	(3%)	258.31	2.15
18	0.04	0.20	49	(11%)	176.46	1.88
19	0.04	0.15	50	(5%)	383.26	2.15
20	0.04	0.11	52	(0%)	(90%)	2.65

Table 1. Benchmarking 1000 random instances of $F \to \bot$.

Vars.	P_{next}	% Valid	Time (secs.) jStar	Smallfoot	SLP
10	0.70	53	(10%)	28.50	1.29
11	0.69	53	(3%)	56.72	1.24
12	0.69	53	(1%)	106.69	1.49
13	0.70	53	(1%)	166.53	1.82
14	0.69	53	(0%)	271.48	2.19
15	0.69	52	(0%)	404.61	2.49
16	0.69	49	(0%)	(80%)	2.68
17	0.71	53	(0%)	(79%)	3.06
18	0.70	49	(0%)	(49%)	3.60
19	0.70	53	(0%)	(30%)	3.65
20	0.70	49	(0%)	(19%)	3.87

Table 2. Benchmarking 1000 random instances of $F \to G$.

clause $\mathsf{lseg}(b, c) * \mathsf{lseg}(c, e) \to \emptyset \in \mathrm{cnf}(E)$ into the corresponding clause $C' := \Sigma'_R \to \emptyset$ where $\Sigma'_R \equiv \mathsf{lseg}(a, c) * \mathsf{lseg}(c, e)$. Then the application of unfolding is successful since, from C and C', we derive the new pure clause $D_4 \equiv \emptyset \to e \simeq c$, add it to $\mathcal{S}$, and start a new iteration of the main loop.

However now the set $\mathcal{S}^* = \mathrm{Cns}_{\mathcal{I}}(\mathcal{S})$ is inconsistent, since the clauses $D_1, D_4 \in \mathcal{S}$ are contradictory and therefore $\square \in \mathcal{S}^*$. This proves that the original entailment E is indeed valid. Furthermore, from the run of this algorithm it is possible to reconstruct a proof for the unsatisfiability of $\mathrm{cnf}(E)$, which is shown in Figure 4. The figure shows how the empty clause $\square$ is derived from the set of input clauses $\mathrm{cnf}(E)$ using the appropriate inference rules from the $\mathcal{SI}$ proof system. Each derived clause is numbered, and vertical dots are used to denote the reuse of a clause previously derived in the tree. The 'active' part of each clause, on which the respective inference operates, is also highlighted with a frame box.

6. Experimental evaluation

In order to empirically evaluate of the algorithm that we described in the previous section, we implemented a theorem proving tool for separation logic entailments that we call SLP. The tool is implemented in Prolog and consists of about a hundred lines of code to encode the logic of the algorithm in Figure 3 together with a declarative specification of the inference rules of the $\mathcal{SI}$ proof system. The pure model finder is implemented in about fifty lines of code, and an additional four hundred lines are for reading the program input, encoding into and manipulating internal data structures, as well as pretty printing.

Copies	Time (secs.) jStar	Smallfoot	SLP
1	0.30	0.01	0.11
2	0.37	0.07	0.06
3	0.89	1.03	0.08
4	2.65	9.53	0.13
5	9.44	55.85	0.38
6	38.09	245.69	2.37
7	166.86	(64%)	20.83
8	(30%)	(15%)	212.17

Table 3. Benchmarking 'clones' of Smallfoot examples.

In a series of benchmarks, we compare the performance of our tool with two other available state-of-the-art verification tools: Smallfoot [6], and jStar [16]. Both of these tools take as input annotated functions with pre- and post-conditions (respectively in C or in Java) and, through a process of symbolic execution, generate a number of verification conditions that have to be discharged in order to prove the validity of the program specifications.

Each of this verifications conditions corresponds to an entailment check which could be alternatively discharged by our tool. Note that, currently, SLP does not perform verification or symbolic execution or programs. Indeed, what we want to compare is only the performance of SLP against the entailment checkers implemented in Smallfoot and jStar. This is easily done with jStar, which provides a command `run_logic` with direct access to the prover; while, in the case of Smallfoot, queries to the entailment checker can be faked with minimal overhead by asking it to verify no-op functions of the form

`fun(`*vars*`) [`F`] { } [`G`]`

which are valid if, and only if, the entailment $F \to G$ holds.

In order to benchmark these separation logic provers on a wide class of formulas with increasing complexity, we first generated some synthetic entailments according to two different random distributions. The first of these distributions generates entailment checks of the form $\Pi \wedge \Sigma \to \bot$, with n program variables from the set $\mathsf{Var} = \{x_1, \ldots, x_n\}$, as follows:

- if $i \neq j$, with probability P_{lseg} include $\mathsf{lseg}(x_i, x_j)$ in Σ;
- if $i < j$, with probability $P_{\not\simeq}$ include $x_i \not\simeq x_j$ in Π.

Note that lseg edges, as well as pure inequalities, are chosen at random independently of each other. Furthermore, this kind of entailments can be proved (or refuted) by superposition, normalization and well-formedness rules only (i.e. the inner loop in the algorithm of Figure 3). Specifically: if these rules are able to prove that $\Pi \wedge \Sigma$ is inconsistent, then the entailment is valid; conversely if these rules are enough to build a model for $\Pi \wedge \Sigma$, then the entailment has a counterexample.

Moreover, note that for fixed values of n and P_{lseg}, we can use the parameter $P_{\not\simeq}$ to tune the proportion of generated entailments that turn out to be valid. When $P_{\not\simeq} = 0$, then Π is empty, there is a trivial model for Σ—the one which makes all variables equal to nil—and the entailment is invalid. Conversely, if $P_{\not\simeq}$ is high enough, adding inequalities between pairs of variables constrains the space of models for $\Pi \wedge \Sigma$ to the point where, with high probability, it becomes empty and renders the entailment valid. We use this feature to calibrate the model so that, roughly, about half of the generated entailments are valid.

Each row in Table 1 shows the time spent by each of the provers—jStar, Smallfoot, and SLP—to check 1000 randomly generated entailments with parameters ranging from 10 to 20 vari-

ables. Each time the provers were run with a single input file containing all the entailments to check, so that we don't have to pay the time of starting up the process a thousand times. When a prover timed out after 10 minutes of execution, we show in parenthesis the percentage of the instances that were successfully solved before hitting the time limit. SLP outperforms the other proves by several orders of magnitude, solving thousands of instances not in minutes but in a few seconds.

Our second random distribution stresses the role of unfolding inferences in the entailment checks. Again we assume a fixed set of n program variables $\mathsf{Var} = \{x_1, \ldots, x_n\}$, and we let π be a random permutation of the indices of these variables such that $\pi(i) \neq i$. Then the spatial atom

$$\Sigma = \mathsf{f}(x_1, x_{\pi(1)}) * \cdots * \mathsf{f}(x_n, x_{\pi(n)})$$

where each f is randomly selected as either next, with probability p_{next}, or lseg, with probability $1 - p_{\mathsf{next}}$. Note that in this case, by construction, Σ is already well-formed. Now Σ' starts as a copy of Σ, and then we randomly 'fold' some paths in Σ'. For this we randomly pick a variable x_i and, if it appears as the address of a basic atom that has not been folded yet, the longest path stating from x_i by yet unfolded atoms is completely folded into an atom $\mathsf{lseg}(x_i, x_i^*)$, where x_i^* is the last variable reached through this path. This operation is repeated until all basic atoms have been folded. Finally we ask to provers to check the validity of $\Sigma \to \Sigma'$.

In this case the parameter p_{next} can be used to tune the proportion of valid and invalid entailments. Table 2 shows the time spent by the provers checking 1000 randomly generated entailments with parameters ranging from 10 to 20 variables. Again SLP outperforms the other provers, which have trouble figuring out the correct unfolding required to prove/disprove each of these entailments.

Our last set of benchmarks comes from the examples included in the Smallfoot distribution. This includes some 18 'real-life' list manipulating programs, along with some specifications to prove. Note that these also include some examples with arbitrary data fields (other than a next pointer) which our implementation can already handle. This involves slight modifications to the $\mathcal{SI}$ rules, not shown for brevity. For the verification of all of these programs, Smallfoot generates about 209 verification conditions that have to be discharged. These are actually some rather simple entailments to check, as all the three separation logic provers are able to tackle them all in under a second.[1] In order to create more challenging benchmarks, which still have some resemblance with those arising from real-life applications, we make use of a simple 'cloning' technique. For each verification condition $\Pi \wedge \Sigma \to \Pi' \wedge \Sigma'$ generated by Smallfoot, we generate the equivalent but more challenging

$$\Pi_1 \wedge \cdots \wedge \Pi_n \wedge \Sigma_1 * \cdots * \Sigma_n \to \Pi'_1 \wedge \cdots \wedge \Pi'_n \wedge \Sigma'_1 * \cdots * \Sigma'_n$$

where each Π_i, Σ_i, Π'_i, and Σ'_i, is a copy of the formulas in the original entailment with their variables renamed appart.

Table 3 shows the running time spent by the provers trying to prove or refute these 209 entailments. In this case, jStar seems to fare better than Smallfoot, however recall that jStar is incomplete and fails to prove 59 of the entailments which are actually valid. Nevertheless, SLP, which is both sound and complete, outperforms the other two provers in all of the cases.

[1] With the caveat that the logic rules provided with jStar are incomplete for the fragment of separation logic under consideration (personal communication with D. Distefano) and, therefore, unable to prove the validity of 59 of these entailments.

7. Conclusions

In this paper we developed a proof system $\mathcal{SI}$ for proving the validity of separation logic entailments with list predicates. A key result from this development is a separation from the reasoning required to deal with pure equality predicates, and the reasoning required to manipulate the spatial information of such formulas. This enabled the design of an efficient, sound, and complete algorithm for this kind of entailments, which leverages techniques that have been developed in the theorem proving community from the past three to four decades. The effectiveness of our proposed algorithm is demonstrated by the implementation of a tool, SLP, which indicates speedups of orders of magnitudes with respect to the available state-of-the-art. We expect these ideas to prove fruitful in the further development and automation of program analysis and verification techniques for heap manipulating programs.

References

[1] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI*, pages 203–213, 2001.

[2] C. Barrett and C. Tinelli. CVC3. In *CAV*, pages 298–302, 2007.

[3] P. Baumgartner and U. Waldmann. Superposition and model evolution combined. In *CADE*, pages 17–34, 2009.

[4] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, number 3328 in LNCS, pages 97–109, 2004.

[5] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.

[6] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2006.

[7] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, pages 178–192, 2007.

[8] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.

[9] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. C. Rinard. Using first-order theorem provers in the Jahob data structure verification system. In *VMCAI*, pages 74–88, 2007.

[10] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4SMT solver. In *CAV*, pages 299–303, 2008.

[11] C. Calcagno, M. Parkinson, and V. Vafeiadis. SmallfootRG. In *SAS*, pages 233–238, 2007.

[12] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.

[13] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.

[14] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[15] L. M. de Moura and N. Bjørner. Tapas theory combinations and practical applications. In *FORMATS*, 2009.

[16] D. Distefano and M. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, pages 213–226, 2008.

[17] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.

[18] R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, pages 161–177, 2009.

[19] B. Dutertre and L. D. Moura. The Yices SMT solver. Technical report, Computer Science Laboratory, SRI International, 2006.

[20] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.

[21] D. Gay and A. Aiken. Memory management with explicit regions. In *PLDI*, pages 313–323, 1998.

[22] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[23] B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Katholieke Universiteit Leuven, Belgium, 2008.

[24] K. Korovin and A. Voronkov. Integrating linear arithmetic into superposition calculus. In *Computer Science Logic (CSL'07)*, volume 4646 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2007.

[25] N. Marti and R. Affeldt. A certified verifier for a fragment of separation logic. *Computer Software*, 25(3):135–147, 2008.

[26] M. Méndez-Lojo and M. V. Hermenegildo. Precise set sharing analysis for Java-style programs. In *VMCAI*, pages 172–187, 2008.

[27] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *PLDI*, pages 221–231, 2001.

[28] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, pages 229–240, 2008.

[29] H. H. Nguyen, V. Kuncak, and W.-N. Chin. Runtime checking for separation logic. In *VMCAI*, pages 203–217, 2008.

[30] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier, 2001.

[31] A. Podelski and T. Wies. Boolean heaps. In *SAS*, pages 268–283, 2005.

[32] A. Podelski and T. Wies. Counterexample-guided focus. In *POPL*, pages 249–260, 2010.

[33] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[34] M. C. Rinard. Integrated reasoning and proof choice point selection in the Jahob system — mechanisms for program survival. In *CADE*, pages 1–16, 2009.

[35] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[36] J. Villard, É. Lozes, and C. Calcagno. Tracking heaps that hop with Heap-Hop. In *TACAS*, pages 275–279, 2010.

[37] H. Yang. An example of local reasoning in bi pointer logic: the schorr-waite graph marking algorithm. In *SPACE workshop*, 2001.

[38] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008.