Satisfiability Modulo Abstraction for Separation Logic with Linked Lists

Aditya Thakur[1], Jason Breck[1], and Thomas Reps[1,2]

[1] University of Wisconsin; Madison, WI, USA {adi,jbreck,reps}@cs.wisc.edu
[2] GrammaTech, Inc.; Ithaca, NY, USA

Abstract. Separation logic is an expressive logic for reasoning about heap structures in programs. This paper presents a semi-decision procedure for deciding unsatisfiability of formulas in a fragment of separation logic that includes predicates describing points-to assertions ($x \mapsto y$), acyclic-list-segment assertions($\mathbf{ls}(x, y)$), logical-and, logical-or, separating conjunction, and septraction (the DeMorgan-dual of separating implication). The fragment that we consider allows negation at leaves, and includes formulas that lie outside other separation-logic fragments considered in the literature.

The semi-decision procedure is designed using concepts from abstract interpretation. The procedure uses an abstract domain of shape graphs to represent a set of heap structures, and computes an abstraction that over-approximates the set of satisfying models of a given formula. If the over-approximation is empty, then the formula is unsatisfiable.

We have implemented the method, and evaluated it on a set of formulas taken from the literature. The implementation is able to establish the unsatisfiability of formulas that cannot be handled by other existing approaches.

1 Introduction

Separation logic [29] is an expressive logic for reasoning about heap-allocated data structures in programs. It provides a mechanism for concisely describing program states by explicitly localizing facts that hold in separate regions of the heap. In particular, a "separating conjunction" ($\varphi_1 * \varphi_2$) asserts that the heap can be split into two disjoint regions ("heaplets") in which φ_1 and φ_2 hold, respectively [29]. A "septraction" ($\varphi_1 \mathrel{-\!\circledast} \varphi_2$) asserts that a heaplet h can be extended by a disjoint heaplet h_1 in which φ_1 holds, to create a heaplet $h_1 \cup h$ in which φ_2 holds [34]. The $-\!\circledast$ operator is sometimes called *existential magic wand*, because it is the DeMorgan-dual of the magic-wand operator "$-\!*$" (also called separating implication); i.e., $\varphi_1 \mathrel{-\!\circledast} \varphi_2$ iff $\neg(\varphi_1 \mathrel{-\!*} \neg\varphi_2)$.

The use of separation logic in manual, semi-automated, and automated verification tools is a burgeoning field [3, 12, 24, 13, 17]. Most of these incorporate some form of automated reasoning for separation logic, but only limited fragments of separation logic are typically handled.

This paper presents a semi-decision procedure for deciding the unsatisfiability of formulas in a fragment of separation logic that includes predicates describing

points-to assertions ($x \mapsto y$), acyclic-list-segment assertions ($\mathbf{ls}(x, y)$), empty-heap assertions (**emp**), and their negations; separating conjunction; septraction; logical-and; and logical-or. The fragment considered only allows negation at the leaves of a formula (§2.1), but still contains formulas that lie outside of previously considered fragments [2, 27, 26, 22, 19]. The semi-decision procedure can prove *validity* of implications of the form

$$\psi \Rightarrow (\varphi_i \wedge \bigwedge_j \psi_j \mathrel{-\!\!*} \varphi_j), \tag{1}$$

where φ_i and φ_j are formulas that contain only $\wedge$, $\vee$, and positive or negative occurrences of **emp**, points-to, or **ls** assertions; and ψ and ψ_j are arbitrary formulas in the logic fragment defined in §2.1. Consequently, we believe that ours is the first procedure that can prove the validity of formulas that contain both **ls** and the magic-wand operator $-\!\!*$. Furthermore, the semi-decision procedure is able to prove *unsatisfiability* of interesting classes of formulas that are outside of previously considered fragments, including (i) formulas that use *conjunctions of separating-conjunctions with* ***ls***, such as

$$(\mathbf{ls}(a1, a2) * \mathbf{ls}(a2, a3)) \wedge (\neg\mathbf{emp} * \neg\mathbf{emp}) \wedge (a1 \mapsto e1 * \mathbf{true}) \wedge e1 = \mathtt{nil},$$

and (ii) formulas that *contain both* ***ls*** *and septraction (*$-\!\!\circledast$*)*, such as

$$(a3 \mapsto a4 \mathrel{-\!\!\circledast} \mathbf{ls}(a1, a4)) \wedge (a3 = a4 \vee \neg\mathbf{ls}(a1, a3)).$$

The former are useful for describing overlapped data structures; the latter are useful in dealing with interference effects when using rely/guarantee reasoning to verify programs with fine-grained concurrency [34, 7].

The key insight behind our approach is that the semi-decision procedure is designed using concepts from abstract interpretation [10]. Given a formula φ, the semi-decision procedure sets up an appropriate abstract domain that is tailored for representing information about the meanings of subformulas of φ. In particular, it uses an abstract domain of shape graphs [30] to represent a set of heap structures. The proof calculus that we present computes an abstraction that over-approximates the set of satisfying models of the given formula. If the over-approximation is empty, then the formula is unsatisfiable. If the formula is satisfiable, then the procedure reports a set of abstract models.

This use of abstract domains to prove unsatisfiability places our work squarely in a recent line of research on using abstract values drawn from an abstract domain as a way to represent knowledge in implementations of decision procedures [14, 33, 32, 15, 16] (a technique we call "Satisfiability Modulo Abstraction"). Our work is the first to apply this idea to a fragment of separation logic.

The nature of our semi-decision procedure is thus much different from other decision procedures for fragments of separation logic that we are aware of. Most previous decision procedures are proof-theoretic. In some sense, our method is model-theoretic: it uses explicitly instantiated sets of 3-valued structures to represent overapproximations of the models of subformulas.

Contributions. The contributions of the paper include the following:

– We show how a canonical-abstraction domain can be used to overapproximate the set of heaps that satisfy a separation-logic formula (§2).
– We present rules for calculating the overapproximation of a separation-logic formula for a fragment of separation logic that consists of separating conjunction, septraction, logical-and, and logical-or (§4).
– The semi-decision procedure is parameterized by a shape abstraction, and can be instantiated to handle (positive or negative) literals for points-to or acyclic-list-segment assertions—and hence can prove the validity of implications of the kind shown in formula (1) (§4).

§3 illustrates the key concepts used in the semi-decision procedure. We evaluated our approach on a set of formulas taken from the literature (§5). To the best of our knowledge, the implementation is able to establish the unsatisfiability of formulas that cannot be handled by other existing approaches.

2 Separation Logic and Canonical Abstraction

In this section, we provide background on separation logic and introduce the separation-logic fragment considered in the paper. We then show how a canonical-abstraction domain can be used to approximate the set of models that satisfy a separation-logic formula.

2.1 Syntax and Semantics of Separation Logic

Formulas in our fragment of separation logic (SL) are defined as follows:

$$\begin{aligned} \varphi &::= \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi * \varphi \mid \varphi \mathrel{-\!\!\circledast} \varphi \mid \mathit{atom} \mid \neg \mathit{atom} \\ \mathit{atom} &::= \mathbf{true} \mid \mathbf{emp} \mid x = y \mid x \mapsto y \mid \mathbf{ls}(x, y) \end{aligned}$$

The set of literals, denoted by *lits*, is the union of the positive and negative atoms of SL.

The semantics of SL is defined with respect to memory "statelets", which consist of a *store* and a *heaplet*. A store is a function from variables to values; a heaplet is a finite function from locations to locations. Let *Loc* and *Var* be disjoint countably infinite sets, neither of which contain nil.

$$\begin{aligned} \mathit{Val} &\stackrel{\text{def}}{=} \mathit{Loc} \uplus \{\mathtt{nil}\} & \mathit{Store} &\stackrel{\text{def}}{=} \mathit{Var} \to \mathit{Val} \\ \mathit{Heaplet} &\stackrel{\text{def}}{=} \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{Val} & \mathit{Statelet} &\stackrel{\text{def}}{=} \mathit{Store} \times \mathit{Heaplet} \end{aligned}$$

Loc represents heap-node addresses. The domain of h, $\mathrm{dom}(h)$, represents the set of addresses of cells in the heaplet. Two heaplets h_1, h_2 are *disjoint*, denoted by $h_1 \# h_2$, if $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$. Given two disjoint heaplets h_1 and h_2, $h_1 \cdot h_2$ denotes their disjoint union $h_1 \uplus h_2$. A *statelet* is denoted by a pair (s, h).

Satisfaction of an SL formula φ with respect to statelet (s, h) is defined in Fig. 1. $[\![\varphi]\!]$ denotes the set of statelets that satisfy φ: $[\![\varphi]\!] \stackrel{\text{def}}{=} \{(s, h) \mid (s, h) \models \varphi\}$.

2.2 2-Valued Logical Structures

We model full states—not statelets—by 2-*valued logical structures*. A logical structure provides an interpretation of a vocabulary $\mathrm{Voc} = \{eq, p_1, \ldots, p_n\}$ of predicate symbols (with given arities). Voc_k denotes the set of k-ary symbols.

$(s,h) \models \varphi_1 \wedge \varphi_2$ iff $(s,h) \models \varphi_1$ and $(s,h) \models \varphi_2$
$(s,h) \models \varphi_1 \vee \varphi_2$ iff $(s,h) \models \varphi_1$ or $(s,h) \models \varphi_2$
$(s,h) \models \varphi_1 * \varphi_2$ iff $\exists h_1, h_2.\ h_1 \# h_2$ and $h_1 \cdot h_2 = h$ and $(s,h_1) \models \varphi_1$ and $(s,h_2) \models \varphi_2$
$(s,h) \models \varphi_1 \mathrel{-\!\!\circledast} \varphi_2$ iff $\exists h_1.\ h_1 \# h$ and $(s,h_1) \models \varphi_1$ and $(s,h_1 \cdot h) \models \varphi_2$
$(s,h) \models \neg atom$ iff $(s,h) \not\models atom$
$(s,h) \models \mathbf{true}$ iff $\mathbf{true}$
$(s,h) \models \mathbf{emp}$ iff $\mathrm{dom}(h) = \emptyset$
$(s,h) \models x = y$ iff $s(x) = s(y)$
$(s,h) \models x \mapsto y$ iff $\mathrm{dom}(h) = \{s(x)\}$ and $h(s(x)) = s(y)$
$(s,h) \models \mathbf{ls}(x,y)$ iff if $s(x) = s(y)$ then $\mathrm{dom}(h) = \emptyset$,
else there is a nonempty acyclic path from $s(x)$ to $s(y)$ in h,
and this path contains all heap cells in h

Fig. 1: Satisfaction of an SL formula φ with respect to statelet (s,h).

Predicate	**Intended Meaning**
$eq(v_1, v_2)$	Do v_1 and v_2 denote the same memory cell?
$q(v)$	Does pointer variable q point to memory cell v?
$n(v_1, v_2)$	Does the n-field of v_1 point to v_2?

Fig. 2: Core predicates used when representing states made up of acyclic linked lists.

Definition 1. *A* 2-***valued logical structure*** *S over Voc is a pair $S = \langle U, \iota \rangle$, where U is the set of* ***individuals****, and ι is the* ***interpretation****. Let $\mathbb{B} = \{0,1\}$ be the domain of truth values. For $p \in Voc_i$, $\iota(p)\colon U^i \to \mathbb{B}$. We assume that $eq \in Voc_2$ is the identity relation: (i) for all $u \in U$, $\iota(eq)(u,u) = 1$, and (ii) for all $u_1, u_2 \in U$ such that u_1 and u_2 are distinct individuals, $\iota(eq)(u_1,u_2) = 0$.*

The set of 2*-valued logical structures over Voc is denoted by 2-STRUCT[Voc].*

A concrete state is modeled by a 2-valued logical structure over a fixed vocabulary $\mathcal{C}$ of *core predicates*. Core predicates are part of the underlying semantics of the linked structures that make up the states of interest. Fig. 2 lists the core predicates that are used when representing states made up of acyclic linked lists.

Without loss of generality, vocabularies exclude constant and function symbols. Constant symbols can be encoded via unary predicates, and n-ary functions via $n+1$-ary predicates. In both cases, we need *integrity rules*—i.e., global constraints that restrict the set of structures considered to the ones that we intend. For instance, we use a subset of the unary predicates, denoted by $PVar \subseteq \mathrm{Voc}_1$, to encode pointer variables (i.e., $x \in PVar$ encodes program variable x), and binary predicate $n \in \mathrm{Voc}_2$ to encode list-node linkages. In essence, the following integrity rules restrict each $x \in PVar$ to serve as a (possibly undefined) constant, and restrict relation n to encode a partial function:

$$\text{for each } x \in PVar, \forall v_1, v_2 : x(v_1) \wedge x(v_2) \Rightarrow eq(v_1, v_2)$$
$$\forall v_1, v_2, v_3 : n(v_3, v_1) \wedge n(v_3, v_2) \Rightarrow eq(v_1, v_2)$$

2.3 Connecting 2-Valued Logical Structures and SL Statelets

We use unary *domain predicates*, typically denoted by d, d', $d_1, \ldots, d_k \in \mathrm{Voc}_1$, to pick out regions of the heap that are of interest in the state that a logical structure models. The connection between 2-valued logical structures and SL statelets is formalized by means of the operation $S|_{(d,\cdot)}$, which performs a projection of structure S with respect to a domain predicate d:

$$S|_{(d,\cdot)} \stackrel{\text{def}}{=} (s, h), \text{ where } s = \begin{pmatrix} \{(p,u) \mid p \in PVar^S, u \in U^S, \text{and } p(u)\} \\ \cup \{(q, \mathtt{nil}) \mid q \in PVar^S \text{ and } \neg\exists v : q(v)\} \end{pmatrix} \quad (2)$$

$$\text{and } h = \{(u_1, u_2) \mid u_1, u_2 \in U^S, d(u_1), \text{and } n(u_1, u_2)\}. \quad (3)$$

The subscript "$(d,\cdot)$" serves as a reminder that in Eqn. (3), only u_1 needs to be in the region defined by d. We lift the projection operation to apply to a set SS of 2-valued logical structures as follows: $\mathrm{SS}|_{(d,\cdot)} \stackrel{\text{def}}{=} \{S|_{(d,\cdot)} \mid S \in \mathrm{SS}\}$.

2.4 Representing Sets of SL Statelets using Canonical Abstraction

In the framework of Sagiv et al. [30] for logic-based abstract-interpretation, 3-*valued logical structures* provide a way to overapproximate possibly infinite sets of 2-valued structures in a finite way that can be represented in a computer. The application of Eqns. (2) and (3) to 3-valued structures means that the abstract-interpretation machinery developed by Sagiv et al. provides a finite way to overapproximate a possibly infinite set of SL statelets.

In 3-valued logic, a third truth value, denoted by 1/2, represents uncertainty. The set $\mathbb{T} \stackrel{\text{def}}{=} \mathbb{B} \cup \{1/2\}$ of 3-valued truth values is partially ordered "$l \sqsubset 1/2$ for $l \in \mathbb{B}$". The values 0 and 1 are *definite* values; 1/2 is an *indefinite* value.

Definition 2. *A* 3***-valued logical structure*** $S = \langle U, \iota \rangle$ *is almost identical to a* 2*-valued structure, except that* ι *maps each* $p \in Voc_i$ *to a* 3*-valued function* $\iota(p) \colon U^i \to \mathbb{T}$*. In addition, (i) for all* $u \in U$*,* $\iota(eq)(u,u) \sqsupseteq 1$*, and (ii) for all* $u_1, u_2 \in U$ *such that* u_1 *and* u_2 *are distinct individuals,* $\iota(eq)(u_1,u_2) = 0$*. (An individual* u *for which* $\iota(eq)(u,u) = 1/2$ *is called a* ***summary individual****.)*

The set of 3*-valued logical structures over Voc is denoted by* 3*-STRUCT*$[Voc]$*. Note that* 2*-STRUCT*$[Voc] \subsetneq$ 3*-STRUCT*$[Voc]$*.*

As we will see below, a summary individual may represent more than one individual from certain 2-valued structures.

A 3-valued structure can be depicted as a directed graph with individuals as graph nodes (see Fig. 3). A summary individual is depicted with a double-ruled border. A unary predicate $p \in PVar$ is represented in the graph by having an arrow from the predicate name p to all nodes of individuals u for which $\iota(p)(u) \sqsupseteq 1$. An arrow between two nodes indicates that a binary predicate holds for the corresponding pair of individuals. A predicate value of 1/2 is indicated by a dotted arrow, a value of 1 by a solid arrow, and a value of 0 by the absence of an arrow. A unary predicate $p \in (\mathrm{Voc}_1 - PVar)$ is listed, with its value, inside the

node of each individual u for which $\iota(p)(u) \sqsupseteq 1$. A nullary predicate is displayed in a rectangular box.

To define a suitable abstraction of 2-valued logical structures, we start with the notion of structure embedding [30]:

Definition 3. *Given $S = \langle U, \iota \rangle$ and $S' = \langle U', \iota' \rangle$, two 3-valued structures over the same vocabulary Voc, and $f : U \to U'$, a surjective function, f* ***embeds*** *S* ***in*** *S', denoted by $S \sqsubseteq^f S'$, if for all $p \in Voc$ and $u_1, \ldots, u_k \in U$,*

$$\iota(p)(u_1, \ldots, u_k) \sqsubseteq \iota'(p)(f(u_1), \ldots, f(u_k))$$

If, in addition,

$$\iota'(p)(u'_1, \ldots, u'_k) = \bigsqcup_{u_1,\ldots,u_k \in U, s.t. f(u_i)=u'_i, 1 \le i \le k} \iota(p)(u_1, \ldots, u_k)$$

then S' is the ***tight embedding of*** *S* ***with respect to*** *f, denoted by $S' = f(S)$. (Note that we overload f to also mean the mapping on structures $f : 3\text{-}STRUCT[Voc] \to 3\text{-}STRUCT[Voc]$ induced by $f : U \to U'$.)*

Intuitively, $f(S)$ is obtained by merging individuals of S and by defining the valuation of predicates accordingly (in the most precise way). The relation $\sqsubseteq^{\mathrm{id}}$, which will be denoted by $\sqsubseteq$, is the natural information order between structures that share the same universe. One has $S \sqsubseteq^f S' \Leftrightarrow f(S) \sqsubseteq^{\mathrm{id}} S'$.

However, embedding alone is not enough. The challenge for representing and manipulating sets of 2-valued structures is that the universe of a structure is of *a priori* unbounded size. Consequently, we need a method that, for a 2-valued structure $\langle U, \iota \rangle \in$ 2-STRUCT[Voc], abstracts U to an abstract universe $U^\sharp$ of bounded size. Intuitively, the solution involves (i) identifying an *a priori* bounded number of abstract individuals $U^\sharp$, (ii) defining a surjective function $f : U \to U^\sharp$, and (ii) performing a tight embedding of S with respect to f. Given $U^\sharp$ and f, we can define the following Galois connection:

$$\wp(\text{2-STRUCT[Voc]}) \underset{\alpha_f}{\overset{\gamma_f}{\leftrightarrows}} \text{3-STRUCT[Voc]}$$
$$\alpha_f(X) = \bigsqcup_{S \in X} f(S) \qquad \gamma_f(S^\sharp) = \{S \mid S \sqsubseteq^f S^\sharp\}$$

In this abstraction, sets of valuations for predicate symbols $\iota : \text{Voc} \to \left(\bigcup_k U^k \to \mathbb{B}\right)$ are abstracted with a single abstract valuation $\iota : \text{Voc} \to \left(\bigcup_k (U^\sharp)^k \to \mathbb{T}\right)$.

The idea behind *canonical abstraction* [30, §4.3] is to choose a subset $\mathbb{A} \subseteq \text{Voc}_1$ of *abstraction predicates*, and to define an equivalence relation $\simeq_{\mathbb{A}^S}$ on U that is parameterized by the logical structure $S = \langle U, \iota \rangle \in$ 2-STRUCT[Voc] to be abstracted:

$$u_1 \simeq_{\mathbb{A}^S} u_2 \;\Leftrightarrow\; \forall p \in \mathbb{A} : \iota(p)(u_1) = \iota(p)(u_2).$$

This equivalence relation defines the surjective function $f^S_{\mathbb{A}} : U \to (U/\simeq_{\mathbb{A}^S})$, which maps an individual to its equivalence class. We thus have the Galois connection

$$\wp(\text{2-STRUCT[Voc]}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \wp(\text{3-STRUCT[Voc]})$$
$$\alpha(X) = \{f^S_{\mathbb{A}}(S) \mid S \in X\} \qquad \gamma(Y) = \{S \mid S^\sharp \in Y \wedge S \sqsubseteq^f S^\sharp\},$$

where $f^S_{\mathbb{A}}$ in the definition of α denotes the tight-embedding function for logical structures induced by the node-embedding function $f^S_{\mathbb{A}} \colon U \to (U/\simeq_{\mathbb{A}^S})$. The abstraction function α is referred to as *canonical abstraction*. Note that there is an upper bound on the size of each structure $\langle U^\sharp, \iota^\sharp \rangle \in$ 3-STRUCT[Voc] that is in the image of α: $|U^\sharp| \leq 2^{|\mathbb{A}|}$—and thus the power-set of the image of α is a finite sublattice of $\wp$(3-STRUCT[Voc]).

For technical reasons, it turns out to be convenient to work with 3-valued structures other than the ones in the image of α; however, we still want to restrict ourselves to a finite sublattice of $\wp$(3-STRUCT[Voc]). With this motivation, we make the following definition:

Definition 4. *A 3-valued structure $\langle U^\sharp, \iota^\sharp \rangle \in$ 3-STRUCT[Voc] is* ***bounded*** *(with respect to abstraction predicates $\mathbb{A}$) if for every $u_1, u_2 \in U^\sharp$, where $u_1 \neq u_2$, there exists an abstraction predicate symbol $p \in \mathbb{A} \subseteq Voc_1$ such that $\iota^\sharp(p)(u_1) = 0$ and $\iota^\sharp(p)(u_2) = 1$, or $\iota^\sharp(p)(u_1) = 1$ and $\iota^\sharp(p)(u_2) = 0$. B-STRUCT[Voc, $\mathbb{A}$] denotes the set of such structures.*

Defn. 4 also imposes an upper bound on the size of a structure $\langle U^\sharp, \iota^\sharp \rangle \in$ B-STRUCT[Voc, $\mathbb{A}$]—again, $|U^\sharp| \leq 2^{|\mathbb{A}|}$—and thus $\wp$(B-STRUCT[Voc, $\mathbb{A}$]) is a finite sublattice of $\wp$(3-STRUCT[Voc]). It defines the abstract domain that we use, the *abstract domain whose elements are subsets of B-STRUCT[Voc, $\mathbb{A}$]*, which will be denoted by $\mathcal{A}$[Voc, $\mathbb{A}$]. (For brevity, we call such a domain a "*canonical-abstraction domain*", and denote it by $\mathcal{A}$ when Voc and $\mathbb{A}$ are understood.) The Galois connection we work with is thus

$$\wp(\text{2-STRUCT[Voc]}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \wp(\text{B-STRUCT[Voc, }\mathbb{A}\text{]}) = \mathcal{A}[\text{Voc}, \mathbb{A}]$$
$$\alpha(X) = \{f^S_{\mathbb{A}}(S) \mid S \in X\} \qquad \gamma(Y) = \{S \mid S^\sharp \in Y \wedge S \sqsubseteq^f S^\sharp\}.$$

3 Overview

In this section, we illustrate the concepts that we use in the semi-decision procedure using a formula that is unsatisfiable over acyclic linked lists: $x \mapsto y * y \mapsto x$. App. A illustrates the procedure using a formula that is satisfiable over acyclic linked lists: $x \mapsto y \mathbin{-\!\circledast} \mathbf{ls}(x, z)$.

Consider $\varphi \stackrel{\text{def}}{=} x \mapsto y * y \mapsto x$. We want to compute $A \in \mathcal{A}$ such that $\gamma(A)|_{(d,\cdot)} \supseteq [\![\varphi]\!]$. The key to handling the $*$ operator is to introduce two new domain predicates d_1 and d_2, which are used to demarcate the heaplets that must satisfy $\varphi_1 \stackrel{\text{def}}{=} x \mapsto y$ and $\varphi_2 \stackrel{\text{def}}{=} y \mapsto x$, respectively. We have designed $\mathcal{A}$ so that there exist $A_1, A_2 \in \mathcal{A}$ such that $\gamma(A_1)|_{(d_1,\cdot)} = [\![x \mapsto y]\!]$ and $\gamma(A_2)|_{(d_2,\cdot)} = [\![y \mapsto x]\!]$, respectively. Tab. 1 describes the abstraction predicates we use. A_1 and A_2 each consist of a single 3-valued structure, shown in Fig. 3(b) and Fig. 3(c), respectively. Furthermore, to satisfy $\varphi_1 * \varphi_2$, d_1 and d_2 are required to be disjoint regions whose union is d. $\mathcal{A}$ also contains an abstract value, which we will call D, that represents this disjointness constraint exactly. D consists of four 3-valued structures. Fig. 3(a) shows the "most general" of them: it represents two disjoint regions, d_1 and d_2, that partition the d region (where each of d_1 and d_2 contain

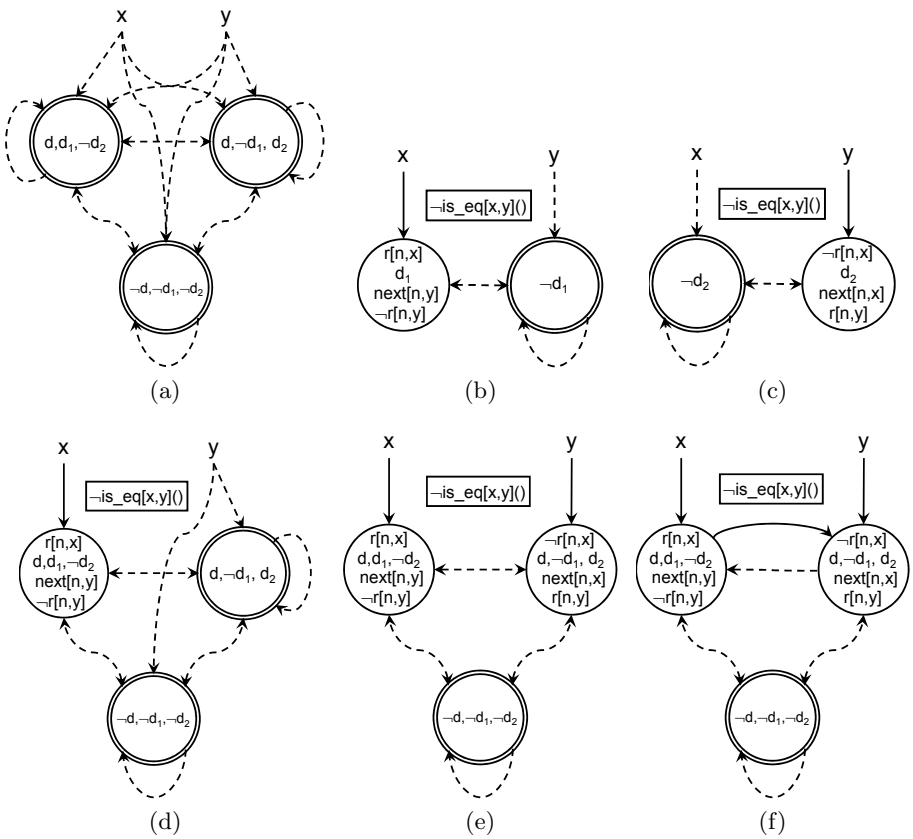

Fig. 3: Structures that arise in the meet operation used to analyze $x \mapsto y * y \mapsto x$.

at least one cell). The summary individual labeled $\neg d, \neg d_1, \neg d_2$ in Fig. 3(a) represents a region that is disjoint from d. (See also Fig. 6.)

Note that here and throughout the paper, for brevity the figures only show predicates that are relevant to the issue under discussion.

Meet for a Canonical-Abstraction Domain. To impose a necessary condition for $x \mapsto y * y \mapsto x$ to be satisfiable, we take the *meet* of D, A_1, and A_2: $[\![x \mapsto y * y \mapsto x]\!] \subseteq D \sqcap A_1 \sqcap A_2$. Figs. 3(d), (e), and (f) show some of the structures that arise in $D \sqcap A_1 \sqcap A_2$.

The meet operation in $\mathcal{A}$ is defined in terms of the greatest-lower-bound operation induced by the approximation order in the lattice B-STRUCT[Voc, $\mathbb{A}$]. Arnold et al. [1] show that in general this operation is NP-complete; however, they define an algorithm based on graph matching that typically performs well in practice [20, §8.3]. To understand some of the subtleties of meet, consider Fig. 3(d), which shows one of the structures in $D \sqcap A_1$ (i.e., Fig. 3(a)$\sqcap$Fig. 3(b)).

– From the standpoint of Fig. 3(b), meet caused the summary individual labeled "$\neg d_1$" to be split into two summary individuals: "$\neg d, \neg d_1, \neg d_2$" and "$d, \neg d_1, d_2$".

– From the standpoint of Fig. 3(a), meet caused the summary individual labeled "$d, d_1, \neg d_2$" to (i) become a non-summary individual, (ii) acquire the value 1 for x, $r[n, x]$, and $next[n, y]$, and (iii) acquire the value 0 for y and $r[n, y]$.

Fig. 3(e) shows one of the structures in $(D \sqcap A_1) \sqcap A_2$, i.e., Fig. 3(d) $\sqcap$ Fig. 3(c), which causes further (formerly indefinite) elements to acquire definite values.

Arnold et al. develop a graph-theoretic notion of the possible correspondences among individuals in the bounded structures that are arguments to meet, and structure the meet algorithm around the set of possible correspondences [1, §4.2].

Improving Precision Using Semantic-Reduction Operators. Fig. 3(e) still contains a great deal of indefinite information because the meet operation does not take into account the integrity constraints on structures. For instance, for the structures that we use to represent states and `SL` statelets, we use a unary predicate $next[n, y]$, which holds for individuals whose n-link points to the individual that is pointed to by y. This predicate has an associated integrity constraint

$$\forall v_1, v_2 . next[n, y](v_1) \wedge y(v_2) \Rightarrow n(v_1, v_2). \tag{4}$$

In particular, in Fig. 3(e) the individual pointed to by x has $next[n, y] = 1$; however, the edge to the individual pointed to by y has the value 1/2.

To improve the precision of the (graph-theoretic) meet, the semi-decision procedure makes use of *semantic-reduction operators*. The notion of semantic reduction was introduced by Cousot and Cousot [11]. Semantic-reduction operators are useful when an abstract domain is a lattice that has multiple elements that represent the same set of states. A semantic reduction operator ρ maps an abstract-domain element A to $\rho(A)$ such that (i) $\rho(A) \sqsubseteq A$, and (ii) $\gamma(\rho(A)) = \gamma(A)$. In other words, ρ maps A to an element that is lower in the lattice—and hence a "better" representation of $\gamma(A)$ in $\mathcal{A}$—while preserving the meaning. In our case, the semantic-reduction operations that we use convert a set of 3-valued structures XS into a "better" set of 3-valued structures XS' that describe the same set of 2-valued structures.

A semantic-reduction operator can have two effects:

1. In some structure $S \in XS$, some tuple $p(u)$ with indefinite value 1/2 may be changed to have a definite value (0 or 1).
2. It may be determined that some structure $S \in XS$ is infeasible: i.e., $\gamma(S) = \emptyset$. In this case, S is removed from XS.

The effect of a precision improvement from a type-1 effect can cause a type-2 effect to occur. For instance, let u_1 and u_2 be the individuals pointed to by x and y, respectively, in Fig. 3(e).

– Fig. 3(f) is Fig. 3(e) after integrity constraint (4) has triggered a type-1 change that improves the value of $n(u_1, u_2)$ from 1/2 to 1.
– A type-2 rule can then determine that the structure shown in Fig. 3(f) is infeasible. In particular, the predicate $r[n, x](v)$ means that individual v is reachable from the individual pointed to by x along n-links. The semantic-reduction rule would find that the values $x(u_1) = 1$, $n(u_1, u_2) = 1$, and $r[n, x](u_2) = 0$ represent an irreconcilable inconsistency in Fig. 3(f): the first

$$\frac{}{\ell \in \mathit{lits}, d \Vdash A_\ell}\ (\ell) \qquad \frac{\varphi_1, d \Vdash A_1 \qquad \varphi_2, d \Vdash A_2}{\varphi_1 \wedge \varphi_2, d \Vdash A_1 \sqcap A_2}\ (\wedge)$$

$$\frac{\varphi_1, d \Vdash S_1 \qquad \varphi_2, d \Vdash A_2}{\varphi_1 \vee \varphi_2, d \Vdash A_1 \sqcup A_2}\ (\vee) \qquad \frac{\varphi_1, d_1 \Vdash A_1 \qquad \varphi_2, d_2 \Vdash A_2}{\varphi_1 * \varphi_2, d \Vdash ([d = d_1 \cdot d_2]^\sharp \sqcap A_1 \sqcap A_2) \downarrow^d}\ (*)$$

$$\frac{\varphi_1, d_1 \Vdash A_1 \qquad \varphi_2, d_2 \Vdash A_2}{\varphi_1 \mathrel{-\!\circledast} \varphi_2, d \Vdash ([d_2 = d \cdot d_1]^\sharp \sqcap A_1 \sqcap A_2) \downarrow^d}\ (-\!\circledast)$$

Fig. 4: Rules for computing an abstract value that overapproximates the meaning of a formula in SL.

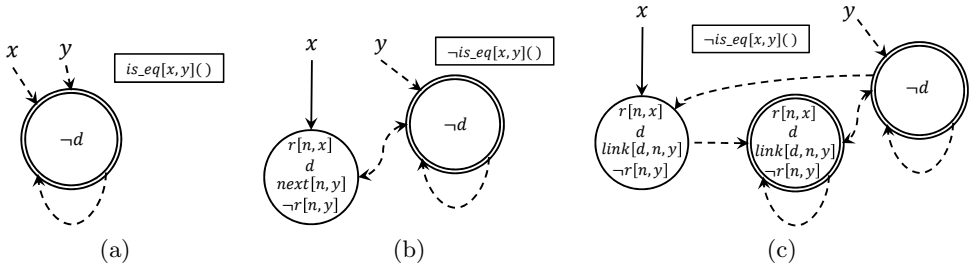

Fig. 5: The abstract value for $\mathbf{ls}(x, y) \in \mathit{atom}$ in the canonical-abstraction domain.

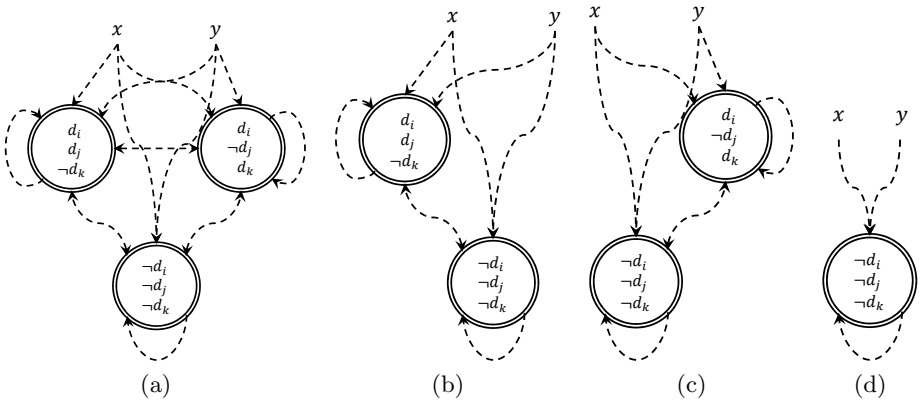

Fig. 6: The abstract value for $[d_i = d_j \cdot d_k]^\sharp$ in the canonical-abstraction domain.

two predicate values mean that u_2 is reachable from the individual pointed to by x along n-links, which contradicts $r[n, x](u_2) = 0$.

The operation that applies type-1 and type-2 rules until no more changes are possible is called *coerce* (because it coerces XS to a better representation XS'). Sagiv et al. [30, §6.4] and Bogudlov et al. [5, 4] discuss algorithms for *coerce*.

Predicate	Intended Meaning
$is\_eq[x, y]()$	Are x and y equal?
$next[n, y](v)$	The target of the n-edge from v is pointed to by y
$t[n](v_1, v_2)$	Is v_2 reachable via zero or more n-edges from v_1?
$r[n, y](v)$	$\exists v_1.y(v_1) \wedge t[n](v_1, v)$
$d(v)$	Is v in heap domain d?
$link[d, n, y](v)$	The target of the n-edge from v is either in d or is pointed to by y

Table 1: Voc consists of the predicates shown above, together with the ones in Fig. 2. All unary predicates are abstraction predicates; that is, $\mathbb{A} = \mathrm{Voc}_1$.

4 Proof System for Separation Logic

This section describes how we compute $A \in \mathcal{A}[\mathrm{Voc}, \mathbb{A}]$ such that A overapproximates the satisfying models of $\varphi \in \mathtt{SL}$. The vocabulary Voc and abstraction predicates $\mathbb{A}$ are listed in Tab. 1.

The semi-decision procedure works with judgments of the form "$\varphi, d \Vdash A$", where d is a domain predicate. The invariant maintained by the semi-decision procedure is that, whenever it establishes a judgment $\varphi, d \Vdash A$, $A \in \mathcal{A}$ overapproximates φ in the following sense: $\gamma(A)|_{(d,\cdot)} \supseteq [\![\varphi]\!]$. Fig. 4 lists the rules used for calculating $\varphi, d \Vdash A$ for $\varphi \in \mathtt{SL}$.

For each literal $\ell \in \mathit{lits}$, there is an abstract value $A_\ell \in \mathcal{A}$ such that $\gamma(A_\ell)|_{(d,\cdot)} = [\![\ell]\!]$. These A_ℓ values are used in the (ℓ)-rule of Fig. 4. Fig. 5 shows the abstract value $A_{\mathbf{ls}}$ used for $\mathbf{ls}(x, y)$. $A_{\mathbf{ls}}$ consists of three structures:

- Fig. 5(a) represents the empty list from x to y. That is, $x = y$ and region d is empty.
- Fig. 5(b) represents a singleton list from x to y. That is, $x \neq y$ and $x \neq \mathtt{nil}$, and for all individuals v in d, v is reachable from x and $link[d, n, y](v)$ is true. (See line 6 of Tab. 1.)
- Fig. 5(c) represents acyclic linked lists of length two or more from x to y.

Fig. 5(b) is the single structure in $A_{x \mapsto y}$. The abstract values for atoms $x = y$, **true**, and **emp** are straightforward. We see that it is possible to represent the positive literals **true**, **emp**, $x = y$, $x \mapsto y$, and $\mathbf{ls}(x, y)$ precisely in $\mathcal{A}$; that is, we have $\gamma A_l|_{(d,\cdot)} = [\![l]\!]$. Furthermore, because the canonical-abstraction domain $\mathcal{A}$ is closed under negation [21, 36], we are able to represent the negative literals $x \neq y$, $\neg$**true**, $\neg$**emp**, $\neg\mathbf{ls}(x, y)$, and $\neg x \mapsto y$ precisely in $\mathcal{A}$, as well.

The rest of the rules in Fig. 4 can be derived by reinterpreting the concrete logical operators using an appropriate abstract operator. In particular, logical-and is reinterpreted as meet, and logical-or is reinterpreted as join. Consequently, the $(\wedge)$-rule and $(\vee)$-rule are straightforward. The $(\wedge)$-rule and $(\vee)$-rule are justified by the following observation: if $\gamma(A_1)|_{(d,\cdot)} \supseteq [\![\varphi_1]\!]$ and $\gamma(A_2)|_{(d,\cdot)} \supseteq [\![\varphi_2]\!]$, then $\gamma(A_1 \sqcap A_2)|_{(d,\cdot)} \supseteq [\![\varphi_1 \wedge \varphi_2]\!]$ and $\gamma(A_1 \sqcup A_2)|_{(d,\cdot)} \supseteq [\![\varphi_1 \vee \varphi_2]\!]$.

For a given structure $A = \langle U, \iota \rangle$ and unary domain predicate d_i, we use the phrase "*individuals in* d_i" to mean the set of individuals $\{u \in U \mid \iota(d_i)(u) = 1\}$.

The $(*)$-rule computes $A \in \mathcal{A}$ such that $\gamma(A)|_{(d,\cdot)} \supseteq [\![\varphi_1 * \varphi_2]\!]$. The handling of separating conjunction $\varphi_1 * \varphi_2$ is based on the following insights:

- The domain predicates d_1 and d_2 are used to capture the heaplets h_1 and h_2 that satisfy φ_1 and φ_2, respectively. That is,

$$\gamma(A_1)|_{(d_1,\cdot)} \supseteq [\![\varphi_1]\!] \text{ and } \gamma(A_2)|_{(d_2,\cdot)} \supseteq [\![\varphi_2]\!]. \tag{5}$$

- $[d = d_1 \cdot d_2]^\sharp \in \mathcal{A}$ is used to express the constraint that the individuals in d_1 are disjoint from d_2, and that the individuals in d are the disjoint union of the individuals in d_1 and d_2. With only a slight abuse of notation, the meaning of $[d = d_1 \cdot d_2]^\sharp$ can be expressed as follows:

$$\gamma([d = d_1 \cdot d_2]^\sharp)|_{(d,\cdot)} \supseteq \{(s, h, h_1, h_2) \mid h_1 \# h_2 \text{ and } h_1 \cdot h_2 = h\}. \tag{6}$$

  Fig. 6 shows the four structures in the abstract value $[d_i = d_j \cdot d_k]^\sharp$, where d_i, d_j, and d_k are domain predicates.
- $(\cdot)\lightning^d$ denotes the structure that results from setting the abstraction predicates to 1/2 for all individuals not in d, and setting all domain predicates other than d to 1/2. In effect, this operation blurs the distinction between individuals in d_1 and d_2, and serves as an abstract method for quantifier elimination.

Using Eqns. (5) and (6) in the definition of $\varphi_1 * \varphi_2$, we have

$$\begin{aligned}
&[\![\varphi_1 * \varphi_2]\!] \\
&= \{(s,h) \mid \exists h_1, h_2.\ h_1 \# h_2 \text{ and } h_1 \cdot h_2 = h \text{ and } (s,h_1) \models \varphi_1 \text{ and } (s,h_2) \models \varphi_2\} \\
&\subseteq \qquad ([d = d_1 \cdot d_2]^\sharp \quad \sqcap \quad A_1 \quad \sqcap \quad A_2)\lightning^d
\end{aligned}$$

The handling of septraction in the $(-\circledast)$-rule is similar to the handling of separating conjunction in the $(*)$-rule, except for the condition that $h_2 = h \cdot h_1$. This requirement is easily handled by using $[d_2 = d \cdot d_1]^\sharp$. App. A illustrates the application of the $(-\circledast)$-rule.

Theorem 1. *The rules in Fig. 4 are sound; that is, if the rules in Fig. 4 say that $\varphi, d \Vdash A$, then $\gamma(A)|_{(d,\cdot)} \supseteq [\![\varphi]\!]$.* □

The proof follows from the fact that each of the abstract operators is sound.

5 Experimental Evaluation

This section presents the results of our experiments to evaluate the costs and benefits of our approach. The experiments were designed to shed light on the following questions:

1. How costly is the semi-decision procedure (in terms of time)?
2. How often is the semi-decision procedure able to determine that a formula is unsatisfiable?

Formula Characteristics													
Formula	**emp**		$x = y$		$x \mapsto y$		$\mathbf{ls}(x, y)$		$\varphi \wedge \varphi$	$\varphi \vee \varphi$	$\varphi * \varphi$	$\varphi \mathrel{-\!\!\circledast} \varphi$	Full
Group	+	−	+	−	+	−	+	−					Corpus
Group 1	1	5	8	8	13	1	19	10	22	4	12	10	23
Group 2	64	22	0	0	22	22	22	22	64	0	64	0	64
Group 3	512	218	0	0	218	218	218	218	512	0	512	512	512
Total	577	245	8	8	253	241	259	250	598	4	588	522	599

Table 2: Number of formulas that contain each of the SL operators in the three groups of formulas used in the experiments. The columns labeled "+" and "−" indicate the number of atoms occurring as positive and negative literals, respectively.

3. For unsatisfiable formulas that are beyond the capabilities of other existing tools, is the semi-decision procedure actually able to determine that the formulas are unsatisfiable?

Setup. The semi-decision procedure is written in OCaml; it compiles a formula to a proof DAG, expressed as an equation system. The abstract-value manipulations in the proof rules of Fig. 4 are performed using ITVLA, a modified version of TVLA [23] that was implemented for performing interprocedural shape analysis [20, §8]. ITVLA (i) replaces TVLA's notion of an intraprocedural control-flow graph by the more general notion of *equation system*, in which transfer functions may depend on more than one argument, and (ii) supports a more general language in which to specify equation systems. In particular, the ITVLA language supports explicit use of the meet operator [1] for a canonical-abstraction domain. Experiments were run on a single core of a 2-processor, 4-core-per-processor 2.27 GHz Xeon computer running Red Hat Linux 6.5.

Test Suite. Our test suite consists of three groups of unsatisfiable formulas:

- Group 1, shown in Tab. 3, was chosen to evaluate our procedure on a wide spectrum of formulas.
- Group 2 was created by replacing the Boolean variables a and b in the template $T_1 \stackrel{\text{def}}{=} \neg a \wedge \mathbf{emp} \wedge (a * b)$ with the 8 literals *lits* of SL; that is, **true**, **emp**, $x \mapsto y$, $\mathbf{ls}(x, y)$, and their negations. Five of the 64 instantiations of template T_1 are shown in Tab. 4.
- Group 3 was created by replacing the Boolean variables a, b, and c in the template $T_2 \stackrel{\text{def}}{=} \mathbf{emp} \wedge a \wedge (b * (c \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \neg a)))$ with the 8 literals *lits* of SL. Five of the 512 instantiations of template T_2 are shown in Tab. 5.

Templates T_1 and T_2 are based on work by Hou et al. [19] on Boolean separation logic. Templates T_1 and T_2 are listed as formulas 15 and 19, respectively, in [19, Tab. 2]. In total, there were 599 formulas in our test suite. Tab. 2 summarizes the characteristics of the corpus based on the occurrences of the SL operators.

Though not shown in this section, we also evaluated our procedure on a set of satisfiable formulas. The procedure reports a set of abstract models when given a satisfiable formula. The time taken to compute these abstract models is similar to that for proving formulas unsatisfiable.

	Formula	Unsat	Time
(1)	$a1 \mapsto a2 \wedge \neg\mathbf{ls}(a1, a2)$	✓	1.74
(2)	$a1 \mapsto a2 * a2 \mapsto a1$	✓	1.70
(3)	$\neg\mathbf{emp} \wedge (\mathbf{ls}(a1, a2) * \mathbf{ls}(a2, a1))$	✓	2.00
(4)	$a1 \neq a2 \wedge (\mathbf{ls}(a1, a2) * \mathbf{ls}(a2, a1))$	✓	1.97
(5)	$(\mathbf{ls}(a1, a2) * \mathbf{ls}(a2, a3)) \wedge \neg\mathbf{ls}(a1, a3)$	✓	4.46
(6)	$\mathbf{ls}(a1, a2) \wedge \mathbf{emp} \wedge a1 \neq a2$	✓	1.34
(7)	$(a1 \mapsto a2 * \mathbf{true}) \wedge (a2 \mapsto a3 * \mathbf{true}) \wedge (\mathbf{true} * a3 \mapsto a1)$	✓	4.25
(8)	$(a1 \mapsto a2 \mathrel{-\!\!\circledast} \mathbf{true}) \wedge (a1 \mapsto a2 * \mathbf{true})$	✓	2.43
(9)	$(\mathbf{ls}(a1, a2) * \neg\mathbf{ls}(a2, a3)) \wedge \mathbf{ls}(a1, a3)$	✓	22.2
(10)	$\mathbf{ls}(a1, a2) \wedge \mathbf{ls}(a1, a3) \wedge \neg\mathbf{emp} \wedge a2 \neq a3$	✓	1.91
(11)	$(\mathbf{ls}(a1, a2) * \mathbf{true} * a3 \mapsto a4) \wedge (\mathbf{true} * (\mathbf{ls}(a2, a1) \wedge a2 \neq a1))$	✓	32.9
(12)	$(a1 \mapsto a2 * \mathbf{ls}(e1, e2)) \wedge (a2 \mapsto a3 * \neg\mathbf{emp}) \wedge (a3 \mapsto a1 * \neg a5 \mapsto a6 * \mathbf{true})$	✓	39.8
(13)	$(\neg\mathbf{emp} * \neg\mathbf{emp}) \wedge (a1 = \mathtt{nil} \vee a1 \mapsto e1 \vee ((a1 \mapsto e1 \wedge e1 = \mathtt{nil}) * \mathbf{true})) \wedge \mathbf{ls}(a1, a2)$	✓	2.34
(14)	$((\mathbf{ls}(a1, a2) \wedge a1 \neq a2) * (\mathbf{ls}(a2, a3) \wedge a2 \neq a3)) \wedge ((\mathbf{ls}(a4, a1) \wedge a4 \neq a1) * a1 \mapsto e1 * \mathbf{true})$	✓	10.8
(15)	$(\mathbf{ls}(a1, a2) \mathrel{-\!\!\circledast} \mathbf{ls}(a1, a2)) \wedge \neg\mathbf{emp}$	✓	1.92
(16)	$(a3 \mapsto a4 \mathrel{-\!\!\circledast} \mathbf{ls}(a1, a4)) \wedge (a3 = a4 \vee \neg\mathbf{ls}(a1, a3))$	✓	2.76
(17)	$((a2 \mapsto a3 \mathrel{-\!\!\circledast} \mathbf{ls}(a2, a4)) \mathrel{-\!\!\circledast} \mathbf{ls}(a1, a4)) \wedge \neg\mathbf{ls}(a1, a3)$	✓	4.60
(18)	$((a2 \mapsto a3 \mathrel{-\!\!\circledast} \mathbf{ls}(a2, a4)) \mathrel{-\!\!\circledast} \mathbf{ls}(a3, a1)) \wedge a2 = a4$	✓	4.02
(19)	$(a1 \mapsto a2 \mathrel{-\!\!\circledast} \mathbf{ls}(a1, a3)) \wedge (\neg\mathbf{ls}(a2, a3) \vee (\mathbf{true} \wedge (a1 \mapsto e1 * \mathbf{true})) \vee a1 = a3)$	✓	4.12
(20)	$((\mathbf{ls}(a1, a2) \wedge a1 \neq a2) \mathrel{-\!\!\circledast} \mathbf{ls}(e1, e2)) \wedge e1 \neq a1 \wedge e2 = a2 \wedge \neg\mathbf{ls}(e1, a1)$	✓	6.03
(21)	$a1 \neq a4 \wedge (\mathbf{ls}(a1, a4) \mathrel{-\!\!\circledast} \mathbf{ls}(e1, e2)) \wedge a4 = e2 \wedge \neg\mathbf{ls}(e1, a1)$	✓	6.92
(22)	$((\mathbf{ls}(a1, a2) \wedge a1 \neq a2) \mathrel{-\!\!\circledast} \mathbf{ls}(e1, e2)) \wedge e2 \neq a2 \wedge e1 = a1 \wedge \neg\mathbf{ls}(a2, e2)$	?	6.98
(23)	$((a2 \mapsto a3 \mathrel{-\!\!\circledast} \mathbf{ls}(a2, a4)) \mathrel{-\!\!\circledast} \mathbf{ls}(a3, a1)) \wedge (\neg\mathbf{ls}(a4, a1) \vee a2 = a4)$	?	4.71

Table 3: Unsatisfiable formulas.

	Formula	Unsat	Time
(1)	$\neg(a1 \mapsto a2) \wedge \mathbf{emp} \wedge (a1 \mapsto a2 * a3 \mapsto a4)$	✓	3.65
(2)	$a1 \mapsto a2 \wedge \mathbf{emp} \wedge (\neg(a1 \mapsto a2) * a3 \mapsto a4)$	✓	5.01
(3)	$\neg(a1 \mapsto a2) \wedge \mathbf{emp} \wedge (a1 \mapsto a2 * \mathbf{ls}(a3, a4))$	✓	6.17
(4)	$\mathbf{ls}(a1, a2) \wedge \mathbf{emp} \wedge (\neg\mathbf{ls}(a1, a2) * \mathbf{ls}(a3, a4))$	✓	76.2
(5)	$\mathbf{ls}(a1, a2) \wedge \mathbf{emp} \wedge (\neg\mathbf{ls}(a1, a2) * \neg\mathbf{ls}(a3, a4))$	✓	386

Table 4: Example instantiations of $T_1 \stackrel{\text{def}}{=} \neg a \wedge \mathbf{emp} \wedge (a * b)$, where $a, b \in \mathit{lits}$.

We now answer Questions 1–3 posed at the beginning of this section using the three groups of formulas.

Group 1 Results. The running time of our procedure on the formulas listed in Tab. 3 was often on the order of five seconds. The procedure was able to prove unsatisfiability for all formulas, except (22) and (23). We believe that formulas (9)–(23) are beyond the scope of existing tools. Formulas (9)–(14) demonstrate that we can handle formulas that describe overlapping data structures, including conjunctions of separating conjunctions. Formulas (15)–(21) demonstrate that we can handle formulas that contain occurrences of both **ls** and septraction.

	Formula	Unsat	Time
(1)	$\mathbf{emp} \wedge \mathbf{ls}(a1, a2) \wedge (\mathbf{ls}(a3, a4) * (\mathbf{ls}(a5, a6) \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \neg\mathbf{ls}(a1, a2))))$	✓	9.03
(2)	$\mathbf{emp} \wedge \neg\mathbf{emp} \wedge (\mathbf{ls}(a3, a4) * (\neg(a5 \mapsto a6) \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \mathbf{emp})))$	✓	3.25
(3)	$\mathbf{emp} \wedge a1 \mapsto a2 \wedge (a3 \mapsto a4 * (a5 \mapsto a6 \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \neg(a1 \mapsto a2))))$	✓	3.57
(4)	$\mathbf{emp} \wedge \mathbf{ls}(a1, a2) \wedge (\neg\mathbf{ls}(a3, a4) * (\mathbf{ls}(a5, a6) \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \neg\mathbf{ls}(a1, a2))))$	✓	25.3
(5)	$\mathbf{emp} \wedge \mathbf{ls}(a1, a2) \wedge (\neg\mathbf{ls}(a3, a4) * (\neg\mathbf{ls}(a5, a6) \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \neg\mathbf{ls}(a1, a2))))$	✓	25.5

Table 5: Example instantiations of $T_2 \stackrel{\text{def}}{=} \mathbf{emp} \wedge a \wedge (b * (c \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \neg a)))$, where $a, b, c \in \mathit{lits}$.

Group 2 Results. The 64 formulas instantiated from the template $T_1 \stackrel{\text{def}}{=} \neg a \wedge \mathbf{emp} \wedge (a * b)$ took between 0.82 and 386 seconds to check, with a mean of 12.4 and a median of 1.98 seconds. Our procedure was able to prove unsatisfiability for all 64 formulas. All instantiations of T_1 that contain an occurrence of the **ls** predicate are beyond the capabilities of existing tools.

The formulas that took the greatest amount of time and the second-greatest amount of time are (5) and (4), respectively, in Tab. 4. In both cases, a large amount of time was required because of the presence of $\neg\mathbf{ls}$, which is represented by 24 structures—a much larger number than is needed for the other literals.

Group 3 Results. The 512 formulas instantiated from the template $T_2 \stackrel{\text{def}}{=} \mathbf{emp} \wedge a \wedge (b * (c \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \neg a)))$, took between 0.79 and 25.5 seconds to check using our procedure, with a mean of 2.98, and a median of 2.33 seconds. Our procedure was able to prove unsatisfiability for all 512 formulas. All instantiations of T_2 that contain an occurrence of the **ls** predicate are beyond the capabilities of existing tools.

6 Related Work

The literature related to reasoning about separation logic is vast, and we mention only a small portion of it in this section. Decidability results related to first-order separation logic are discussed in [8, 6]. A fragment of separation logic for which it is decidable to check validity of entailments was introduced in [2]. The fragment includes points-to and linked-list predicates; but no septraction, or negations of points-to or linked-list predicates. Most approaches use a syntactic proof-theoretic procedure for this fragment [2, 27]. One exception is the approach by Cook et al. [9], which uses a more semantic approach that represents separation-logic formulas as graphs in a particular normal form and then proves that one formula entails another by computing a homomorphism between the corresponding graphs. More recent approaches deal with fragments of separation logic that are incomparable to ours [26, 22, 19]; in particular, none of these approaches handle linked lists.

The explicit use of abstract values drawn from an abstract domain as a way to represent knowledge in implementations of decision procedures is a technique that has been receiving increased attention of late [14, 33, 32, 15, 16]. As far as we know, our work is the first to apply this idea to a fragment of separation logic.

Many researchers pigeonhole TVLA [23] as a system exclusively tailored for "shape analysis". In fact, it is actually a metasystem for (i) defining a family of logical structures 2-STRUCT[Voc], and (ii) defining canonical-abstraction domains whose elements represent sets of 2-STRUCT[Voc]. The ITVLA [20, §8] variant of TVLA is a different packaging of the classes that make up the TVLA implementation, and demonstrates better that canonical abstraction is a general-purpose method for abstracting the structures that are a logic's domain of discourse.

To simplify matters, the separation-logic fragment addressed in this paper does not allow one to make assertions about numeric-valued variables and numeric-valued fields. Our approach could be extended to support such capabilities using methods developed in work on abstract interpretation that combines canonical abstraction with numeric abstractions [18, 25].

7 Conclusion and Future Work

This paper showed how to create a semi-decision procedure for a fragment of separation logic. The fragment of separation logic that we use has empty-heap assertions (**emp**), equalities ($x = y$), points-to assertions ($x \mapsto y$), acyclic-list-segment assertions ($\mathbf{ls}(x, y)$), and their negations as literals; it provides the connectives $*$, $-\!\!\circledast$, $\wedge$, and $\vee$. We believe that this is an interesting fragment, in that it contains formulas for which existing approaches do not apply.

For each SL formula φ, the procedure performs a bottom-up evaluation of the formula, using a particular shape-analysis interpretation; if the answer is the empty set of 3-valued structures, then φ is unsatisfiable. Thus, the work reported in the paper supports the thesis that abstract-interpretation concepts can help in the design and implementation of decision procedures.

Moreover, if φ is satisfiable, then the procedure reports a set of abstract models—i.e., a value in the canonical-abstraction domain that overapproximates $[\![\varphi]\!]$. As we have shown in other work (using a variety of other techniques, and for a variety of other logics), a decision-procedure-like method that is prepared to return such "residual" answers provides a way to generate sound abstract transformers automatically [28, 35, 33, 31]. In particular, when φ specifies the transition relation between the pre-state and post-state of a concrete transformer τ, a residuating decision procedure provides a way to create a sound abstract transformer $\tau^\sharp$ for τ, directly from a specification in logic of τ's concrete semantics. Consequently, the work reported in the paper also supports the thesis that abstract-interpretation-based decision procedures provide much promise for automating the construction of program-analysis tools. Using our semi-decision procedure, we now have a way to create abstract transformers based on canonical-abstraction domains directly from a specification of the semantics of a language's concrete transformers, written in SL.

Although TVLA and separation logic have both been applied to the problem of analyzing programs that manipulate linked data structures, there has been only rather limited crossover of ideas between the two approaches. Our semi-decision procedure is built on the connection between TVLA states and

SL statelets described in §2.3, which represents the first formal connection between the two approaches. For this reason, the semi-decision procedure should be of interest to both communities: (i) for the TVLA community, the procedure illustrates a different and intriguing use for canonical-abstraction domains; (ii) for the separation-logic community, the procedure shows how using TVLA and canonical-abstraction domains leads to a model-theoretic approach to the decision problem for SL that is capable of handling formulas that are beyond the capabilities of existing tools.

We believe that the approach presented in this paper has the potential to be extended to deal with richer fragments of separation logic—in particular, fragments that contain both separating implication and acyclic linked-list predicates.

References

1. G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. Combining shape analyses by intersecting abstractions. In *VMCAI*, 2006.
2. J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*. 2004.
3. J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: modular automatic assertion checking with separation logic. In *FMCO*, 2005.
4. I. Bogudlov, T. Lev-Ami, T. Reps, and M. Sagiv. Revamping TVLA: Making parametric shape analysis competitive. Tech. Rep. TR-2007-01-01, Tel-Aviv Univ., Tel-Aviv, Israel, 2007.
5. I. Bogudlov, T. Lev-Ami, T. Reps, and M. Sagiv. Revamping TVLA: Making parametric shape analysis competitive (tool paper). In *CAV*, 2007.
6. R. Brochenin, S. Demri, and E. Lozes. On the almighty wand. *Information and Computation*, 211:106–137, 2012.
7. C. Calcagno, V. Vafeiadis, and M. Parkinson. Modular safety checking for fine-grained concurrency. In *Static Analysis Symp.*, 2007.
8. C. Calcagno, H. Yang, and P. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FSTTCS*. 2001.
9. B. Cook, C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*, 2011.
10. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
11. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
12. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
13. D. Distefano and M. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, 2008.
14. V. D'Silva, L. Haller, and D. Kroening. Satisfiability solvers are static analyzers. In *SAS*, 2012.
15. V. D'Silva, L. Haller, and D. Kroening. Abstract conflict driven learning. In *Symp. on Princ. of Prog. Lang.*, 2013.
16. V. D'Silva, L. Haller, and D. Kroening. Abstract satisfaction. In *Symp. on Princ. of Prog. Lang.*, 2014.

17. K. Dudka, P. Muller, P. Peringer, and T. Vojnar. Predator: A tool for verification of low-level list manipulations. In *TACAS*, 2013.
18. D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *TACAS*, pages 512–529, 2004.
19. Z. Hou, R. Clouston, R. Gore, and A. Tiu. Proof search for propositional abstract separation logics via labelled sequents:. In *Symp. on Princ. of Prog. Lang.*, 2014.
20. B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. *TOPLAS*, 32(2), 2010.
21. V. Kuncak and M. Rinard. On the boolean algebra of shape analysis constraints. Technical Report MIT-LCS-TR-916, M.I.T. CSAIL, Aug. 2003.
22. W. Lee and S. Park. A proof system for separation logic with magic wand. In *Symp. on Princ. of Prog. Lang.*, 2014.
23. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, pages 280–301, 2000.
24. S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *Static Analysis Symp.*, 2007.
25. B. McCloskey, T. Reps, and M. Sagiv. Statically inferring complex heap, array, and numeric invariants. In *SAS*, 2010.
26. J. Park, J. Seo, and S. Park. A theorem prover for Boolean BI. In *Symp. on Princ. of Prog. Lang.*, 2013.
27. J. A. N. Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *Prog. Lang. Design and Impl.*, 2011.
28. T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, 2004.
29. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. Symp. on Logic in Comp. Sci.*, 2002.
30. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.
31. A. Thakur, M. Elder, and T. Reps. Bilateral algorithms for symbolic abstraction. In *SAS*, 2012.
32. A. Thakur and T. Reps. A generalization of Stålmarck's method. In *SAS*, 2012.
33. A. Thakur and T. Reps. A method for symbolic computation of precise abstract operations. In *CAV*, 2012.
34. V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.
35. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, 2004.
36. G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. *ACM Trans. Comput. Log.*, 8(1), 2007.

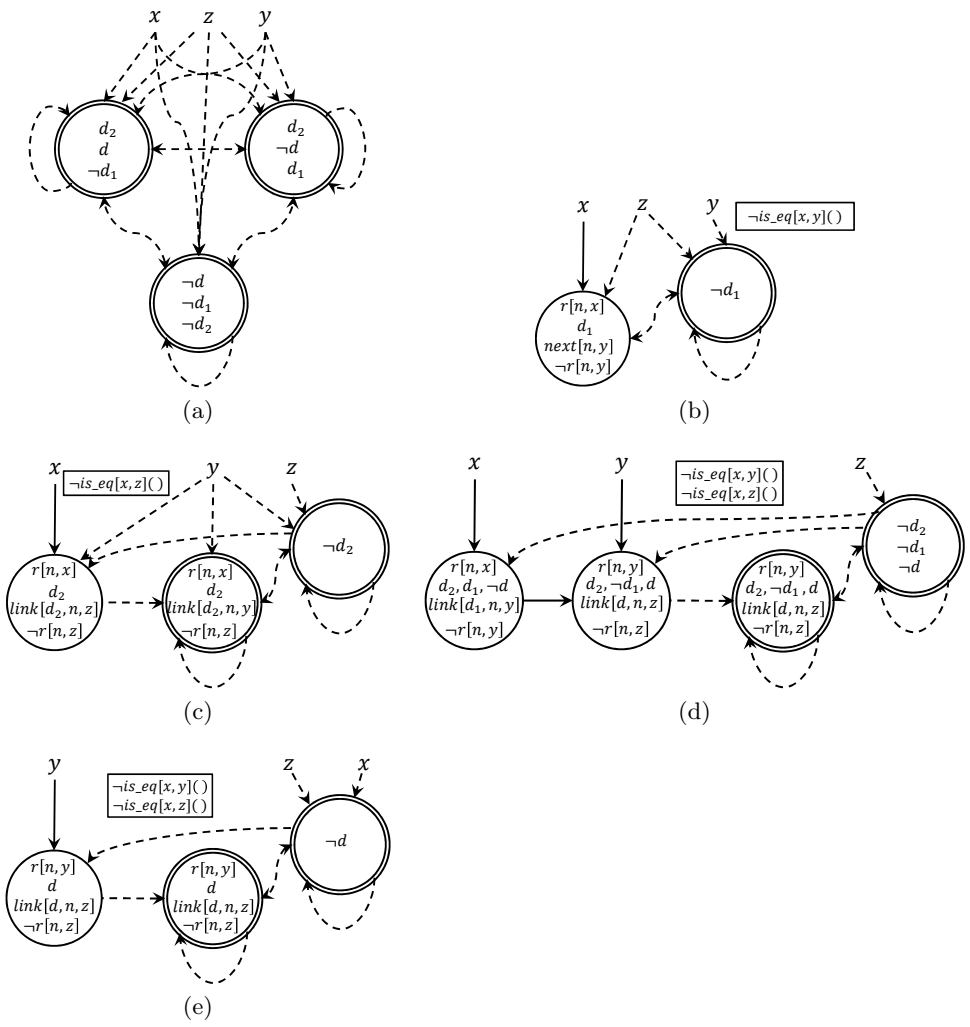

Fig. 7: Some of the structures that arise in the meet operation used to evaluate $x \mapsto y \mathbin{-\!\circledast} \mathbf{ls}(x, z)$.

A A Satisfiable Formula

Consider the formula $\varphi \stackrel{\text{def}}{=} x \mapsto y \mathbin{-\!\circledast} \mathbf{ls}(x, z)$. We want to compute $A \in \mathcal{A}$ such that $\gamma(A)|_{(d,\cdot)} \supseteq [\![\varphi]\!]$. Similar to what was done in §3 for the $*$ operator, we introduce two new domain predicates d_1 and d_2, which are used to demarcate the heaplets that must satisfy $\varphi_1 \stackrel{\text{def}}{=} x \mapsto y$ and $\varphi_2 \stackrel{\text{def}}{=} \mathbf{ls}(x, z)$. By design, there exist $A_1, A_2 \in \mathcal{A}$ such that $\gamma(A_1)|_{(d_1,\cdot)} = [\![x \mapsto y]\!]$ and $\gamma(A_2)|_{(d_2,\cdot)} = [\![\mathbf{ls}(x, z)]\!]$, respectively. A_1 consists of the single 3-valued structure shown in Fig. 7(b). Fig. 7(c) shows one of the structures in A_2; it represents an acyclic linked list from x to z whose length is greater than 1. Furthermore, to satisfy $\varphi_1 \mathbin{-\!\circledast} \varphi_2$, d and d_1 are required to be disjoint regions whose union is d_2. $\mathcal{A}$ also contains an

abstract value, which we will call D, that represents this disjointness constraint exactly. D consists of four 3-valued structures. Fig. 7(a) shows the "most general" of them: it represents two disjoint regions, d and d_1, that partition the d_2 region (where each of d and d_1 contain at least one cell). The summary individual labeled $\neg d, \neg d_1, \neg d_2$ in Fig. 7(a) represents a region that is disjoint from d_2.

To impose a necessary condition for $x \mapsto y \mathrel{-\!\!\circledast} \mathbf{ls}(x, z)$ to be satisfiable, we take the *meet* of D, A_1, and A_2: $[\![x \mapsto y \mathrel{-\!\!\circledast} \mathbf{ls}(x, z)]\!] \subseteq D \sqcap A_1 \sqcap A_2$. Fig. 7(d) shows one of the structures that arises in $D \sqcap A_1 \sqcap A_2$, after the semantic-reduction operators have been applied. A few points to note about this resultant structure:

- The summary individual in region d_2 present in the $\mathbf{ls}(x, z)$ structure in Fig. 7(c) is split in Fig. 7(d) into a singleton individual pointed to by y and a summary individual.
- The individual pointed to by x is in regions d_1 and d_2, but not d.
- The individual pointed to by y is in regions d and d_2, but not d_1.
- The variables x and y are not equal.
- All the individuals in d are reachable from y, not reachable from z, and have $\mathit{link}[d, n, z]$ true.

Fig. 7(e) shows the structure after we have projected the heap onto the heap region d; that is, the values of the domain predicates d_1 and d_2 have been set of 1/2 on all individuals, and all the abstraction predicates have been set to 1/2 on all individuals not in d. In effect, this operation blurs the distinction between the region that is outside d, but in d_2, and the region that is outside of d and d_2. Note that the fact that x and y are not equal is preserved by the projection operation. This projection operation, denoted by $(\cdot)\!\downarrow^d$ in §4, serves as an abstract method for quantifier elimination.

Note that Fig. 7(e) represents an acyclic linked-list from y to z with $x \neq y$, which is one of the models that satisfies $x \mapsto y \mathrel{-\!\!\circledast} \mathbf{ls}(x, z)$.