# 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. [2]GrammaTech, Inc.; Ithaca, NY, USA.
{adi, jbreck, reps}@cs.wisc.edu

## ABSTRACT

Separation logic is an expressive logic for reasoning about heap structures in programs. This paper presents a semi-decision procedure for checking unsatisfiability of formulas in a fragment of separation logic that includes 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 previous approaches.

## Categories and Subject Descriptors

F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—Mechanical theorem proving

## General Terms

Verification, Logic, Reasoning

## Keywords

Separation logic, abstract interpretation, shape analysis

## 1. INTRODUCTION

Separation logic [33] 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 $\varphi_1$ and $\varphi_2$ hold, respectively [33]. A "septraction" ($\varphi_1 \mathbin{-\!\circledast} \varphi_2$) asserts that a heaplet $h$ can be extended by a disjoint heaplet $h_1$ in which $\varphi_1$ holds, to create a heaplet $h_1 \cup h$ in which $\varphi_2$ holds [38]. The $\mathbin{-\!\circledast}$ operator is sometimes called *existential magic wand*, because it is the DeMorgan-dual of the magic-wand operator "$-\!*$" (also called separating implication); i.e., $\varphi_1 \mathbin{-\!\circledast} \varphi_2$ iff $\neg(\varphi_1 -\!* \neg\varphi_2)$.

The use of separation logic in manual, semi-automated, and automated verification tools is a burgeoning field [5, 14, 27, 15, 19]. 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 checking the unsatisfiability of formulas in a fragment of separation logic. The key insight behind our semi-decision procedure is that it is designed using concepts from abstract interpretation [12]. Given a formula $\varphi$, the semi-decision procedure sets up an appropriate abstract domain that is tailored for representing information about the meanings of subformulas of $\varphi$. It uses an abstract domain of shape graphs [34] to represent a set of heap structures. The proof calculus that we present performs a bottom-up evaluation of $\varphi$, using a particular shape-graph interpretation. It computes an abstract value that over-approximates the set of satisfying models of $\varphi$. If the over-approximation is the empty set of shape graphs, then $\varphi$ is unsatisfiable. If $\varphi$ 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 [16, 37, 36, 17, 18], a technique we call *Satisfiability Modulo Abstraction (SMA)*. Our work is the first to apply this idea to a fragment of separation logic.

One of the main advantages of the SMA approach is that it is able to reuse abstract-interpretation machinery to implement decision procedures. In [37], for instance, the polyhedral abstract domain—implemented in PPL [3]—is used to implement a decision procedure for the logic of linear rational arithmetic. In this paper, we use an abstract domain of shapes—implemented in TVLA [34]—in a novel way to implement a semi-decision procedure for separation logic. The challenge was to instantiate the parametric framework of TVLA to represent the literals precisely and to capture the spatial constraints of our fragment of separation logic.

This research is supported, in part, by NSF under grant CCF-0904371; by ONR under grant N00014-11-C-0447; by AFRL under contract FA8650-10-C-7088; and by DARPA under cooperative agreement HR0011-12-2-0012. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies. A. Thakur is supported, in part, by a Google Ph.D. Fellowship. T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

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.

The fragment of separation logic that our approach handles includes points-to assertions ($x \mapsto y$), acyclic-list-segment assertions ($\mathbf{ls}(x, y)$), empty-heap assertions ($\mathbf{emp}$), 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 [4, 30, 29, 25, 22]. The semi-decision procedure can prove *validity* of implications of the form

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

where $\varphi_i$ and $\varphi_j$ are formulas that contain only $\wedge$, $\vee$, and positive or negative occurrences of $\mathbf{emp}$, points-to, or $\mathbf{ls}$ assertions; and $\psi$ and $\psi_j$ 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 $\mathbf{ls}$ and the magic-wand operator $\mathbin{-\!\!*}$. 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*** or *negations below separating-conjunctions*, such as

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

and (ii) formulas that *contain both* ***ls*** *and septraction* ($\mathbin{-\!\!\circledast}$), such as $(a3 \mapsto a4 \mathbin{-\!\!\circledast} \mathbf{ls}(a1, a4)) \wedge (a3 = a4 \vee \neg\mathbf{ls}(a1, a3))$. The former are useful for describing overlaid data structures; the latter are useful in dealing with interference effects when using rely/guarantee reasoning to verify programs with fine-grained concurrency [38, 9].

The contributions of our work 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 $\mathbf{ls}$ assertions—and hence can prove the validity of implications of the kind shown in formula (1) (§4).

§3 illustrates the key concepts used in our semi-decision procedure. Our semi-decision procedure is implemented in a tool called SMASLTOV (Satisfiability Modulo Abstraction for Separation Logic ThrOugh Valuation), which is available at [1]. We evaluated SMASLTOV on a set of formulas taken from the literature (§5). To the best of our knowledge, SMASLTOV is able to establish the unsatisfiability of formulas that cannot be handled by previous approaches.

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

Figure 1: **Satisfaction of an SL formula with respect to a statelet.**

## 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:

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

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* $s$ and a *heaplet* $h$. 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 not containing nil.

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

*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 $\varphi$ with respect to statelet $(s, h)$ is defined in Fig. 1. Furthermore, in this paper, we consider a formula to be satisfiable only if it is satisfiable over an *acyclic* heap. $[\![\varphi]\!]$ denotes the set of statelets that satisfy $\varphi$: $[\![\varphi]\!] \stackrel{\text{def}}{=} \{(s, h) \mid (s, h) \models \varphi\}$.

### 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). $\mathrm{Voc}_k$ denotes the set of $k$-ary symbols.

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 $\iota$ is the* ***interpretation****. Let $\mathbb{B} = \{0, 1\}$ be the domain of*

Table 1: **Core predicates used when representing states made up of acyclic linked lists.**

| 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 $v_2$? |

*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. Tab. 1 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. The set of unary predicates, $\mathrm{Voc}_1$, always contains predicates that encode the variables of the formula. In a minor abuse of notation, we overload "$x$" to denote both the name of variable $x$ and the unary predicate $x(\cdot)$ that encodes the variable. The binary predicate $n \in \mathrm{Voc}_2$ encodes list-node linkages. In essence, the following integrity rules restrict each $x \in \mathit{Var} \subseteq \mathrm{Voc}_1$ to serve as a constant, and restrict relation $n$ to encode a partial function:

$$\begin{aligned} \text{for each } x \in \mathit{Var}, \forall v_1, v_2 : x(v_1) \wedge x(v_2) &\quad\Rightarrow\quad eq(v_1, v_2) \\ \forall v_1, v_2, v_3 : n(v_3, v_1) \wedge n(v_3, v_2) &\quad\Rightarrow\quad eq(v_1, v_2) \end{aligned}$$

## 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 $d$:

$$S|_{(d,\cdot)} \overset{\text{def}}{=} (s, h), \text{where}$$

$$s = \left( \begin{array}{ll} & \{(p,u) \mid p \in \mathit{Var}^S, u \in U^S, \text{and } p(u)\} \\ \cup & \{(q,\mathtt{nil}) \mid q \in \mathit{Var}^S \text{ and } \neg\exists v : q(v)\} \end{array} \right) \quad (2)$$

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

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: $\mathrm{SS}|_{(d,\cdot)} \overset{\text{def}}{=} \{S|_{(d,\cdot)} \mid S \in \mathrm{SS}\}$.

## 2.4 Representing Sets of SL Statelets using Canonical Abstraction

In the framework of Sagiv et al. [34] 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 $\mathbb{T} \overset{\text{def}}{=} \mathbb{B} \cup \{1/2\}$ 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 $\iota$ 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. 2). A summary individual is depicted with a double-ruled border. A unary predicate $p \in \mathit{Var}$ 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. (To reduce clutter, in the figures in this paper, the only binary predicate shown is the predicate $n \in \mathrm{Voc}_2$.) 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 $p \in (\mathrm{Voc}_1 - \mathit{Var})$ 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 [34]:

Definition 3. *Given $S = \langle U, \iota \rangle$ and $S' = \langle U', \iota' \rangle$, two 3-valued structures over the same vocabulary Voc, and $f\colon U \to U'$, 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 $f\colon$ 3-STRUCT[Voc] $\to$ 3-STRUCT[Voc] induced by $f\colon U \to U'$.)*

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 $S \sqsubseteq^f S' \Leftrightarrow f(S) \sqsubseteq^{\mathrm{id}} S'$. Henceforth, we use $S \sqsubseteq^f S'$ to mean "there exists a surjective $f : U \to U'$ such that $f(S) \sqsubseteq^{\mathrm{id}} S'$".

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. The idea behind *canonical abstraction* [34, §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 $f^S_{\mathbb{A}} \colon U \to (U/\simeq_{\mathbb{A}^S})$, which maps an individual to its equivalence class. We thus have the Galois connection

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

where $f^S_{\mathbb{A}}$ in the definition of $\alpha$ 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 $\alpha$ 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 $\alpha$: $|U^\sharp| \leq 2^{|\mathbb{A}|}$—and thus the power-set of the image of $\alpha$ 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 $\alpha$; however, we still want to restrict ourselves to a finite sublattice of $\wp$(3-STRUCT[Voc]). With this motivation, we make the following definition [2]:

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 $p \in \mathbb{A} \subseteq \text{Voc}_1$ 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

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

The ordering on $\wp$(B-STRUCT[Voc, $\mathbb{A}$]) $= \mathcal{A}$[Voc, $\mathbb{A}$] is the Hoare ordering: $S_1 \sqsubseteq S_2$ if for all $s_1 \in S_1$ there exists $s_2 \in S_2$ such that $s_1 \sqsubseteq^f s_2$.

## 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 heaps: $x \mapsto y * y \mapsto x$. App. A illustrates the procedure using a formula that is satisfiable over acyclic heaps: $x \mapsto y \mathbin{-\!\!\circledast} \mathbf{ls}(x, z)$.

Consider $\varphi \overset{\text{def}}{=} x \mapsto y * y \mapsto x$. 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 $\varphi_1 \overset{\text{def}}{=} x \mapsto y$ and $\varphi_2 \overset{\text{def}}{=} y \mapsto x$, 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. 2 describes the abstraction predicates we use. $A_1$ and $A_2$ each consist of a single 3-valued structure, shown in Fig. 2(b) and Fig. 2(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. 2(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 at least one cell). The summary individual labeled $\neg d, \neg d_1, \neg d_2$ in Fig. 2(a) represents a region that is disjoint from $d$. (See also Fig. 5.)

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. 2(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. [2] show that in general this operation is NP-complete; however, they define an algorithm based on graph matching that typically performs well in practice [23, §8.3]. To understand some of the subtleties of meet, consider Fig. 2(d), which shows one of the structures in $D \sqcap A_1$ (i.e., Fig. 2(a) $\sqcap$ Fig. 2(b)).

- From the standpoint of Fig. 2(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. 2(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 $\mathit{next}[n, y]$, and (iii) acquire the value 0 for $y$ and $r[n, y]$.

Fig. 2(e) shows one of the structures in $(D \sqcap A_1) \sqcap A_2$, i.e., Fig. 2(d) $\sqcap$ Fig. 2(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 [2, §4.2].

**Improving Precision Using Semantic-Reduction Operators.** Fig. 2(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 $\mathit{next}[n, y]$, which holds for individuals whose $n$-link points to the individual that is pointed to by $y$. This predicate has an associated integrity constraint

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

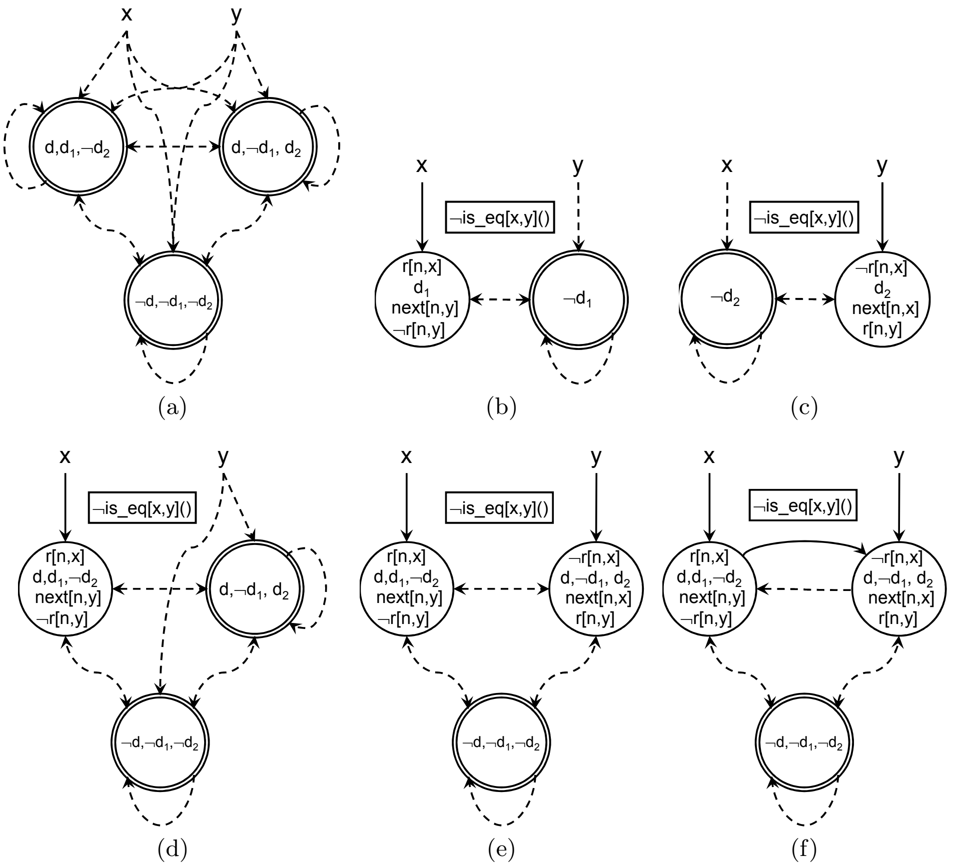


Figure 2: **Structures that arise in the meet operation used to analyze $x \mapsto y * y \mapsto x$.**

In particular, in Fig. 2(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. Similarly, we force the semi-decision procedure to consider only acyclic heaps by imposing the integrity constraint $\neg\exists v_1, v_2.n(v_1, v_2) \wedge t[n](v_2, v_1)$.

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 [13]. 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 $\rho$ 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, $\rho$ 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. 2(e).

- Fig. 2(f) is Fig. 2(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. 2(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. 2(f): the first 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. [34, §6.4] and Bogudlov et al. [6, 7] discuss algorithms for *coerce*.

## 4. PROOF SYSTEM

This section describes how we compute $A \in \mathcal{A}[\text{Voc}, \mathbb{A}]$ such that $A$ overapproximates the satisfying models of $\varphi \in$ SL. The vocabulary Voc and abstraction predicates $\mathbb{A}$ are listed in Tab. 2.

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,

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

| 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 $y$ |

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

$$\frac{\varphi_1, d \Vdash S_1 \qquad \varphi_2, d \Vdash A_2}{\varphi_1 \vee \varphi_2, d \Vdash A_1 \sqcup A_2}\ (\vee)$$

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

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

Figure 3: **Rules for computing an abstract value that overapproximates the meaning of a formula in SL.**

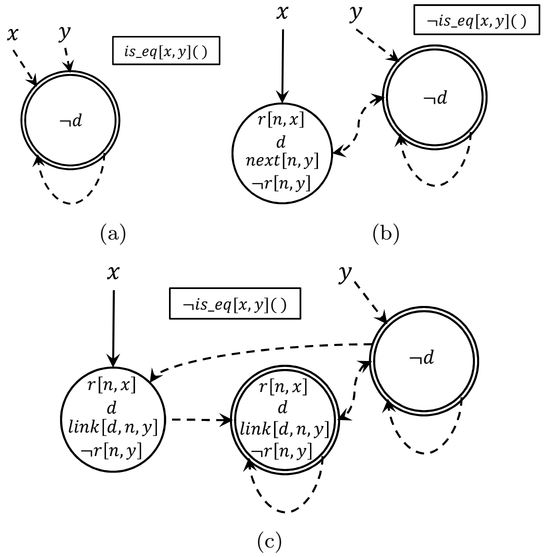


Figure 4: **The abstract value for $\mathbf{ls}(x,y) \in$ *atom* in the canonical-abstraction domain.**

whenever it establishes a judgment $\varphi, d \Vdash A$, $A \in \mathcal{A}$ overapproximates $\varphi$ in the following sense: $\gamma(A)|_{(d,\cdot)} \supseteq [\![\varphi]\!]$. Fig. 3 lists the rules used for calculating $\varphi, d \Vdash A$ for $\varphi \in$ SL. Using these rules, the semi-decision procedure performs a bottom-up evaluation of the formula $\varphi$; if the answer is the empty set of 3-valued structures, then $\varphi$ is unsatisfiable.

For each literal $\ell \in lits$, there is an abstract value $A_\ell \in \mathcal{A}$ such that $\gamma(A_\ell)|_{(d,\cdot)} = [\![\ell]\!]$. These $A_\ell$ values are used in the ($\ell$)-rule of Fig. 3. Fig. 4 shows the abstract value $A_{\mathbf{ls}}$ used for $\mathbf{ls}(x,y)$. $A_{\mathbf{ls}}$ consists of three structures:

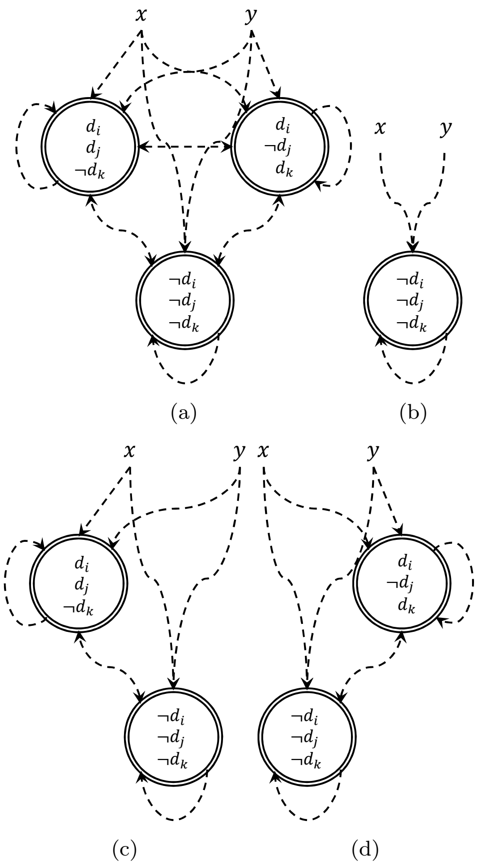


Figure 5: **The abstract value for $[d_i = d_j \cdot d_k]^\sharp$ in the canonical-abstraction domain.**

- Fig. 4(a) represents the empty list from $x$ to $y$. That is, $x = y$ and region $d$ is empty.
- Fig. 4(b) represents a singleton list from $x$ to $y$. That is, $x \neq y$ and $x \neq$ `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. 2.)
- Fig. 4(c) represents acyclic linked lists of length two or more from $x$ to $y$.

Fig. 4(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 [24, 40], 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. 3 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 $\varphi_1$ and $\varphi_2$, respectively. That is,

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

- $[d = d_1 \cdot d_2]^\sharp \in \mathcal{A}$ 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:

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

  Fig. 5 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

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

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. 3 are sound; that is, if the rules in Fig. 3 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.

**Discussion.** As discussed in [31, §4], there exist no methods that handle negations below a separating conjunction. Our fragment of separation logic admits negations at the leaves of formulas, and, thus, is the first approach that can handle formulas with negations below a separating conjunction.

It is, however, non-trivial to extend our technique to handle general negation. Let $(\cdot)^{\mathsf{c}}$ denote the set-complement operation. Let $\neg^{\#}(\cdot)$ denote the abstract negation operation; that is, $\gamma(\neg^{\#}(A)) \supseteq \gamma(A)^{\mathsf{c}}$, and $\neg^{\#}(A) \sqsupseteq \alpha(\gamma(A)^{\mathsf{c}})$. Suppose that $\gamma(A)|_{(d,\cdot)} \supseteq [\![\varphi]\!]$; in general, $\gamma(\neg^{\#}(A))|_{(d,\cdot)}$ is not guaranteed to overapproximate the models of $\neg\varphi$.

Furthermore, it is non-trivial to extend our technique to prove validity of general implications. Suppose that we would like to prove the validity of $\varphi_1 \Rightarrow \varphi_2$, where $\varphi_1, \varphi_2 \in \mathtt{SL}$. Let $A_1$ overapproximate the set of models of $\varphi_1$, and $A_2$ overapproximate the set of models of $\varphi_2$. $A_1 \sqsubseteq A_2$ does not imply $[\![\varphi_1]\!] \subseteq [\![\varphi_2]\!]$.

# 5. EXPERIMENTAL EVALUATION

This section presents the results of our experiments to evaluate the costs and benefits of our approach. Our implementation, which is called SMASLTOV, is available together with our benchmarks at [1]. The experiments were designed to shed light on the following questions:

1. How fast is the semi-decision procedure?
2. How often is the semi-decision procedure able to determine that a formula is unsatisfiable?
3. For unsatisfiable formulas that are beyond the capabilities of other tools, is the semi-decision procedure actually able to prove the formulas unsatisfiable?

**Setup.** The semi-decision procedure is written in OCaml; it compiles a formula to a proof DAG written in the language of ITVLA [23, §8]. We ported the frontend of ITVLA to the latest version of TVLA [26] in order to make use of TVLA's enhanced speed [6] and ITVLA's language features. 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 [2] for a canonical-abstraction domain. The abstract-value manipulations in the proof rules of Fig. 3 are performed by the TVLA backend. TVLA has a significant startup cost and a smaller shutdown cost. We chose to amortize these costs by running TVLA in a batch mode, in which a single invocation of TVLA checks several separation-logic formulas.

We report trimmed means of all time measurements; that is, we made each measurement five times, discarded the highest and lowest values, and report the mean of the remaining three values. 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. We tested each group with a single invocation of TVLA.

- Group 1, shown in Tab. 4, 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 $T_1 \overset{\text{def}}{=} \neg a \wedge \mathbf{emp} \wedge (a * b)$ with the 8 literals *lits* of $\mathtt{SL}$; 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. 5.

- Group 3 was created by replacing the Boolean variables $a$, $b$, and $c$ in the template $T_2 \overset{\text{def}}{=} \mathbf{emp} \wedge a \wedge (b * (c -\!\!\circledast (\mathbf{emp} \wedge \neg a)))$ with the 8 literals *lits* of $\mathtt{SL}$. Five of the 512 instantiations of template $T_2$ are shown in Tab. 6.

Templates $T_1$ and $T_2$ are based on work by Hou et al. [22] on Boolean separation logic. Templates $T_1$ and $T_2$ are listed as formulas 15 and 19, respectively, in [22, Tab. 2]. In total, there were 599 formulas in our test suite. Tab. 3 summarizes the characteristics of the corpus based on the occurrences of the $\mathtt{SL}$ operators.

In Tabs. 4, 5, and 6, a ✓ in the U-column indicates that the semi-decision procedure was able to prove the formula

Table 3: **Number of formulas that contain each of the SL operators in Groups 1, 2, and 3. The columns labeled "+" and "−" indicate the number of atoms occurring as positive and negative literals, respectively.**

| | $\mathbf{emp}$ | | $x = y$ | | $x \mapsto y$ | | $\mathbf{ls}(x, y)$ | | $\varphi \wedge \varphi$ | $\varphi \vee \varphi$ | $\varphi * \varphi$ | $\varphi \mathbin{-\!\!\circledast} \varphi$ | Full |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | + | − | + | − | + | − | + | − | | | | | 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 4: **Unsatisfiable formulas. The time is in seconds.**

| | Formula | U | Time |
|---|---|---|---|
| **(1)** | $a1 \mapsto a2 \wedge \neg\mathbf{ls}(a1, a2)$ | ✓ | 0.12 |
| **(2)** | $a1 \mapsto a2 * a2 \mapsto a1$ | ✓ | 0.08 |
| **(3)** | $\neg\mathbf{emp} \wedge (\mathbf{ls}(a1, a2) * \mathbf{ls}(a2, a1))$ | ✓ | 0.27 |
| **(4)** | $a1 \neq a2 \wedge (\mathbf{ls}(a1, a2) * \mathbf{ls}(a2, a1))$ | ✓ | 0.25 |
| **(5)** | $(\mathbf{ls}(a1, a2) * \mathbf{ls}(a2, a3)) \wedge \neg\mathbf{ls}(a1, a3)$ | ✓ | 0.85 |
| **(6)** | $\mathbf{ls}(a1, a2) \wedge \mathbf{emp} \wedge a1 \neq a2$ | ✓ | 0.09 |
| **(7)** | $(a1 \mapsto a2 * \mathbf{true}) \wedge (a2 \mapsto a3 * \mathbf{true}) \wedge (\mathbf{true} * a3 \mapsto a1)$ | ✓ | 0.72 |
| **(8)** | $(a1 \mapsto a2 \mathbin{-\!\!\circledast} \mathbf{true}) \wedge (a1 \mapsto a2 * \mathbf{true})$ | ✓ | 0.77 |
| **(9)** | $(\mathbf{ls}(a1, a2) * \neg\mathbf{ls}(a2, a3)) \wedge \mathbf{ls}(a1, a3)$ | ✓ | 2.02 |
| **(10)** | $\mathbf{ls}(a1, a2) \wedge \mathbf{ls}(a1, a3) \wedge \neg\mathbf{emp} \wedge a2 \neq a3$ | ✓ | 0.13 |
| **(11)** | $(\mathbf{ls}(a1, a2) * \mathbf{true} * a3 \mapsto a4) \wedge (\mathbf{true} * (\mathbf{ls}(a2, a1) \wedge a2 \neq a1))$ | ✓ | 7.94 |
| **(12)** | $(a1 \mapsto a2 * \mathbf{ls}(e1, e2)) \wedge (a2 \mapsto a3 * \neg\mathbf{emp}) \wedge (a3 \mapsto a1 * \neg a5 \mapsto a6 * \mathbf{true})$ | ✓ | 4.64 |
| **(13)** | $(\neg\mathbf{emp} * \neg\mathbf{emp}) \wedge (a1 = \mathtt{nil} \vee a1 \mapsto e1 \vee ((a1 \mapsto e1 \wedge e1 = \mathtt{nil}) * \mathbf{true})) \wedge \mathbf{ls}(a1, a2)$ | ✓ | 0.20 |
| **(14)** | $((\mathbf{ls}(a1, a2) \wedge a1 \neq a2) * (\mathbf{ls}(a2, a3) \wedge a2 \neq a3)) \wedge ((\mathbf{ls}(a4, a1) \wedge a4 \neq a1) * a1 \mapsto e1 * \mathbf{true})$ | ✓ | 1.45 |
| **(15)** | $(\mathbf{ls}(a1, a2) \mathbin{-\!\!\circledast} \mathbf{ls}(a1, a2)) \wedge \neg\mathbf{emp}$ | ✓ | 0.18 |
| **(16)** | $(a3 \mapsto a4 \mathbin{-\!\!\circledast} \mathbf{ls}(a1, a4)) \wedge (a3 = a4 \vee \neg\mathbf{ls}(a1, a3))$ | ✓ | 0.20 |
| **(17)** | $((a2 \mapsto a3 \mathbin{-\!\!\circledast} \mathbf{ls}(a2, a4)) \mathbin{-\!\!\circledast} \mathbf{ls}(a1, a4)) \wedge \neg\mathbf{ls}(a1, a3)$ | ✓ | 0.65 |
| **(18)** | $((a2 \mapsto a3 \mathbin{-\!\!\circledast} \mathbf{ls}(a2, a4)) \mathbin{-\!\!\circledast} \mathbf{ls}(a3, a1)) \wedge a2 = a4$ | ✓ | 0.62 |
| **(19)** | $(a1 \mapsto a2 \mathbin{-\!\!\circledast} \mathbf{ls}(a1, a3)) \wedge (\neg\mathbf{ls}(a2, a3) \vee (\mathbf{true} \wedge (a1 \mapsto e1 * \mathbf{true})) \vee a1 = a3)$ | ✓ | 0.45 |
| **(20)** | $((\mathbf{ls}(a1, a2) \wedge a1 \neq a2) \mathbin{-\!\!\circledast} \mathbf{ls}(e1, e2)) \wedge e1 \neq a1 \wedge e2 = a2 \wedge \neg\mathbf{ls}(e1, a1)$ | ✓ | 0.88 |
| **(21)** | $a1 \neq a4 \wedge (\mathbf{ls}(a1, a4) \mathbin{-\!\!\circledast} \mathbf{ls}(e1, e2)) \wedge a4 = e2 \wedge \neg\mathbf{ls}(e1, a1)$ | ✓ | 1.23 |
| **(22)** | $((\mathbf{ls}(a1, a2) \wedge a1 \neq a2) \mathbin{-\!\!\circledast} \mathbf{ls}(e1, e2)) \wedge e2 \neq a2 \wedge e1 = a1 \wedge \neg\mathbf{ls}(a2, e2)$ | ✓ | 0.89 |
| **(23)** | $((a2 \mapsto a3 \mathbin{-\!\!\circledast} \mathbf{ls}(a2, a4)) \mathbin{-\!\!\circledast} \mathbf{ls}(a3, a1)) \wedge (\neg\mathbf{ls}(a4, a1) \vee a2 = a4)$ | ? | 0.71 |

Table 5: **Example instantiations of $T_1 \overset{\text{def}}{=} \neg a \wedge \mathbf{emp} \wedge (a * b)$, where $a, b \in$ *lits*. The time is in seconds.**

| | Formula | U | Time |
|---|---|---|---|
| **(1)** | $\neg(a1 \mapsto a2) \wedge \mathbf{emp} \wedge (a1 \mapsto a2 * a3 \mapsto a4)$ | ✓ | 0.83 |
| **(2)** | $a1 \mapsto a2 \wedge \mathbf{emp} \wedge (\neg(a1 \mapsto a2) * a3 \mapsto a4)$ | ✓ | 0.32 |
| **(3)** | $\neg(a1 \mapsto a2) \wedge \mathbf{emp} \wedge (a1 \mapsto a2 * \mathbf{ls}(a3, a4))$ | ✓ | 0.62 |
| **(4)** | $\mathbf{ls}(a1, a2) \wedge \mathbf{emp} \wedge (\neg\mathbf{ls}(a1, a2) * \mathbf{ls}(a3, a4))$ | ✓ | 8.46 |
| **(5)** | $\mathbf{ls}(a1, a2) \wedge \mathbf{emp} \wedge (\neg\mathbf{ls}(a1, a2) * \neg\mathbf{ls}(a3, a4))$ | ✓ | 10.3 |

unsatisfiable; a ? indicates that the semi-decision procedure was not able to prove the formula unsatisfiable.

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 (see App. A).

Table 6: **Example instantiations of $T_2 \overset{\text{def}}{=} \mathbf{emp} \wedge a \wedge (b * (c \mathbin{-\!\!\circledast} (\mathbf{emp} \wedge \neg a)))$, where $a, b, c \in$ *lits*. The time is in seconds.**

| | Formula | U | Time |
|---|---|---|---|
| **(1)** | $\mathbf{emp} \wedge \mathbf{ls}(a1, a2) \wedge (\mathbf{ls}(a3, a4) * (\mathbf{ls}(a5, a6) \mathbin{-\!\!\circledast} (\mathbf{emp} \wedge \neg\mathbf{ls}(a1, a2))))$ | ✓ | 0.37 |
| **(2)** | $\mathbf{emp} \wedge \neg\mathbf{emp} \wedge (\mathbf{ls}(a3, a4) * (\neg(a5 \mapsto a6) \mathbin{-\!\!\circledast} (\mathbf{emp} \wedge \mathbf{emp})))$ | ✓ | 0.17 |
| **(3)** | $\mathbf{emp} \wedge a1 \mapsto a2 \wedge (a3 \mapsto a4 * (a5 \mapsto a6 \mathbin{-\!\!\circledast} (\mathbf{emp} \wedge \neg(a1 \mapsto a2))))$ | ✓ | 0.49 |
| **(4)** | $\mathbf{emp} \wedge \neg\mathbf{ls}(a1, a2) \wedge (\neg\mathbf{ls}(a3, a4) * (\mathbf{ls}(a5, a6) \mathbin{-\!\!\circledast} (\mathbf{emp} \wedge \mathbf{ls}(a1, a2))))$ | ✓ | 3.97 |
| **(5)** | $\mathbf{emp} \wedge \neg\mathbf{ls}(a1, a2) \wedge (\neg\mathbf{ls}(a3, a4) * (\mathbf{emp} \mathbin{-\!\!\circledast} (\mathbf{emp} \wedge \mathbf{ls}(a1, a2))))$ | ✓ | 9.51 |

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. 4 was often on the order of one second. The TVLA startup and shutdown time for Group 1 was 10.9 seconds. The procedure was able to prove unsatisfiability for all formulas, except (23). We believe that formulas (9)–(23) are beyond the scope of previously 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 containing **ls** and septraction together.

**Group 2 Results.** The 64 formulas instantiated from the template $T_1 \overset{\text{def}}{=} \neg a \wedge \mathbf{emp} \wedge (a * b)$ took between 0.0003 and 10.31 seconds to check, with a mean of 0.56 and a median of 0.03 seconds. Our procedure was able to prove unsatisfiability for all 64 formulas. The TVLA startup and shutdown time for Group 2 was 3.39 seconds. 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 most time were (5) and (4) in Tab. 5. In both cases, a large amount of time was required because of the presence of ¬**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 $T_2 \overset{\text{def}}{=} \mathbf{emp} \wedge a \wedge (b * (c \mathbin{-\!\!\circledast} (\mathbf{emp} \wedge \neg a)))$ took between 0.0001 and 9.51 seconds to check using our procedure, with a mean of 0.12, and a median of 0.04 seconds. Our procedure was able to prove unsatisfiability for all 512 formulas. The TVLA startup and shutdown time for Group 3 was 10.12 seconds. All instantiations of $T_2$ that contain an occurrence of **ls** 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 [10, 8]. A fragment of separation logic for which it is decidable to check validity of entailments was introduced by Berdine et al. [4]. The fragment includes points-to and linked-list predicates, but no septraction, or negations of points-to or linked-list predicates. More recent approaches deal with fragments of separation logic that are incomparable to ours [29, 25, 22]; in particular, none of the latter papers handle linked lists. We based our experiments on formulas listed in Hou et al.'s work on Boolean separation logic [22]—the only paper we found that listed formulas outside the syntactic fragment defined by Berdine et al. We believe that our technique represents the first important step in designing a verification system that uses a richer fragment of separation logic.

Most approaches to separation-logic reasoning use a syntactic proof-theoretic procedure [4, 30]. Two exceptions are the approaches of Cook et al. [11] and Enea et al. [20], which use a more semantics-based approach: they represent separation-logic formulas as graphs in a particular normal form, and then prove that one formula entails another by finding a homomorphism between the corresponding graphs. Our approach is also semantics-based, but has more of an algebraic flavor: our method performs a bottom-up evaluation of a formula $\varphi$ using a particular shape-analysis interpretation (Fig. 3); if the answer is the empty set of 3-valued structures, then $\varphi$ is unsatisfiable.

To deal with overlaid data-structures, Enea et al. [20] introduce the $*_w$ operator: the $*_w$ operator specifies data structures that share sets of objects as long as they are built over disjoint sets of *fields*. Their approach, however, does not handle conjunctions of separating conjunctions or negations of the **ls**-predicate. Thus, [20] cannot handle formulas (9)–(14) in Tab. 4, even though these formulas do not contain septraction. Note that, for instance, the logical conjunction in formula (9) cannot be replaced by the $*_w$ operator.

Piskac et al. [31] present a decision procedure for a decidable fragment of separation logic based on a reduction to a particular decidable first-order theory. Unlike our approach, the approach in [31] does not handle septraction or negations below a separating conjunction.

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 [16, 37, 36, 17, 18]. Our work is the first to apply this idea to a fragment of separation logic.

Many researchers pigeonhole TVLA [26] 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 [23, §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 that combines canonical abstraction with numeric abstractions [21, 28].

# 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 $\vee$. This fragment contains formulas that cannot be handled by previous approaches.

For each SL formula $\varphi$, 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 $\varphi$ 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 $\varphi$ 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 [32, 39, 37, 35]. In particular, when $\varphi$ specifies the transition relation between the pre-state and post-state of a concrete transformer $\tau$, a residuating decision procedure provides a way to create a sound abstract transformer $\tau^\sharp$ for $\tau$, directly from a specification in logic of $\tau$'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. The domains that we use are tailored for the particular formula, but, more importantly, provide an encoding that can be connected to the SL semantics: see Eqns. (2) and (3) in §2.3, and the use of domain predicates to express disjointness in §3. (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 our approach 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.

# 8. REFERENCES

[1] https://www.github.com/smasltov-team/SMASLTOV.

[2] G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. Combining shape analyses by intersecting abstractions. In *VMCAI*, 2006.

[3] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *SCP*, 72(1–2):3–21, 2008.

[4] J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*. 2004.

[5] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: modular automatic assertion checking with separation logic. In *FMCO*, 2005.

[6] I. Bogudlov, T. Lev-Ami, T. Reps, and M. Sagiv. Revamping TVLA: Making parametric shape analysis competitive. In *CAV*, 2007.

[7] 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.

[8] R. Brochenin, S. Demri, and E. Lozes. On the almighty wand. *Information and Computation*, 211:106–137, 2012.

[9] C. Calcagno, V. Vafeiadis, and M. Parkinson. Modular safety checking for fine-grained concurrency. In *SAS*, 2007.

[10] C. Calcagno, H. Yang, and P. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FSTTCS*. 2001.

[11] B. Cook, C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*, 2011.

[12] 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.

[13] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.

[14] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.

[15] D. Distefano and M. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, 2008.

[16] V. D'Silva, L. Haller, and D. Kroening. Satisfiability solvers are static analyzers. In *SAS*, 2012.

[17] V. D'Silva, L. Haller, and D. Kroening. Abstract conflict driven learning. In *POPL*, 2013.

[18] V. D'Silva, L. Haller, and D. Kroening. Abstract satisfaction. In *POPL*, 2014.

[19] K. Dudka, P. Muller, P. Peringer, and T. Vojnar. Predator: A tool for verification of low-level list manipulations. In *TACAS*, 2013.

[20] C. Enea, V. Saveluc, and M. Sighireanu. Compositional invariant checking for overlaid and nested linked lists. In *ESOP*, 2013.

[21] D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *TACAS*, 2004.

[22] Z. Hou, R. Clouston, R. Gore, and A. Tiu. Proof search for propositional abstract separation logics via labelled sequents:. In *POPL*, 2014.

[23] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. *TOPLAS*, 32(2), 2010.

[24] 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.

[25] W. Lee and S. Park. A proof system for separation logic with magic wand. In *POPL*, 2014.

[26] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, 2000.

[27] S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, 2007.

[28] B. McCloskey, T. Reps, and M. Sagiv. Statically inferring complex heap, array, and numeric invariants. In *SAS*, 2010.

[29] J. Park, J. Seo, and S. Park. A theorem prover for Boolean BI. In *POPL*, 2013.

[30] J. A. N. Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, 2011.

[31] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using SMT. In *CAV*, 2013.

[32] T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, 2004.

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

[34] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.

[35] A. Thakur, M. Elder, and T. Reps. Bilateral algorithms for symbolic abstraction. In *SAS*, 2012.

[36] A. Thakur and T. Reps. A generalization of Stålmarck's method. In *SAS*, 2012.

[37] A. Thakur and T. Reps. A method for symbolic computation of precise abstract operations. In *CAV*, 2012.

[38] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.

[39] G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, 2004.

[40] G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. *ACM Trans. Comput. Log.*, 8(1), 2007.

# APPENDIX

## A. A SATISFIABLE FORMULA

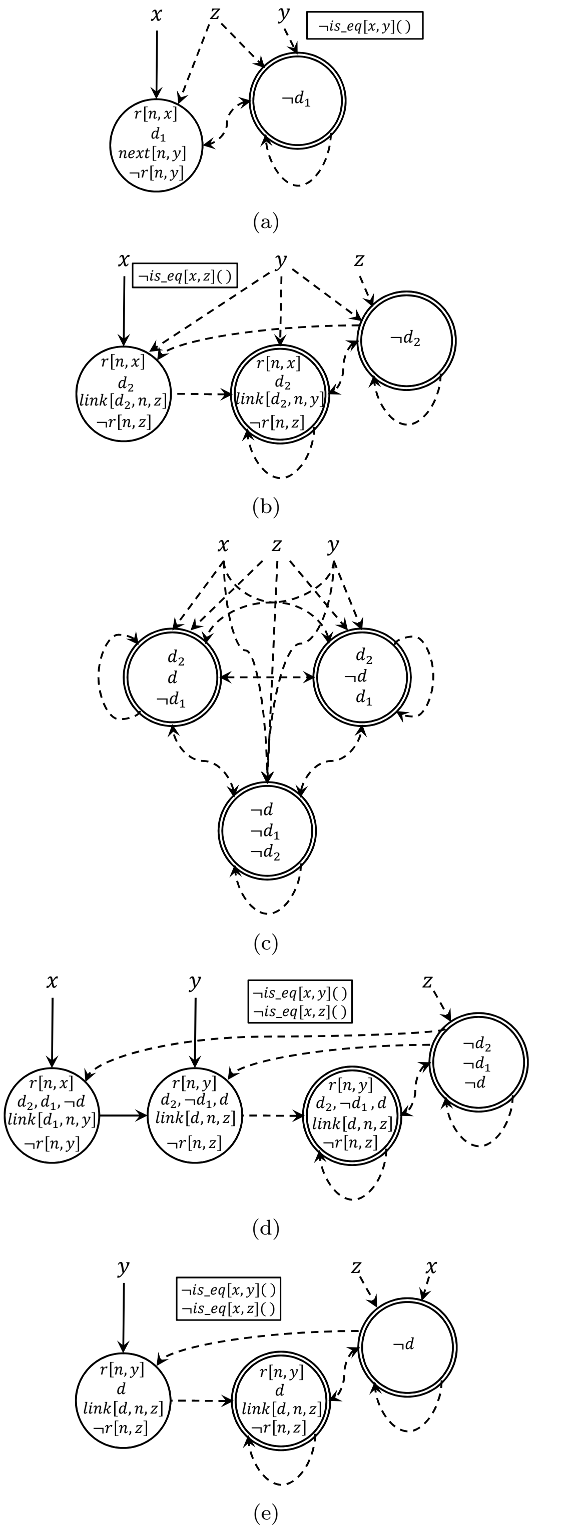
Consider the formula $\varphi \overset{\text{def}}{=} x \mapsto y \mathbin{-\!\!\circledast} \mathbf{ls}(x,z)$. We want to compute $A \in \mathcal{A}$ such that $\gamma(A)|_{(d,\cdot)} \supseteq [\![\varphi]\!]$. Similar to what was done in §3 for the $*$ operator, we introduce two new domain predicates $d_1$ and $d_2$, which are used to demarcate the heaplets that must satisfy $\varphi_1 \overset{\text{def}}{=} x \mapsto y$ and $\varphi_2 \overset{\text{def}}{=} \mathbf{ls}(x,z)$. 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. 6(a). Fig. 6(b) 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. 6(c) 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. 6(c) represents a region that is disjoint from $d_2$.

To impose a necessary condition for $x \mapsto y \mathbin{-\!\!\circledast} \mathbf{ls}(x,z)$ to be satisfiable, we take the *meet* of $D$, $A_1$, and $A_2$: $[\![x \mapsto y \mathbin{-\!\!\circledast} \mathbf{ls}(x,z)]\!] \subseteq D \sqcap A_1 \sqcap A_2$. Fig. 6(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. 6(b) is split in Fig. 6(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 $link[d,n,z]$ true.

Fig. 6(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 $(\cdot)\not\downarrow^{d}$ in §4, serves as an abstract method for quantifier elimination.

Note that Fig. 6(e) represents an acyclic linked-list from $y$ to $z$ with $x \neq y$, which is one of the models that satisfies $x \mapsto y \mathbin{-\!\!\circledast} \mathbf{ls}(x,z)$.



(a)

(b)

(c)

(d)

(e)

Figure 6: **Some of the structures that arise in the meet operation used to evaluate $x \mapsto y \mathbin{-\!\!\circledast} \mathbf{ls}(x,z)$.**