BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2022
http://excel.fit.vutbr.cz

A Decision Procedure For Strong-Separation Logic

Tomáš Dacík*

Abstract
Separation logic (SL) is one of the most successful tools for verification of programs that manipulate dynamically allocated memory. Its expressive power comes at a cost of undecidability when several of its features, namely negations, inductive predicates describing data structures and separating implications are combined. To circumvent this problem, the recently introduced strong-separation logic (SSL) uses a stricter definition of the semantics, making it decidable, while remaining suitable for verification. However, there is currently no implementation of a decision procedure for SSL. In this work, we propose a decision procedure for SSL based on a translation to first-order formulae that can be later solved by a specialized solver. Our preliminary experimental results show that our approach can effectively solve formulae obtained from verification tools based on SL and also outperform existing decision procedure based on similar translation.

Keywords: Logic — Separation logic — Decision procedure

Supplementary Material: Downloadable Code

*xdacik00@stud.fit.vut.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

In recent years, logic proved to be a very useful tool in many approaches to formal verification of software and hardware. Formulae in various logics can be used not only for specification of the correct behavior but also as a backend technology in tools that attempt to verify the specification. Prominent examples are propositional logic and satisfiability modulo first-order theories (SMT). SMT solvers underwent rapid development and can be used, for example, to discharge preconditions generated by deductive verification tools or for automatic generation of test cases for real life programs.

Another category are non-classical logics that can express some aspects of computer programs. An example is separation logic (SL) [1] that is the subject of this work. In general, it can express properties about shared resources and their disjointness while allowing a modular reasoning. In the most common setting, the shared resource is computer memory – SL can express properties such as that a heap contains two disjoint null-terminated linked lists from memory locations x and y. A common problem of heap properties is *aliasing* – can it be the case that x and y alias, and heap does therefore contain a single list only?

Separation logic solves this problem by introducing two spatial connectives – the *separating conjunction* (denoted by $*$, and pronounced "and separately") and the *separating implication* (denoted as $-\!\!*$, and often called "magic wand"). Informally, formula $\varphi * \psi$ states that the heap can be split into two disjoint parts such that the formula φ is satisfied in the first and ψ is satisfied in the second. For example, the formula $x \mapsto z * y \mapsto z$ informally states that the heap contains two pointers, and the semantics of the separating conjunction makes sure that their source locations x and y are distinct.

Another ingredient of SL are inductive predicates describing data structures such as lists or trees. Some variants of SL allow arbitrary user-defined predicates, but in this work, we assume a single fixed inductive predicate $\mathsf{ls}(x,y)$ representing acyclic single-linked lists. As shown by Demri [2], a separation logic that combines all previously mentioned components and boolean connectives is under the classical semantics undecidable. Most verification tools therefore do not work with the magic wand which is, however, necessary to succinctly express even such a trivial property

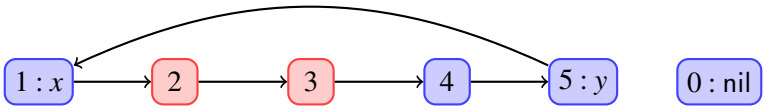


Figure 1. A stack-heap model with $s = \{x \mapsto 1, y \mapsto 5, \mathsf{nil} \mapsto 0\}$ and $h = \{1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 4, 4 \mapsto 5, 5 \mapsto 1\}$. The model is equivalent to a model where the red-marked locations 2 and 3 (more precisely any two of 2, 3 and 4) are removed (there is no SSL formula that can distinguish those models). It holds $(s,h) \models \mathsf{ls}(x,y) * y \mapsto x$.

as "*location x is allocated*". The magic wand also naturally appears in so-called *bi-abductive analysis* [3], where it is usually only approximated.

To tackle this, Pagel and Zuleger introduced a so-called *strong-separating* semantics under which the mentioned fragment becomes decidable. In [4], they introduce a concept of *abstract memory states* (a finite abstraction over possibly infinite sets of models) and a decision procedure based on their enumeration that runs in polynomial space. However, there is no implementation of their decision procedure. This work is focused on design and implementation of another decision procedure for SSL. Rather than performing an enumeration, we perform a translation to first-order formulae that can be later solved by an SMT solver.

Related work. A translation of SL to SMT was first time proposed in [5] and [6] for a fragment with lists and trees, respectively. The approach uses an intermediate logic that is later translated to SMT. Our approach is, however, more close to the approach proposed in [7] which establishes a small-model property for separation logic with data predicates and performs a direct translation implemented in a tool called SLOTH. A similar translation was designed in [8] for SSL with data but not implemented. All those works consider only such fragments of SL where boolean connectives cannot appear under separating conjunction, and the magic wand cannot appear at all. A fragment with the magic wand, arbitrary combinations of boolean and spatial connectives, but no inductive predicates is supported by the SMT solver CVC4 for which it implements a specialized theory solver [9]. There also exist solvers for SL with general inductive predicates but their discussion is out of the scope of this paper.

2. Strong-Separation Logic

In this section, we introduce the notation used in this paper and formally introduce strong-separation logic. We also briefly discuss its properties that are essential for an effective implementation of our decision procedure. The section is based on [4] where those properties are described in details and formally proved.

2.1 Preliminaries

We use predicate $\mathsf{distinct}(x_1,\ldots,x_n)$ to denote that all x_i are pairwise different. We write $f : X \rightharpoonup Y$ to denote a partial function from X to Y, $\mathsf{dom}(f)$ to denote its domain, and $f(x) = \bot$ if f is undefined for x.

Let $G = (V, \rightarrow)$ be an oriented graph. A path π is a sequence of vertices $\langle x_1, x_2, \ldots, x_n \rangle$ such that for all $1 \leq i < n$, it holds that $x_i \rightarrow x_{i+1}$. A path π is *simple* if it does not contain any vertex more than once. All simple paths are therefore acyclic. The domain of a path π is the set $\mathsf{dom}(\pi) = \{x_1, x_2, \ldots, x_{n-1}\}$ and the length of the path is defined as $|\pi| = |\mathsf{dom}(\pi)|$.

2.2 Syntax

Let **Var** be an infinite set of variables with a distinguished variable $\mathsf{nil} \in$ **Var**. We consider a quantifier-free fragment of separation logic given by the following grammar where $x, y \in$ **Var**:

$$\begin{aligned} \varphi_{atom} &::= x = y \mid x \neq y \mid x \mapsto y \mid \mathsf{ls}(x,y) \\ \varphi &::= \varphi_{atom} \mid \varphi * \varphi \mid \varphi \mathrel{-\!\!\circledast} \varphi \\ &\quad \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \wedge_{\neg} \varphi \mid \neg\varphi \end{aligned}$$

An atomic formula φ_{atom} is either an equality $x = y$, a disequality $x \neq y$, a points-to assertion $x \mapsto y$, or a list-segment predicate $\mathsf{ls}(x,y)$. Full SSL is obtained by allowing an arbitrary combination of boolean connectives, separating conjunction and septraction ($\mathrel{-\!\!\circledast}$), which is a dual connective to the magic wand. In this work, we will focus on a *positive fragment* of SSL that does not allow negation, and requires that each septraction has a positive polarity (i.e., that it lies under an even number of negations). This intuitively means that septractions are used only for an existential quantification over heaps. In the positive fragment, the so-called *guarded negation* $\varphi \wedge_{\neg} \psi$[1] can still be used. We use $\mathsf{vars}(\varphi)$ to denote the set of all variables of φ.

2.3 Stack-Heap Models

We assume an infinite set of heap memory locations denoted as **Loc**. SSL is interpreted over *stack-heap models* (s,h), where *stack* is a finite partial function $s :$ **Var** $\rightharpoonup$ **Loc** and *heap* is a finite partial function

[1] A formula $\varphi \wedge_{\neg} \psi$ is semantically equivalent to the formula $\varphi \wedge \neg\psi$, but we rather treat it as a standalone binary connective.

$$
\begin{array}{ll}
(s,h) \models x = y & \text{iff } s(x) = s(y) \text{ and } \mathsf{dom}(h) = \emptyset \\
(s,h) \models x \neq y & \text{iff } s(x) \neq s(y) \text{ and } \mathsf{dom}(h) = \emptyset \\
(s,h) \models x \mapsto y & \text{iff } h = \{s(x) \mapsto s(y)\} \\
(s,h) \models \mathsf{ls}(x,y) & \text{iff } \mathsf{dom}(h) = \emptyset \text{ and } s(x) = s(y) \text{ or there exist } n \geq 1, l_0, ..., l_n \text{ such that} \\
 & \quad \mathsf{distinct}(\ell_0, ..., \ell_n) \text{ and } h = \{\ell_0 \mapsto \ell_1, ..., \ell_{n-1} \mapsto \ell_n\} \text{ and } s(x) = \ell_0, s(y) = \ell_n \\
 & \\
(s,h) \models \varphi_1 \wedge \varphi_2 & \text{iff } (s,h) \models \varphi_1 \text{ and } (s,h) \models \varphi_2 \\
(s,h) \models \varphi_1 \wedge_{\neg} \varphi_2 & \text{iff } (s,h) \models \varphi_1 \text{ and } (s,h) \not\models \varphi_2 \\
(s,h) \models \varphi_1 \vee \varphi_2 & \text{iff } (s,h) \models \varphi_1 \text{ or } (s,h) \models \varphi_2 \\
(s,h) \models \neg\varphi & \text{iff } (s,h) \not\models \varphi \\
 & \\
(s,h) \models \varphi_1 * \varphi_2 & \text{iff } \exists h_1, h_2.\ h = h_1 \uplus^s h_2 \text{ and } (s,h_1) \models \varphi_1 \text{ and } (s,h_2) \models \varphi_2 \\
(s,h) \models \varphi_1 \mathrel{-\!\!\circledast} \varphi_2 & \text{iff } \exists h_1.\ (s,h_1) \models \varphi_1, h \uplus^s h_1 \neq \bot \text{ and } (s, h \uplus^s h_1) \models \varphi_2
\end{array}
$$

Figure 2. The semantics of strong-separation logic.

$h : \mathbf{Loc} \rightharpoonup \mathbf{Loc}$. We further require that the stack of each model maps nil to some location that is not in the domain of its heap, i.e., $s(\mathsf{nil}) \neq \bot$ and $s(\mathsf{nil}) \notin \mathsf{dom}(h)$.

As demonstrated in Figure 1, a stack-heap model (s,h) can be represented as an oriented graph where vertices are heap locations and edges represent heap pointers. To capture also the stack, each vertex is labeled by variables that are mapped to it. In the rest of the text, we identify the model and its graph representation. We use $\mathsf{locs}(h) = \mathsf{dom}(h) \cup \mathsf{img}(h)$ to denote all heap locations of the model. A location ℓ is called *allocated* if $\ell \in \mathsf{dom}(h)$. We further call ℓ *named* if at least one variable is mapped to it, and *anonymous* otherwise.

2.4 Semantics

The definition of the semantics of separation logic is based on a notion of *disjointness* of two heaps. In classical SL, heaps h_1 and h_2 are disjoint if their domains are disjoint. The strong-separating semantics adds another condition that locations shared by both heaps have to be named. The *disjoint union* of heaps wrt. a stack s is then defined as:

$$
h_1 \uplus^s h_2 = \begin{cases} h_1 \cup h_2 & \text{if } \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset \\ & \wedge\ \mathsf{locs}(h_1) \cap \mathsf{locs}(h_2) \subseteq \mathsf{img}(s) \\ \bot & \text{otherwise} \end{cases}
$$

Now, we can define the semantics of SSL as in Figure 2. We use the so-called precise semantics under which (dis)equalites are satisfied on empty heaps only. Similarly, points-to assertions are satisfied only by a single pointer from x to y. A list-segment predicate $\mathsf{ls}(x,y)$ is satisfied either by an empty heap if $s(x) = s(y)$ or by a non-empty sequence of pointers from x to y such that all locations in the sequence are distinct. Lists therefore cannot be cyclic [2] or lasso-shaped.

The semantics of boolean connectives is defined in the usual way. The semantics of spatial connectives is defined using the operator $\uplus^s$. We can further define the empty heap predicate, the magic wand connective, and classical boolean constants as syntactic sugar:

$$
\begin{array}{ll}
\mathsf{emp} \triangleq \mathsf{nil} = \mathsf{nil} & \mathsf{false} \triangleq \mathsf{emp} \wedge_{\neg} \mathsf{emp} \\
\varphi \mathrel{-\!\!*} \psi \triangleq \neg(\varphi \mathrel{-\!\!\circledast} \neg\psi) & \mathsf{true} \triangleq \neg\mathsf{false}
\end{array}
$$

Example 2.1. Let $\varphi_1 \triangleq x \mapsto y * x \mapsto z$. The formula φ_1 is unsatisfiable because it requires the location x to be allocated in both sub-heaps, which is forbidden by the semantics of the separating conjunction. On the other hand, the formula $\varphi_2 \triangleq x \mapsto y * z \mapsto y$ is satisfiable. Notice that φ_2 implicitly asserts that locations represented by x and z differs.

Example 2.2. Let $\varphi_3 \triangleq (x \mapsto \mathsf{nil}) \mathrel{-\!\!\circledast} \mathsf{true}$. The formula is satisfied by models that can be extended by a pointer from x to nil, i.e., by models that do not allocate x. This formula can be also expressed using the magic wand as $\neg(x \mapsto \mathsf{nil} \mathrel{-\!\!*} \mathsf{false})$ but cannot be succinctly expressed without the magic wand or septraction combined with negation.

[2] A cyclic list can be defined by the formula $\mathsf{ls}(x,y) * y \mapsto x$.

Unlike in the classical SL, satisfiability and entailment are defined wrt. a set of variables $\mathbf{x} \subseteq \mathbf{Var}$. A formula φ with $\mathsf{vars}(\varphi) \subseteq \mathbf{x}$ is satisfiable wrt. $\mathbf{x}$ if there exists a model $(s,h) \models \varphi$ such that $\mathsf{dom}(s) = \mathbf{x}$. A formula φ entails a formula ψ wrt. $\mathbf{x}$, written $\varphi \models_{\mathbf{x}} \psi$, if each model of φ with $\mathsf{dom}(s) = \mathbf{x}$ is also a model of ψ. Validity of an entailment can be reduced to unsatisfiability of formula $\varphi \wedge_{\neg} \psi$ using classical boolean equivalences.

2.5 Small Models and Footprints

In the positive fragment, semantics of SL and SSL coincide and one can therefore consider SSL a "backward compatible" extension of positive SL. This also means that for a satisfiable positive formula, it is always sufficient to set $\mathbf{x} = \mathsf{vars}(\varphi)$ and we can therefore ignore the $\mathbf{x}$ component of the input when dealing with positive formulae.

Moreover, a satisfiable positive formula φ has a model of size at most $2 \cdot \mathsf{vars}(\varphi)$. The idea behind is that SSL cannot speak about lengths of lists greater than 2 without using additional variables. This is sketched in Figure 1 where two red locations can be removed to obtain an equivalent model. In our decision procedure, this can be utilized to work with a finite domain of locations.

The semantics of spatial operators involves quantification over sub-heaps whose translation can be potentially very expensive. In [5] and [8] this problem is prevented by establishing a so-called *unique footprint property*. It states that for each positive formula φ and each fixed model (s,h), the following holds: if $(s,h) \models \varphi * \mathsf{true}$, then there exists the unique set F such that $(s,h|_F) \models \varphi$. This set is called as the *unique footprint* of φ in (s,h). When translating a separating conjunction, it is not necessary to consider all sub-heaps, but only sub-heaps induced by the footprints of its arguments (if the footprint is unique, than its induced heap is also unique). For the example, the model in Figure 1 satisfies a formula $\mathsf{ls}(y,x) * \mathsf{true}$ and the footprint of a subformula $\mathsf{ls}(y,x)$ is determined as a domain of simple path (such a path is always unique) from y to x which is $\{5\}$.

The unique footprint property does not longer holds when a disjunction is added to the logic, but if fortunately still holds that there are only polynomially many footprints that need to be verified to determine satisfiability of a separating conjunction.

3. Decision Procedure for Positive SSL

In this section, we propose a new decision procedure for the positive SSL. Rather than performing a custom enumeration as in [4], we try to leverage capabilities of modern SMT solvers and propose a translation to SMT inspired by [8]. We extend a fragment that can be translated by considering septractions and arbitrary combination of boolean and spatial operators. We also have an extension for the full SSL that needs an additional treatment of quantifiers, but do not discuss it in this paper for space reasons. As we already mentioned, semantics of SL and SSL correspond on the positive fragments. Our decision procedure is therefore applicable for SL as well. Our work is therefore also a contribution in the context of SL that, despite theoretical results, still lacks tools fully supporting boolean operators. This problem was recently addressed in [10] where the authors need a support for disjunction when translating quantitative separation logic into the classical one. Finally, we also propose several ways to optimize encoding of list-segment predicates compared to [8].

3.1 Translation to SMT

The main idea of the decision procedure is to translate an input formula to an equisatisfiable formula in a combination of theories of finite sets and arrays – the sets are used to encode the domains of heaps, while arrays are used to encode their mappings. In the rest of this section, we fix an input formula φ and its location bound n. By default, we can use the general bound $2 \cdot \mathsf{vars}(\varphi)$, later in this section we will show how this bound can be improved.

Utilizing the small model property, we can restrict the infinite domain of locations **Loc** to its finite subset $\mathbf{L} = \{\ell_1, \ell_2, ..., \ell_n\}$ consisting of n distinct location constants. Because SSL formulae cannot distinguish isomorphic models, it does not matter which particular subset we choose. In the translation, this is ensured by the following formula[3]:

$$\varphi_n^{card} \triangleq \exists \ell_1, ..., \ell_n.\, \mathsf{distinct}(\ell_1, ..., \ell_n) \wedge \forall x. \bigvee_{1 \leq i \leq n} x = \ell_i$$

Because all functions are total in SMT, we need to encode partial heap functions by two components – an array h that encodes the mapping and a set F that encodes the domain. The stack image of a variable x is encoded simply by a constant symbol x. A model $\mathcal{M}$ of the translated formula can then be used to obtain a

[3] In many-sorted first-order logic used by SMT solvers, this constraint can be replaced by declaring $\mathbf{L}$ as a finite sort of cardinality n.

- The translation for an atomic formula φ:

$$\begin{array}{lll}
\mathrm{T}_n(x = y): & \widetilde{\varphi} \triangleq x = y & \mathcal{A} \triangleq F_\varphi = \emptyset \\
\mathrm{T}_n(x \neq y): & \widetilde{\varphi} \triangleq x \neq y & \mathcal{A} \triangleq F_\varphi = \emptyset \\
\mathrm{T}_n(x \mapsto y): & \widetilde{\varphi} \triangleq h[x] = y & \mathcal{A} \triangleq F_\varphi = \{x\} \\
\mathrm{T}_n(\mathsf{ls}(x,y)): & \widetilde{\varphi} \triangleq \mathsf{reach}_n(x,y) & \mathcal{A} \triangleq \mathsf{path}_n(F_\varphi, x, y)
\end{array}$$

- The recursive translation of a binary connective $\psi_1 \bowtie \psi_2$. We use $(\widetilde{\psi_i}, \mathcal{A}_i)$ and F_i to denote the translation of ψ_i and the footprint of ψ_i, respectively:

$$\begin{array}{lll}
\mathrm{T}_n(\psi_1 \wedge \psi_2): & \widetilde{\varphi} \triangleq \widetilde{\psi_1} \wedge \widetilde{\psi_2} \wedge F_1 = F_2 & \mathcal{A} \triangleq \mathcal{A}_1 \wedge \mathcal{A}_2 \wedge F = F_1 \\
\mathrm{T}_n(\psi_1 \wedge_\neg \psi_2): & \widetilde{\varphi} \triangleq \widetilde{\psi_1} \wedge (\neg\widetilde{\psi_2} \vee F_1 \neq F_2) & \mathcal{A} \triangleq \mathcal{A}_1 \wedge \mathcal{A}_2 \wedge F = F_1 \\
\mathrm{T}_n(\psi_1 \vee \psi_2): & \widetilde{\varphi} \triangleq (\widetilde{\psi_1} \wedge F = F_1) \vee (\widetilde{\psi_2} \wedge F = F_2) & \mathcal{A} \triangleq \mathcal{A}_1 \wedge \mathcal{A}_2 \wedge (F = F_1 \vee F = F_2) \\
 & & \\
\mathrm{T}_n(\psi_1 * \psi_2): & \widetilde{\varphi} \triangleq \widetilde{\psi_1} \wedge \widetilde{\psi_2} \wedge F_1 \cap F_2 = \emptyset & \mathcal{A} \triangleq \mathcal{A}_1 \wedge \mathcal{A}_2 \wedge F = F_1 \cup F_2 \\
\mathrm{T}_n(\psi_1 \mathrel{-\!\circledast} \psi_2): & \widetilde{\varphi} \triangleq \widetilde{\psi_1}[h_\varphi/h] \wedge \widetilde{\psi_2}[h_\varphi/h] \wedge F_1 \subseteq F_2 & \mathcal{A} \triangleq \mathcal{A}_1[h_\varphi/h] \wedge \mathcal{A}_2[h_\varphi/h] \wedge F = F_2 \setminus F_1 \\
 & & \wedge\ \forall x \in F.\ h[x] = h_\varphi[x]
\end{array}$$

Figure 3. The translation of an SSL formula to first-order logic.

stack-heap model of the input formula as follow:

$$s(x) = \begin{cases} x^{\mathcal{M}} & \text{if } x \in \mathbf{x} \\ \bot & \text{otherwise} \end{cases}$$

$$h(\ell) = \begin{cases} h^{\mathcal{M}}[\ell] & \text{if } \ell \in F^{\mathcal{M}} \\ \bot & \text{otherwise} \end{cases}$$

The translation is represented by a function $T(\varphi, n)$ that takes an input formula φ and calls an auxiliary function $T_n(\varphi)$ defined in Figure 3 that performs a recursive translation with the given bound n. In each step, the translation $T_n(\varphi)$ produces a pair $(\widetilde{\varphi}, \mathcal{A})$. The first component is called the *semantics* and it represents constraints on the stack and heap implied by the formula, such as two variables are equal or the heap contains a pointer. Those constraints may rely on auxiliary symbols introduced during the translation. The intended meaning of those auxiliary symbols is ensured by the second component $\mathcal{A}$ called *axioms*. The main role of the axioms is to ensure that a symbol F_ψ for each subformula ψ is interpreted as its footprint. After the translation is finished, those components are combined into a final formula – a conjunction of the axioms, the semantics and three additional constraints that (i) connects the footprint of whole φ to the domain of then heap, (ii) defines the cardinality of the location domain and (iii) ensures that nil is not allocated:

$$\begin{aligned} T(\varphi, n) \triangleq\ & \text{let } (\widetilde{\varphi}, \mathcal{A}) = T_n(\varphi) \text{ in} \\ & \widetilde{\varphi} \wedge \mathcal{A} \wedge F = F_\varphi \wedge \varphi_n^{card} \wedge \mathsf{nil} \notin F \end{aligned}$$

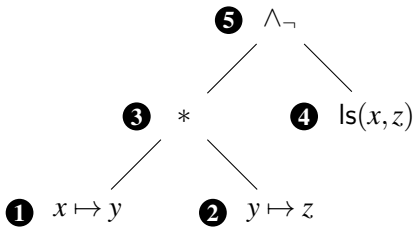
As already mentioned, the *axioms* make sure that all footprints are defined correctly (therefore they are computed separately so they are never negated) and the *semantics* checks whether a formula holds or not. Both equality and dis-equality can be satisfied only on the empty footprint and the translation of their semantics is straightforward. Similarly for a points-to predicate $x \mapsto y$ that can be satisfied only on an a singleton footprint $\{x\}$. A list-segment predicate $\mathsf{ls}(x,y)$ can be satisfied only on a simple path from x to y if there exists such a path. How those properties are translated is explained in Section 3.2.

The translation of the conjunction expresses that both of its operands are satisfied on an equal footprint. Similarly, the translation of the guarded negation expresses that the first operand is satisfied while the second is not – either its semantics does not hold or its footprint does not match. In the case of the disjunction, at least one of its operands has to be satisfied on the correct footprint. Notice that in this case, we cannot say which footprint will be used and an SMT solver therefore has to guess the footprint of the disjunction. Finally, the separating conjunction requires that both of its operands are satisfied on disjoint footprints, the footprint of the separating conjunction itself is then the union of footprints of its operands.

The translation of the septraction is more involved. For each septraction $\varphi \triangleq \psi_1 \mathrel{-\!\circledast} \psi_2$ we need to introduce a fresh heap h_φ that is used to find a model of ψ_1 and ψ_2 such that a footprint of ψ_1 is contained in a

Semantics:

$$\widetilde{\varphi}_1 \triangleq h[x] = y$$
$$\widetilde{\varphi}_2 \triangleq h[y] = z$$
$$\widetilde{\varphi}_3 \triangleq \widetilde{\varphi}_1 \wedge \widetilde{\varphi}_2 \wedge F_1 \cap F_2 = \emptyset$$
$$\widetilde{\varphi}_4 \triangleq \mathsf{reach}_2(x,z)$$
$$\widetilde{\varphi}_5 \triangleq \widetilde{\varphi}_3 \wedge (\neg\widetilde{\varphi}_4 \vee F_3 \neq F_4)$$



Axioms:

$$\mathcal{A}_1 \triangleq F_1 = \{x\}$$
$$\mathcal{A}_2 \triangleq F_2 = \{y\}$$
$$\mathcal{A}_3 \triangleq \mathcal{A}_1 \wedge \mathcal{A}_2 \wedge F_3 = F_1 \cup F_2$$
$$\mathcal{A}_4 \triangleq \mathsf{path}_2(F_4,x,z)$$
$$\mathcal{A}_5 \triangleq \mathcal{A}_3 \wedge \mathcal{A}_4 \wedge F_5 = F_3$$

Auxiliary predicates:

$$\mathsf{reach}_2(x,z) \triangleq x = z \vee h[x] = z \vee h^2[x] = z$$
$$\mathsf{path}_2(F_4,x,z) \triangleq \big(\neg\mathsf{reach}_2(x,z) \wedge F_4 = \emptyset\big) \quad \text{(no path)}$$
$$\vee \ \big(x = z \wedge F_4 = \emptyset \wedge z \notin F_4\big) \quad \text{(length 0)}$$
$$\vee \ \big(h[x] = z \wedge F_4 = \{x\} \wedge z \notin F_4\big) \quad \text{(length 1)}$$
$$\vee \ \big(h^2[x] = z \wedge F_4 = \{x, h[x]\} \wedge z \notin F_4\big) \quad \text{(length 2)}$$

Final formula:

$$\mathsf{T}(\varphi,2) \triangleq \widetilde{\varphi}_5 \wedge \mathcal{A}_5 \wedge F = F_5 \wedge \mathsf{nil} \notin F \wedge \varphi_5^{card}$$

Figure 4. An example of the translation for a formula $\varphi \triangleq (x \mapsto y * y \mapsto z) \wedge_{\neg} \mathsf{ls}(x,z)$.

footprint ψ_2 (i.e., that their difference is well-defined). The introduction of the fresh heap is ensured by substituting h_φ for the currently used heap h in all previously computed semantics constraints and axioms. The new axiom makes sure that h and h_φ equals on the footprint of the septraction which is the difference of footprints of its operands. In other words, an SMT solver has to find a model of operands of the septraction such that the first one can be "septracted" from the second one – the resulting heap is then propagated as a model of the septraction itself.

3.2 Translation of List-Segment Predicates

To translate list-segment predicates, we use the fact that edges are defined by a total function in SMT encoding. Therefore, there exists a path from x to y iff there exists $0 \leq i < n$ such that $h^i[x] = y$ (where an expression $h^i[\cdot]$ denotes an i-times iterated read from the array h and $h^0[x] = x$). This can be directly turned into a definition of bounded reachability parametrised by n:

$$\mathsf{reach}^i_n(x,y) \triangleq h^i[x] = y$$
$$\mathsf{reach}_n(x,y) \triangleq \bigvee_{0 \leq i < n} \mathsf{reach}^i_n(x,y)$$

To axiomatize footprints of lists, we further define a predicate $\mathsf{reachable}^{<i}_n(L,x)$ that asserts that L is a set of all locations reachable from x in less than i steps. Clearly, L contains all locations $h^j[x]$ where $j < i$:

$$\mathsf{reachable}^{<i}_n(L,x) \triangleq \begin{cases} L = \emptyset & \text{if } i = 0 \\ L = \{x, \ldots, h^{i-1}[x]\} & \text{if } i > 0 \end{cases}$$

Finally, we will define a predicate $\mathsf{path}_n(F,x,y)$ that ensures that a footprint F is interpreted as a domain of simple path from x to y:

$$\mathsf{path}_n(F,x,y) \triangleq \bigvee_{0 \leq i < n} \Big(\mathsf{reach}^i_n(x,y) \wedge \mathsf{reachable}^{<i}_n(F,x) \wedge y \notin F\Big) \vee \Big(\neg\mathsf{reach}_n(x,y) \wedge F = \emptyset\Big)$$

The main idea is to make a case distinction over all possible lengths of paths. We have to also make sure that the path is simple, this is ensured by adding a constraint $y \notin F$ to all clauses – every path that is not simple has to be an extension of the unique simple path and therefore its domain will contain y. The last clause of the definition handles the case when there is no path from x to y, in such a case semantics of the list-segment predicate is not satisfied and we do not care about interpretation of its footprint and simply assert it to be an empty set.

Complexity. If the input formula is positive, then its translation is quantifier-free (the only quantifier in the translation of the septraction can be rewritten using enumeration) and has a polynomial size (precisely

$\mathcal{O}(n^3)$). Because the theory of sets can be reduced to the extended theory of arrays (a set is represented by an array mapping elements to boolean values, and operations over sets are translated using array combinators that point-wise apply some function to a tuple of arrays) which is decidable in **NP**, our decision procedure also runs in **NP** for positive formulae.

3.3 Example of Translation

To give a better intuition how our translation works, we will demonstrate it on checking validity of an entailment $x \mapsto y * y \mapsto z \models \mathsf{ls}(x,z)$. Note that the entailment is not valid because its left-hand side can be satisfied with a cycle of two pointers which does not satisfy acyclicity required by the list-segment predicate. The entailment can be reduced to checking whether a formula $\varphi \triangleq (x \mapsto y * y \mapsto z) \wedge_{\neg} \mathsf{ls}(x,z)$ is unsatisfiable.

The whole translation is demonstrated in Figure 4. Because we distinguish sub-formulae by their identifiers, the figure also shows how identifiers are assigned to each subformula represented by a node in AST of the formula. For simplicity, we assume the precise location bound for a minimal model which is 2.

Notice in particular the definition of the predicate path_2. In a model containing pointers $x \mapsto y$ and $y \mapsto z$ such that $s(x) = s(z)$ there would be multiple paths from x to z, e.g., ε or $\langle x,y,z \rangle$. The additional constraint that $z \notin F_4$ ensures that only the clause that corresponds to the simple path (namely the clause named *length 0*) is satisfied. Therefore, the top-level guarded negation is satisfied because footprint F_3 (which is always equal to $\{x,y\}$) is not equal to the footprint F_4 (which is empty).

3.4 Bounds on Number of Locations and List Lengths

The size of the translated formula is dominated by encoding the list-segment predicates where the size of their encoding depends on the considered number of locations. The bound established in Section 2 is not tight in many cases and can be significantly improved. However, in many cases, this is still not enough, and therefore we also compute a so-called *list bound* for each list-segment predicate. This bound is an interval $[m,n]$ such that it is enough to consider lists of length from the interval only.

We first compute three relations, that for each pair of variables (x,y), tell us which equalities, disequalities and pointers hold in every model of φ. We denote those must-relations by $x \circleddash y$, $x \circlearrowright y$, and $x \circleddash\!\!\rightarrow y$ respectively. The location bound can be refined by replacing the set of variables $\mathbf{x}$ by its partition by the relation $\circleddash$. We can further subtract one for each equivalence class x that is guaranteed to have a named successor:

$$\mathsf{bound}(\varphi) = 2 \cdot |\mathbf{x}/_{\circleddash}| - |\{x \in \mathbf{x}/_{\circleddash} \mid \exists y.\ x \circleddash\!\!\rightarrow y\}|$$

The list bound of the predicate $\mathsf{ls}(x,y)$ can be computed using a must-points-to relation and sets of must-allocated variables that can be computed for each subformula of φ. For space reasons, we describe our approach on two examples only.

Example 3.1. The entailment $x \mapsto y * y \mapsto z \models \mathsf{ls}(x,z)$ can be reduced to unsatisfiability of the following formula $(x \mapsto y * y \mapsto z) \wedge_{\neg} \mathsf{ls}(x,z)$. From this formula, we can derive that each of its models contains pointers from x to y and from y to z, moreover, locations x and y will always be distinct. However, we cannot derive any must-equality and the location bound is therefore $n = 2 \cdot |\mathsf{vars}(\varphi)| - |\{x,y\}| = 4$. The list bound for predicate $\mathsf{ls}(x,z)$ is $[2,2]$ because must pointers $x \circleddash\!\!\rightarrow y$ and $y \circleddash\!\!\rightarrow z$ fix the shape of the list to the exact length 2.

Example 3.2. For the formula $\varphi \triangleq \mathsf{ls}(x,y) * \mathsf{ls}(y,z) * y \neq z$ we can use information about must-allocated variables to compute the list bounds for the predicate $\mathsf{ls}(x,y)$ to be $\mathsf{bound}(\varphi) - 1$. This is because the location y is surely allocated in list $\mathsf{ls}(y,z)$, and therefore cannot be allocated in the second list $\mathsf{ls}(x,y)$.

4. Implementation and Experiments

The proposed decision procedure is implemented in a prototype tool called ASTRAL. ASTRAL is written in the OCaml programming language and publicly available under MIT license[4]. It currently uses SMT solver Z3 [11] as the backend.

We have evaluated ASTRAL on a benchmark from SL-COMP[5], a competition of solvers for separation logic. We performed experiments on two categories, namely QF_SHLS_SAT and QF_SHLST_ENTL, i.e., satisfiability and entailment of quantifier-free symbolic heaps with lists, respectively. In our logic, a formula ψ is a *symbolic heap* if it has the form $* \psi_i$[6] where each ψ_i is an atomic formula. Since both categories are contained in the positive fragment, their semantics correspond to the strong-separating semantics. All experiments were conducted on a machine with 2.5GHz Intel Core i5-7300HQ processor and 16 GB RAM, running Ubuntu 18.04. The time limit was set to 60 seconds in all experiments.

[4] https://github.com/TDacik/Astral

[5] https://sl-comp.github.io/

[6] In classical semantics, symbolic heaps have form $\Pi \wedge \Sigma$ where Π is conjunction of (dis)equalities and Σ is separating conjunction of spatial predicates. This definition can be easily converted to ours by replacing all conjunctions with separation conjunctions.

Table 1. The summary of experiments on SL-COMP benchmarks. All times are in seconds (the timeout is 60 s).

		ASTRAL			ASTERIX		
Category	Formulae	Total time	Max. time	Timeouts	Total time	Max. time	Timeouts
QF_SHLS_SAT	110	11.92	0.57	0	0.4565	0.0057	0
QF_SHLS_ENTL (random)	210	758.15	-	78	0.8636	0.0058	0
QF_SHLS_ENTL (verif. conditions)	86	10.37	2.64	0	0.3362	0.0056	0

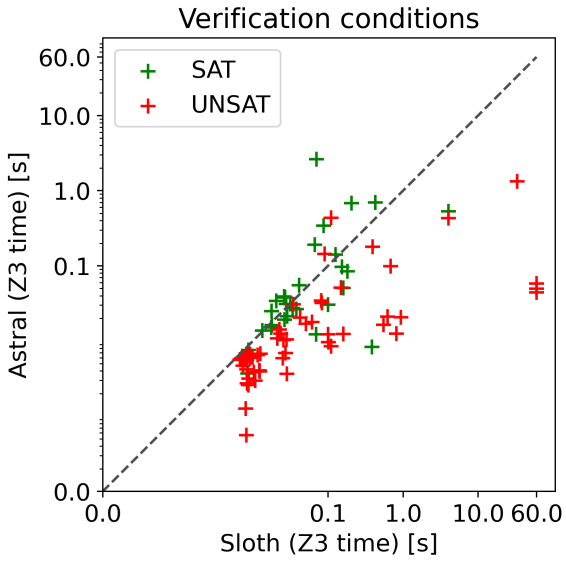


Figure 5. The comparison of ASTRAL and SLOTH on formulae from the category QF_SHLS_ENTL obtained from verification tools. Only the time of an SMT solver call is measured (timeout is 60 s).

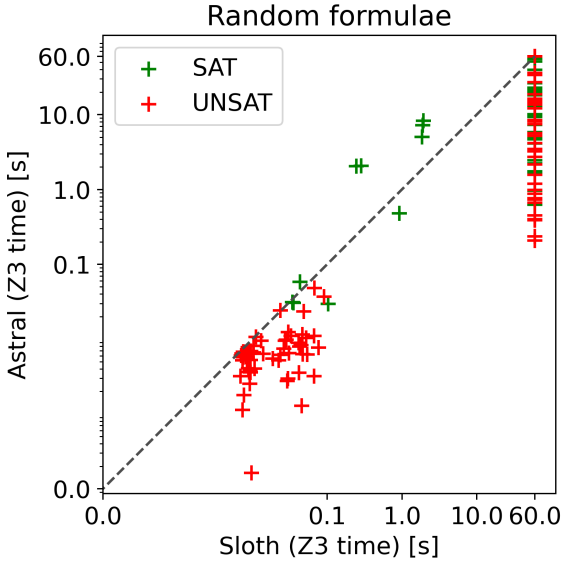


Figure 6. The comparison of ASTRAL and SLOTH on randomly-generated formulae from the category QF_SHLS_ENTL . Only the time of an SMT solver call is measured (timeout is 60 s).

Notice that the symbolic heap fragment is only a small subset of our logic, in fact, our decision procedure is even sub-optimal for this fragment, since both satisfiability and entailment can be solved in (deterministic) polynomial time [12]. Therefore, we do not expect to perform better than solvers specialized for symbolic heap fragment. However, we can still use this fragment to compare against other decision procedures and to evaluate the impact of our heuristics.

The results are summarized in Table 1 and compared with the tool ASTERIX [13] that won the last edition of SL-COMP in both categories. ASTRAL can solve all satisfiability problems very quickly, but not as fast as ASTERIX. However, notice that running times for single formulae are often so low that side aspects like implementation language can make a big difference. On entailment problems, ASTRAL times out roughly in one third of all cases. All of them are formulae that are randomly generated and contain many list predicates (up to 20). On a subset of this category representing real-life entailment problems (usually containing less than 5 list predicates) ASTRAL performs significantly better.

We also compared ASTRAL with SLOTH which uses a very similar translation but without some of our optimizations. The results are given in Figure 5 and Figure 6 for verification conditions and randomly-generated formulae, respectively. Because SLOTH is implemented in Python, we only compare the time consumed by the Z3 solver. When the overall time is considered, the results are even better for our solver because all translation optimizations such as bound computation takes only negligible time, while SLOTH usually spends several seconds on translation.

While differences for verification conditions are usually in fractions of seconds, there are three cases when SLOTH times out even for those simple formulae – this could be a complication for its employing in a verification tool that can make many of such queries when analysing a program. For randomly generated formulae, the difference is more visible. ASTRAL is able to solve 52 formulae that are too complicated for SLOTH. Our further experiments show that the computation of list-length bounds helps in many cases, but even when it is turned off, ASTRAL can still solve formulae that SLOTH cannot. This suggests that our translation of list-segment predicates can scale better with the growing number of list-segment predicates.

5. Conclusions

We presented a decision procedure for strong-separation logic based on a translation to SMT. While the idea of such a translation is not new, we have significantly extended the fragment that can be translated by adding a support for septractions and allowing arbitrary combination of boolean and spatial connectives. We have also developed a new method of decreasing sizes of encoded list-segment predicates. We performed experiments on two restricted fragments of our logic that show that our improvements indeed significantly help to reduce the running time. Our decision procedure cannot beat specialized solvers for those fragments, but can outperform existing translation-based solver. Our implementation can also effectively solve all real-life formulae originating from verification tools based on separation logic.

Our future work will focus on experiments on formulae of full SSL (including septractions and/or magic wands) and investigation of how SSL can be used in program verification, for the example, in bi-abductive analysis. Another research direction can focus on extending expressibility of our logic, e.g., by adding predicates describing more complex data structures.

Acknowledgements

I would like to thank my supervisor Tomáš Vojnar for his help. Further, I would like to thank Florian Zuleger and Adam Rogalewicz for consultations, and to all members of the VeriFIT research group for an inspiring working environment. I would also like to thank Bára and Jakub for making an awesome ASTRAL's logo.

The work was supported by the H2020 ECSEL project Arrowhead Tools.

References

[1] John Reynolds. Separation logic: A Logic for Shared Mutable Data Structures. *Proceedings - Symposium on Logic in Computer Science*, pages 55– 74, 02 2002.

[2] Stéphane Demri, Étienne Lozes, and Alessio Mansutti. The Effects of Adding Reachability Predicates in Propositional Separation Logic. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018*. Springer, 2018.

[3] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM*, 58:26, 01 2011.

[4] Jens Pagel and Florian Zuleger. Strong-separation logic. *ACM Trans. Program. Lang. Syst.*, nov 2021.

[5] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating Separation Logic Using SMT. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 773–789, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[6] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating Separation Logic with Trees and Data. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, page 711–728, Berlin, Heidelberg, 2014. Springer-Verlag.

[7] Jens Katelaan, Dejan Jovanovic, and Georg Weissenbacher. A Separation Logic with Data: Small Models and Automation. In *IJCAR*, 2018.

[8] Jens Pagel. Decision Procedures for Separation Logic: Beyond Symbolic Heaps, 2020.

[9] Andrew Reynolds, Radu Iosif, Cristina Serban, and Tim King. A Decision Procedure for Separation Logic in SMT. In *Automated Technology for Verification and Analysis 14th International Symposium (ATVA 2016)*, volume 9938, pages 244–261, Chiba, Japan, October 2016.

[10] Kevin Batz, Ira Fesefeldt, Marvin Jansen, Joost-Pieter Katoen, Florian Keßler, Christoph Matheja, and Thomas Noll. Foundations for Entailment Checking in Quantitative Separation Logic. In Ilya Sergey, editor, *Programming Languages and Systems*, pages 57–84, Cham, 2022. Springer International Publishing.

[11] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 4963:337–340, 04 2008.

[12] Byron Cook, Christoph Haase, Joël Ouaknine, Matthew Parkinson, and James Worrell. Tractable Reasoning in a Fragment of Separation Logic. In *Proceedings of the 22nd International Conference on Concurrency Theory*, CONCUR'11, page 235–249, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation Logic Modulo Theories. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8301, 03 2013.