Sloth: Separation Logic and Theories via Small Models

Jens Katelaan[1], Dejan Jovanović[2], and Georg Weissenbacher[1]

[1] TU Wien, Vienna, Austria
[2] SRI International

Abstract. We present Sloth, a solver for **s**eparation **lo**gic modulo **th**eory constraints specified in the separation logic $\mathbf{SL}^*_{\mathsf{data}}$, a propositional separation logic that we recently introduced in [14]. $\mathbf{SL}^*_{\mathsf{data}}$ admits NP decision procedures despite its high expressiveness; features of the logic include support for list and tree fragments, universal data constraints about both lists and trees, and Boolean closure of spatial formulas. Sloth solves $\mathbf{SL}^*_{\mathsf{data}}$ constraints via an SMT encoding of $\mathbf{SL}^*_{\mathsf{data}}$ that is based on the logic's small-model property. We argue that, while clearly still a work in progress, Sloth already demonstrates that $\mathbf{SL}^*_{\mathsf{data}}$ lends itself to the automation of nontrivial examples. These results complement the theoretical work presented in [14].

1 Introduction

Separation logics [28] enable concise, compositional specifications of programs with dynamic memory. Compositionality, based on the frame rule for the separating conjunction, has been the key to separation logic's impressive success, enabling the automated analysis of millions of lines of code [6,5]. Such scalability has, however, only been achieved for very restricted fragments of separation logic: Formulas without Boolean structure or data constraints (symbolic heaps) [3]. When restricted to individual pointers and lists, and when forbidding all quantification, satisfiability and entailment of symbolic heaps can even be decided in polynomial time [9]. But without Boolean structure, without the possibility to reason about data structures other than lists, and without the possibility to refer to the data stored in the data structures, the expressiveness of this polynomial-time decidable fragment is severely limited.

Unfortunately, adding features to the logic to increase its expressiveness quickly leads to intractability. For example, the full Boolean closure of even propositional separation logic immediately leads to Pspace hardness [7].

In [14], we proposed the propositional separation logic $\mathbf{SL}^*_{\mathsf{data}}$ that aims at striking the right balance between expressiveness and complexity: $\mathbf{SL}^*_{\mathsf{data}}$ is expressive enough to model interesting (shape and functional) properties of programs while retaining NP decidability. $\mathbf{SL}^*_{\mathsf{data}}$ has the following features.

1. Boolean closure of spatial formulas.[3] In addition to gains in expressiveness, this makes it possible to reduce entailment to satisfiability.
2. Support for list segments and tree fragments (partial trees). This enables specifications of programs that locally violate shape and/or data properties by asserting properties of multiple parts of a data structure separately.
3. Sufficiently expressive data constraints to capture many natural data-structure properties: Sortedness of lists, binary search trees, max heaps, etc.
4. Per-field allocation to allow easy extension of $\mathbf{SL}^*_{\mathsf{data}}$ to overlaid, nested and doubly-linked data structures. Such extensions remain future work, however.

We will illustrate all of these features by examples in Section 2. Despite this expressiveness, $\mathbf{SL}^*_{\mathsf{data}}$ has a small-model property that ensures NP decidability of the satisfiability problem of the logic. To the best of our knowledge, $\mathbf{SL}^*_{\mathsf{data}}$ is the first separation logic that combines all these features while remaining decidable in NP.

With the exception of the Grit logic [23], we are not aware of any separation logic that supports both reasoning about trees and support for data constraints while remaining decidable in NP. The complexity result for Grit [23] is, however, derived from the analysis of a specific encoding, which is based on local theory extensions and which formalized trees as overlaid reversed lists—a tree is an overlaid structure of paths that follow parent pointers from the leaves to the root. In contrast, we use a direct encoding of trees based on the small-model property, simplifying reasoning about trees. In addition, we also allow tree fragments, which are not supported by Grit.

We give a brief overview of other related work with a particular focus on decision procedures. Several other SAT and SMT encodings of heap logics have been proposed in the literature. These are either limited to lists [13,15,21,22], do not support any recursively defined structures, but the magic wand [26] and $\exists^*\forall^*$ quantifier prefixes [27]. Strand supports arbitrary MSO-definable structures and reduces only the data part of the assertions to SMT [17]. Decision procedures for various symbolic-heap fragments of separation logic without data were proposed in [3,11,12,18]; decidability of satisfiability and undecidability of entailment for general symbolic-heap separation logic where shown in [4] and [1], respectively. The satisfiability problem of a fragment with symbolic heaps and limited arithmetic was shown decidable in [16]. For a detailed discussion of the expressiveness and complexity of many variants and fragments of separation logic, see [10]. In addition, a myriad of semi-decision procedures for variants of separation logic have been proposed; see, for example [8,25].

The paper is structured as follows. In Section 2, we showcase $\mathbf{SL}^*_{\mathsf{data}}$ by presenting a number of examples. We omit a formal definition of the logic, which you can find in [14]. We continue with a high-level summary of our decision procedure for $\mathbf{SL}^*_{\mathsf{data}}$ in Section 3 before discussing the implementation of Sloth in Section 4. We present an evaluation of Sloth in Section 5 and conclude in Section 6.

[3] But no Boolean structure inside spatial formulas—i.e., below the separating conjunction—to avoid Pspace hardness

$$
\begin{array}{rll}
t & := \mathbf{null} \mid x \in \mathcal{X} & \\
A_{Spatial} & ::= t \to_f t \mid \mathsf{list}(t, \mathbf{s}) \mid \mathsf{tree}(t, \mathbf{s}) \mid \mathcal{F}_{\mathsf{loc}} \mid \mathcal{F}_{\mathsf{data}} & \text{Spatial atoms} \\
F_{Spatial} & ::= A_{Spatial} \mid F_{Spatial} * F_{Spatial} & \text{Spatial formulas} \\
F & ::= F_{Spatial} \mid \neg F \mid F \vee F \mid F \wedge F & \mathbf{SL}^*_{\mathsf{data}} \text{ formulas}
\end{array}
$$

Fig. 1: Simplified syntax of the separation logic $\mathbf{SL}^*_{\mathsf{data}}$. In the definition of $A_{Spatial}$, $\mathbf{s}$ denotes a (possibly empty) sequence of terms and $f \in \{\mathsf{n}, \mathsf{l}, \mathsf{r}, \mathsf{d}\}$.

2 $\mathbf{SL}^*_{\mathsf{data}}$ by Example

$\mathbf{SL}^*_{\mathsf{data}}$ is parametric in a location theory $\mathcal{T}_{\mathsf{loc}}$ (over a sort loc) and a data theory $\mathcal{T}_{\mathsf{data}}$ (over a sort data) with atomic formulas $\mathcal{F}_{\mathsf{loc}}$ and $\mathcal{F}_{\mathsf{data}}$, respectively. Figure 1 contains a slightly simplified definition of the syntax of $\mathbf{SL}^*_{\mathsf{data}}$. We assume a countable infinite set of (sorted) variables $\mathcal{X}$ and a dedicated constant $\mathbf{null}$ of sort loc. We denote with $\mathbf{s}$ a vector $\langle s_1, \ldots, s_n \rangle$ of variables from $\mathcal{X}$.

Formulas are Boolean combinations of *spatial formulas*. As usual, a spatial formula is a separating conjunction of *spatial atoms*. For $\mathbf{SL}^*_{\mathsf{data}}$, these are points-to assertions, list predicates, tree predicates, as well as atoms of the location theory and the data theory. We omit a formal semantics of the logic, which you can find in [14]. Instead, we illustrate the use of the logic by means of examples. Let us begin with a simple spatial formula. The formula

$$(x \to_{\mathsf{n}} y) * (x \to_{\mathsf{d}} d) * (d > 0)$$

expresses that the heap consists of a list node x which points to y via the n-field (n for next) and which stores the data value d in its d-field. The value of d is constrained to be positive by conjoining the data atom $d > 0$. $\mathbf{SL}^*_{\mathsf{data}}$ is a separation logic with *precise* semantics in the usual separation-logic sense (see e.g. [3]), meaning that the above formula describes a heap in which *only* x is allocated. Throughout this paper, we assume that $\mathcal{T}_{\mathsf{data}}$ is the theory of linear integer arithmetic. This is the only data theory that is currently supported by Sloth. Note that this is a limitation of the implementation, not of $\mathbf{SL}^*_{\mathsf{data}}$.

$\mathbf{SL}^*_{\mathsf{data}}$ supports both lists and *list segments*. For example,

$$\mathsf{list}(x, y) * (y \to_{\mathsf{n,d}} (z, d)) * \mathsf{list}(z)$$

expresses that (1) the heap contains a list segment from x to y, (2) the node referenced by y points to z and contains data d, and (3) z points to a list from z to $\mathbf{null}$.

More interestingly, $\mathbf{SL}^*_{\mathsf{data}}$ also supports both trees and *tree fragments*. Consider the formula

$$F := \mathsf{tree}(t, \langle u, v \rangle) * (u \to_{(\mathsf{l,r,d})} (\mathbf{null}, \mathbf{null}, d)) * (v \to_{(\mathsf{l,r,d})} (\mathbf{null}, \mathbf{null}, e)) * (d > e).$$

It asserts that the heap contains (1) a *tree fragment* rooted in t with *stop nodes* u and v, (2) u is a leaf that contains the data value d, (3) v is a leaf that contains

the data value e and (4) $d > e$. Intuitively, the semantics of the tree fragment $\mathsf{tree}(t, \langle u, v\rangle)$ is as follows. As usual in separation logic, there is no sharing of nodes within the model of the predicate. The location t is the root of a tree, from which all but two paths end in **null**; additionally, there are two distinct paths ending in in u and v, respectively. Additionally, u and v are ordered, meaning that u comes before v in an in-order depth-first traversal of the tree. Consequently, for example, any tree satisfying F is *not* a binary search tree, because the node containing d comes before the node containing e in the depth-first traversal and $d > e$.

$\mathbf{SL}^*_{\mathsf{data}}$ predicate calls can also be parameterized by *data predicates* (a possibility omitted in Figure 1 for the sake of simplicity). Data predicates come in two shapes: Unary data predicates that assert properties of the data stored in all nodes in the model of the data structure; and binary predicates, which relate the data value of each node to the data values of its descendants. For instance,

$$\mathsf{list}(x, \{(\alpha > 0), (\mathsf{n}, \alpha < \beta)\})$$

asserts that x points to the head of a sorted list of positive integers. Here α and β are logical variables that range over the data stored in all locations of the list. Thus $(\alpha > 0)$ asserts that the d-field of every location in the list contains a positive number. $(\mathsf{n}, \alpha < \beta)$ expresses that for all locations ℓ_1, ℓ_2 in the list with $\ell_1 \to_{\mathsf{d}} \alpha$ and $\ell_2 \to_{\mathsf{d}} \beta$, if ℓ_2 can be reached from ℓ_1 by first taking an n pointer then $\alpha < \beta$ holds. This enforces sortedness of the list.

To see why it is useful to pair each binary data predicate with a pointer field, consider

$$G := \mathsf{tree}(t, \{(\mathsf{l}, \beta < \alpha), (\mathsf{r}, \beta > \alpha)\}).$$

This expresses that t is the root of a tree and that for all pairs of nodes ℓ_1 and ℓ_2 in the tree containing data values α and β, respectively, if ℓ_2 can be reached from ℓ_1 by first taking an l (i.e., left) pointer then $\beta < \alpha$ holds; and analogously for taking an r (right) pointer and $\beta > \alpha$. Thanks to the implicit universal quantification over tree locations ℓ_1 and ℓ_2, this ensures that for every node in the tree all left descendants store smaller data values and all right descendants store larger data values—the location t is the root of a binary search tree.

Finally, spatial $\mathbf{SL}^*_{\mathsf{data}}$ formulas are closed under Boolean operators. For example, we can assert $F \wedge G$ for F and G as above. This formula is unsatisfiable, as G states that t is the root of a binary search tree, whereas no model of F is a binary search tree. Entailment problems can be modeled via negation, e.g. to check whether $\mathsf{list}(x, y) * \mathsf{list}(y) \models \mathsf{list}(x)$, check whether

$$(\mathsf{list}(x, y) * \mathsf{list}(y)) \wedge \neg\mathsf{list}(x)$$

is unsatisfiable. (Which is the case.) For further examples as well as the formal semantics of $\mathbf{SL}^*_{\mathsf{data}}$, see [14].

3 Deciding $\mathbf{SL}^*_{\mathsf{data}}$

We briefly summarize our decision procedure for $\mathbf{SL}^*_{\mathsf{data}}$. Further details and correctness proofs can be found in [14].

Small-model property. Despite its expressiveness, all satisfiable $\mathbf{SL}^*_{\mathsf{data}}$ formulas have models whose size (i.e., number of allocated locations) is linear in the number of variables and the number of data predicates that occur in the formula:

Theorem 1 (Small-model property for $\mathbf{SL}^*_{\mathsf{data}}$ [14]). *Let F be a satisfiable $\mathbf{SL}^*_{\mathsf{data}}$ formula with n_{list} list variables, n_{tree} tree variables, m_{list} list predicates with data constraints, m_{tree} tree predicates with data constraints, and at most $k \geq 1$ stop locations per* tree *predicate. Then there is a heap interpretation $\mathcal{M}$[4] that satisfies F such that $|\mathcal{M}| \leq \max(4, 2n_{\mathsf{list}} + (3+k)n_{\mathsf{tree}} + 2m_{\mathsf{list}} + 2m_{\mathsf{tree}})$.*

Note that while this bound is tight for worst-case instances, by further analysis of the input formula we can often derive even lower size bounds. We exemplify this in Section 4.

Complexity results. The small-model property implies that the satisifiability problem for $\mathbf{SL}^*_{\mathsf{data}}$ is in NP if the data theory is in NP: We can guess a polynomially-sized model and then check it in deterministic polynomial time. Since NP hardness is trivial ($\mathbf{SL}^*_{\mathsf{data}}$ subsumes propositional logic), NP completeness follows. Unlike with symbolic-heap separation logics such as [1,3,9,11], the entailment problem $F \models G$ for $\mathbf{SL}^*_{\mathsf{data}}$ can be solved by checking unsatisfiability of $F \wedge \neg G$. The entailment problem for $\mathbf{SL}^*_{\mathsf{data}}$ is thus coNP complete.

Observe that these complexity bounds are obtained completely independently of an encoding of $\mathbf{SL}^*_{\mathsf{data}}$ into SMT—in contrast to, for example, [22,23].

Solving $\mathbf{SL}^*_{\mathsf{data}}$ via SMT encodings. Thanks to the small-model property, a variety of decision procedures for $\mathbf{SL}^*_{\mathsf{data}}$ are conceivable. For example, given a size bound n, we could explicitly enumerate models up to size n. In [14], we instead proposed to axiomatize lists and trees of size at most n in an appropriate SMT theory and thus solve $\mathbf{SL}^*_{\mathsf{data}}$ by reduction to SMT. Note that having a size bound is crucial for such an encoding, because it depends on axiomatizing reachability within the data structures—which is only possible given a fixed size bound, as unbounded reachability is not expressible in first-order logic.

Note further that this approach is not quite the same as unfolding the recursive predicate definitions, which would also be possible, but would lead to an exponential-size encoding, as there are $\mathcal{O}(2^n)$ trees of size n.

The encoding proposed and proved correct in [14]—and implemented in Sloth—reduces $\mathbf{SL}^*_{\mathsf{data}}$ to the disjoint theory combination $\mathcal{T}_{\mathsf{array}} \oplus \mathcal{T}_{\mathsf{data}} \oplus \mathcal{T}_{\mathsf{loc}}$, where by $\mathcal{T}_{\mathsf{array}}$ we denote the theory of arrays extended with combinators that can express constant arrays and point-wise array operations [19]. Intuitively, this encoding works as follows. (For the details of the encoding, see [14].)

[4] *Heap interpretations* are the models of $\mathbf{SL}^*_{\mathsf{data}}$ formulas. Intuitively, a heap interpretation is a structure that interprets every pointer field as a partial function. The size of the structure corresponds to the size of the domains of these partial functions. See [14] for the formal definitions.

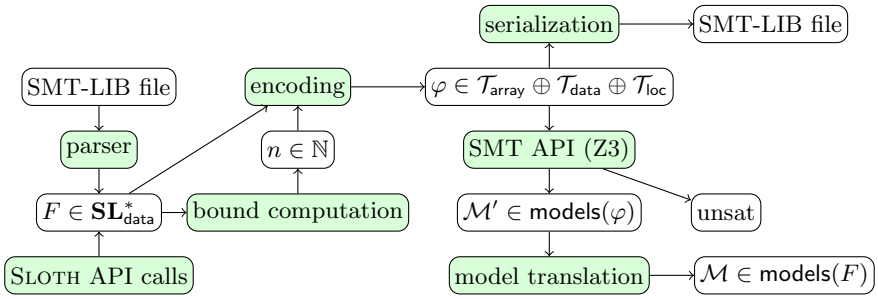


Fig. 2: Solving $\mathbf{SL}^*_{\mathsf{data}}$ formulas in Sloth.

- Sets of locations (representing footprints of spatial formulas) are represented as arrays mapping loc to Bool; set operations are expressed by mapping point-wise operations over these arrays. Intersection of sets corresponds to point-wise conjunction of arrays, for example. This allows for encoding the union and disjointness constraints implied by the use of separating conjunctions.
- Each pointer field is represented as an array from loc to loc.
- Lists and trees of size at most n are encoded by encoding (1) that the heap contains at most n locations and (2) that these locations satisfy the structural properties of the data structure(s) in the $\mathbf{SL}^*_{\mathsf{data}}$ formula. For trees, for example, such properties include that every path ends in **null** or a stop location, that every node has at most one parent in the tree, etc. Note that these properties depend on reachability predicates, which are SMT-definable thanks to the size bound.

4 Implementation

Sloth (a **S**eparation **Lo**gic modulo **Th**eories solver for $\mathbf{SL}^*_{\mathsf{data}}$) is implemented in Python 3 on top of Z3's [20] Python API.[5] The Sloth source code is available at https://github.com/katelaan/sloth.

Both the location theory and the data theory are currently fixed in Sloth: Locations are interpreted as integers; valid $\mathcal{F}_{\mathsf{loc}}$ assertions are equality and disequality assertions about pairs of locations (so $\mathcal{F}_{\mathsf{loc}}$ is the empty theory). $\mathcal{T}_{\mathsf{data}}$ is always instantiated as linear integer arithmetic (LIA). Consequently, $\mathcal{F}_{\mathsf{data}}$ is the set of all atomic LIA formulas supported by Z3. In the future, we plan to support both multiple location theories and multiple data theories.

Figure 2 illustrates the tool architecture of Sloth. The Sloth command-line interface (CLI) processes a custom extension of the SMT-LIB format [2]. This custom extension supports all features of $\mathbf{SL}^*_{\mathsf{data}}$ on top of SMT-LIB. We explain the format later in this section.

[5] Perhaps the name is also related to the anticipated performance of the tool given this technology stack.

In addition, Sloth also exposes an API on top of Z3's python API. Sloth can thus also be used in interactive Python sessions and as a library. In the future, we plan to build additional features on top of the Sloth API. The API could, for example, serve as a backend for program verifiers or for more high-level decision procedures (e.g. with limited support for quantification). We show an example interactive session later in this section.

Once an AST of the input formula $F \in \mathbf{SL}^*_{\mathsf{data}}$ has been built—whether through parsing SMT-LIB or through Sloth API calls—Sloth first computes a number $n \in \mathbb{N}$ that is an upper bound for the size of minimal models of F. We explain the bound-computation algorithm later in this section. Based on the bound n, F is then translated into an SMT formula $\varphi \in \mathcal{T}_{\mathsf{array}} \oplus \mathcal{T}_{\mathsf{data}} \oplus \mathcal{T}_{\mathsf{loc}}$ as outlined in Section 3. The encoding implemented in Sloth very closely follows the encoding proposed in [14].

By default, Sloth feeds the encoded formula into Z3 through Z3's Python API. If Z3 concludes that φ is unsatisfiable, F is unsatisfiable as well. If Z3 proves the satisfiability of the formula, Sloth requests a model $\mathcal{M}' \models \varphi$. It then transforms $\mathcal{M}'$ into a model $\mathcal{M} \models F$ of the original formula. $\mathcal{M}$ is output on the command line (when using the CLI) or returned as Python object (when using the Sloth API). It is also possible to have Sloth serialize the expression DAG of the encoding φ and write it to an SMT-LIB file. Any SMT solver that supports the theory combination $\mathcal{T}_{\mathsf{array}} \oplus \mathcal{T}_{\mathsf{data}} \oplus \mathcal{T}_{\mathsf{loc}}$ can then be called directly on the encoding.

Size bound computation in Sloth. In most cases, the size bound in Theorem 1 is not tight. For example, consider the (classical) conjunction $\mathsf{list}(x) \wedge (y \rightarrow_{\mathsf{n}} z)$. Theorem 1 gives a bound of 6 for this formula ($n_{\mathsf{list}} = 3$, $n_{\mathsf{tree}} = m_{\mathsf{list}} = m_{\mathsf{tree}} = 0$), whereas the actual bound is 1: Every model of the formula must interpret x as equal to y and z as equal to **null** and contain a single pointer from x to **null**. Sloth is able to compute such lower size bounds in many cases, including for the above example. Roughly, the bound n for a formula $F \in \mathbf{SL}^*_{\mathsf{data}}$ is computed as follows:

1. If F contains a top-level conjunct without predicate calls, let n be the number of distinct variables that occur on the left-hand side of points-to assertions.
2. Otherwise, compute the size bound for each spatial formula in F *as if it did not contain any data predicates*.[6] Let m be the maximum over these bounds.
3. Let u and b be the number of unary and binary predicates that occur in spatial formulas that are under the scope of an odd number of negations.
4. Let $n := m + u + 2b$.

Thanks to step (1), Sloth computes the optimal bound in the example above. In steps (2) to (4), we exploit: (a) We can take the maximum of the bounds rather than add the bounds, because all spatial formulas in the formula have to be true in the same global footprint. (b) We only need to retain witnesses for data predicates if they have to be falsified in the model.

[6] I.e., apply the bound for the logic without data predicates, cf. Theorem 1 in [14].

The SLOTH *input format.* To illustrate the SLOTH input format, we show the input files for several of the examples provided in Section 2.[7]

– $(x \to_{\mathsf{n}} y) * (x \to_{\mathsf{d}} d) * (d > 0)$:

```
(declare-const x sl.list.loc)
(declare-const y sl.list.loc)
(declare-const d Int)
(assert (sl.sepcon (sl.sepcon (sl.list.next x y) (sl.list.data x d))
                   (> d 0)))
```

Note the use of the LIA assertion `(> d 0)` below the scope of the separating conjunction.

– $\mathsf{list}(x, y) * (y \to_{\mathsf{n,d}} (z, d)) * \mathsf{list}(z)$:

```
(declare-const x sl.list.loc)
(declare-const y sl.list.loc)
(declare-const z sl.list.loc)
(declare-const d Int)
(assert (sl.sepcon
         (sl.sepcon (sl.list.seg x y) (sl.list.next y z))
         (sl.sepcon (sl.list.data y d) (sl.list z))))
```

– $\mathsf{tree}(t, \{(\mathsf{l}, \beta < \alpha), (\mathsf{r}, \beta > \alpha)\})$ (t is a binary search tree).

```
(declare-const t sl.tree.loc)
(assert (sl.tree.dpred.left (> sl.alpha sl.beta) t))
(assert (sl.tree.dpred.right (< sl.alpha sl.beta) t))
```

The two data predicates for `l` and `r` have to be asserted separately. For example, `(sl.tree.dpred.left (> sl.alpha sl.beta) t)` corresponds to the $\mathbf{SL}^{*}_{\mathsf{data}}$ formulas $\mathsf{tree}(t, \{\mathsf{l}, \alpha > \beta\})$.

Using the SLOTH *API.* We conclude this section with an example Python session illustrating some of the features of the SLOTH API. We check the satisfiability of and get a model for $(\mathsf{list}(x, y) * x \neq y) \wedge \neg((x \to_{\mathsf{n,d}} (z, d)) * (z \to_{\mathsf{n,d}} (y, e)))$. Further usage examples are distributed together with the source code.

```
from z3 import And, Not, Ints
from sloth import *
x, y, z = sl.list.locs('x y z')
d, e = Ints('d e')
# Construct SL* expressions with the same syntax as in the SMT-LIB extension
# SL* expressions can be freely combined with z3 expressions (And, Not,...)
expr1 = sl.sepcon(sl.list.seg(x, y), sl.list.neq(x,y))
expr2 = Not(sl.sepcon(sl.list.next(x, z), sl.list.next(z,y),
            sl.list.data(x,d), sl.list.data(y,e)))
expr = And(expr1, expr2)
```

[7] All these and many more examples are available in the SLOTH github repository.

```
# Check satisfiability
is_sat(expr)
Out: True
# Get model
model(expr)
Out: Model [
       Struct sl.list [
         locs = Integers(6:[z, x], 7:[y])
         null = 2
         next = 6->7
         data = 6->9
       ]
     ]
```

5 Evaluation

Table 1 shows the performance of Sloth on a selection of list and tree benchmarks. Where the $\mathbf{SL}^*_{\mathsf{data}}$ formula was too big to fit in the table, it can be found in Appendix A.[8]

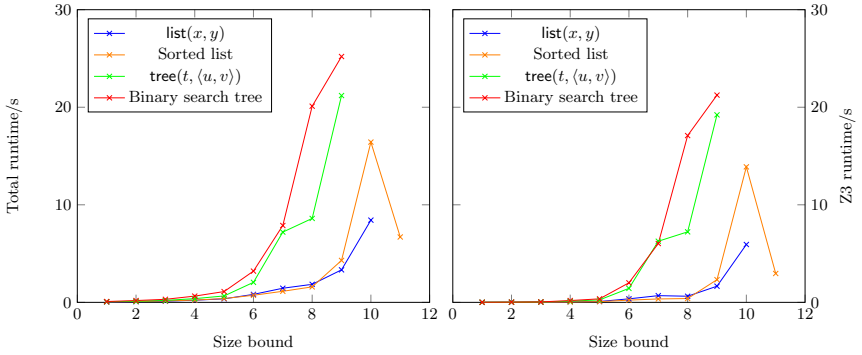
For each benchmark, the table contains the following data. The bound computed by the bound-computation algorithm (i.e., the bound underlying the SMT encoding); whether the benchmark is satisfiable or unsatisfiable; the total runtime of Sloth; the time Z3 needs to solve the expression DAG of the encoding[9]; and the size in kB of an SMT-LIB dump of the encoding.

[8] The benchmarks as well as instructions for running the benchmarks are also included in the Sloth source code repository at https://github.com/katelaan/sloth.

[9] The expression DAG is constructed incrementally through the Z3 Python API during the preprocessing/encoding phase of Sloth.

Benchmark	Bound	Result	Time (sec)		Encoding size (kB)
			Total	z3	
$\mathsf{list}(x, y)$	3	SAT	0.069	0.008	6.07
Sorted list segment	3	SAT	0.083	0.008	6.92
$\mathsf{list}(x) \wedge \neg\mathsf{list}(x)$	2	UNSAT	0.064	0.010	4.70
$(\mathsf{list}(x, y) * \mathsf{list}(y)) \wedge \neg\mathsf{list}(x)$	4	UNSAT	0.598	0.299	37.86
$\mathsf{tree}(t, \langle u, v\rangle)$	6	SAT	0.520	0.019	69.70
$\mathsf{list}(x, y) * \mathsf{tree}(t, \langle u, v\rangle)$	9	SAT	3.731	0.370	497.39
Binary search tree (2 stops)	6	SAT	1.129	0.044	149.98
Tree but not BST (1 stop)	8	SAT	4.140	0.766	492.85
BST but not tree (1 stop)	4	UNSAT	0.601	0.184	53.56
9 allocated tree nodes	9	SAT	0.341	0.104	23.05
BST of size 9 (2 stops)	9	SAT	25.200	21.244	584.41

Table 1: Performance of Sloth.



(a) Total time (b) SMT runtime

Fig. 3: Runtime of Sloth for increasing bounds with a timeout of 30 seconds.

Note that for most benchmarks, the runtime is dominated not by the Z3 call, but by the time that Sloth spends processing the input and constructing the encoding. We are convinced that we will be able to significantly reduce this time in the future, as we have not yet made any attempt at performance optimization.

Z3's performance significantly suffers when combining predicate calls with many, deeply nested separating conjunctions, as we can see in the last row of Table 1. We further illustrate this in Figure 3. Each line of the graph in Figure 3 corresponds to a single benchmark F. Each data point corresponds to a modified version of the benchmark that enforces a model size bound of n through explicit allocation of n locations by means of points-to assertions. Formally, to obtain the data point with size bound n for a list benchmark F, we called Sloth on the formula $F \wedge (y_1 \rightarrow_{\mathsf{n,d}} (y_2, d_1) * \cdots * y_n \rightarrow_{\mathsf{n,d}} (y_{n+1}, d_n))$; for tree benchmarks, we conjoined points-to assertions that allocate a balanced tree of appropriate size, formalized in Appendix A. As expected, the runtime on tree benchmarks increases exponentially as the size bound increases. This illustrates the importance of computing bounds that are as tight as possible. The encoding for both trees and lists is dominated by the encoding of the bounded reachability predicates. This explains why the runtime increases almost as quickly for list benchmarks as for tree benchmarks. This will be improved in a future version of Sloth.

6 Conclusion

We presented Sloth, a prototypical solver for the logic $\mathbf{SL}^*_{\mathsf{data}}$ proposed in [14]. While Sloth is clearly still a work in progress, our evaluation demonstrated that $\mathbf{SL}^*_{\mathsf{data}}$ lends itself to the automation of nontrivial reasoning tasks. A comparison to other tools, in particular to GrassHopper [24], will be carried out in the future. We also plan to extend both $\mathbf{SL}^*_{\mathsf{data}}$ and Sloth with support for doubly-linked, nested data structures, and overlaid data structures; to implement abduction for $\mathbf{SL}^*_{\mathsf{data}}$; and to investigate the possibility of limited support for quantifiers.

References

1. Antonopoulos, T., Gorogiannis, N., Haase, C., Kanovich, M., Ouaknine, J.: Foundations for decision problems in separation logic with general inductive predicates. In: FOSSACS. Springer (2014)
2. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at `www.SMT-LIB.org`
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: FSTTCS (2004)
4. Brotherston, J., Fuhs, C., Pérez, J.A.N., Gorogiannis, N.: A decision procedure for satisfiability in separation logic with inductive predicates. In: CSL-LICS (2014)
5. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: NFM. pp. 3–11 (2015)
6. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM 58(6), 26:1–26:66 (2011)
7. Calcagno, C., Yang, H., O'Hearn, P.: Computability and complexity results for a spatial assertion language for data structures. In: FSTTCS (2001)
8. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Science of Computer Programming 77(9), 1006–1036 (2012)
9. Cook, B., Haase, C., Ouaknine, J., Parkinson, M., Worrell, J.: Tractable reasoning in a fragment of separation logic. In: CONCUR (2011)
10. Demri, S., Deters, M.: Logical investigations on separation logics (2015), http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/DD-esslli15.pdf, lecture Notes, (ESSLLI'15)
11. Iosif, R., Rogalewicz, A., Simacek, J.: The tree width of separation logic with recursive definitions. In: CADE (2013)
12. Iosif, R., Rogalewicz, A., Vojnar, T.: Deciding entailments in inductive separation logic with tree automata. In: ATVA (2014)
13. Itzhaky, S., Banerjee, A., Immerman, N., Nanevski, A., Sagiv, M.: Effectively-propositional reasoning about reachability in linked data structures. In: CAV (2013)
14. Katelaan, J., Jovanović, D., Weissenbacher, G.: A separation logic with data: Small models and automation. In: IJCAR (2018), extended preprint avaiable at http://www.georg.weissenbacher.science/ijcar2018.pdf
15. Lahiri, S., Qadeer, S.: Back to the future: revisiting precise program verification using SMT solvers. POPL (2008)
16. Le, Q.L., Makoto, T., Sun, J., Chin, W.N.: A decidable fragment in separation logic with inductive predicates and arithmetic. In: CAV (2017)
17. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: POPL (2011)
18. Matheja, C., Jansen, C., Noll, T.: Tree-like grammars and separation logic. In: APLAS (2015)
19. de Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: FMCAD. pp. 45–52 (2009)
20. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: TACAS (2008)
21. Navarro Pérez, J., Rybalchenko, A.: Separation logic modulo theories. In: APLAS (2013)

22. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: CAV (2013)
23. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: CAV. Springer (2014)
24. Piskac, R., Wies, T., Zufferey, D.: GRASShopper. complete heap verification with mixed specifications. In: TACAS. pp. 124–139 (2014)
25. Qiu, X., Garg, P., Ştefănescu, A., Madhusudan, P.: Natural proofs for structure, data, and separation. In: PLDI (2013)
26. Reynolds, A., Iosif, R., King, T.: A decision procedure for separation logic in SMT. In: ATVA (2016)
27. Reynolds, A., Iosif, R., Serban, C.: Reasoning in the Bernays-Schoenfinkel-Ramsey Fragment of Separation Logic. In: VMCAI (2017)
28. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)

A Formal Definition of the Benchmarks Used in Section 5

For the sake of completeness, this section formally defines the $\mathbf{SL}^{*}_{\mathsf{data}}$ formulas that were used to generate Table 1 and Figure 3.

Sorted list segment. $\mathsf{list}(x, \{(\alpha > 0), (\mathsf{n}, \alpha < \beta)\})$
Binary search tree (2 stops). $F := \mathsf{tree}(t, \langle u, v\rangle, \{(\mathsf{l}, \beta < \alpha), (\mathsf{r}, \beta > \alpha)\})$
Tree but not BST (1 stop). $\mathsf{tree}(t, u) \wedge \neg(\mathsf{tree}(t, \langle u, v\rangle, \{(\mathsf{l}, \beta < \alpha), (\mathsf{r}, \beta > \alpha)\}))$
BST but not tree (1 stop). $(\neg\mathsf{tree}(t, u)) \wedge (\mathsf{tree}(t, \langle u, v\rangle, \{(\mathsf{l}, \beta < \alpha), (\mathsf{r}, \beta > \alpha)\}))$
9 allocated tree nodes. $G := \circledast_{0 \leq i \leq 8} (y_i \rightarrow_{\mathsf{l,r,d}} (v_{2i+1}, v_{2i+2}, d_i))$, where we use $\circledast$ as shorthand for an iterated separating conjunction and where we define $v_i := y_i$ for $i \leq 8$ and as **null** otherwise.
BST of size 9 (2 stops). $F \wedge G$
Tree benchmarks in Figure 3. We conjoined $\circledast_{0 \leq i < n} (y_i \rightarrow_{\mathsf{l,r,d}} (v_{2i+1}, v_{2i+2}, d_i))$ to obtain a size bound of n.