Separation Logic Modulo Theories

Juan Antonio Navarro Pérez[1] and Andrey Rybalchenko[2]

[1] University College London
[2] Microsoft Research Cambridge and Technische Universität München

Abstract. Logical reasoning about program behaviours often requires dealing with heap structures as well as scalar data types. Advances in Satisfiability Modulo Theories (SMT) offer efficient procedures for dealing with scalar values, yet they lack expressive support for dealing with heap structures. In this paper, we present an approach that integrates separation logic—a prominent logic for reasoning about linked data structures on the heap—and existing SMT solving technology. Our model-based approach communicates heap aliasing information between theory and separation logic reasoning, providing an efficient decision procedure for discharging verification conditions in program analysis and verification.

1 Introduction

Satisfiability Modulo Theory (SMT) solvers play an important role in the construction of abstract interpretation tools [11, 12]. They efficiently reason about various scalar data types, e.g., bit-vectors and numbers, as well as uninterpreted functions and arrays [1,7,14,15,18]. Today's SMT solvers, however, lack support for dealing with dynamically allocated heap data structures. Thus, a combination of theory reasoning with separation logic [25]—a successful logical formalism of resource allocation—has the potential to boost a wide range of program analysis systems: manual/tool assisted proof development [17, 21], extended static checking [5, 16], and automatic inference of heap shapes [2, 8].

In this paper we develop a method to augment an SMT solver with separation logic reasoning for linked list segments and their length. Our method decides the validity of entailments of the form $\Pi \wedge \Sigma \rightarrow \Pi' \wedge \Sigma'$, where Π, Π' are *arbitrary* theory assertions decided by the SMT solver, while Σ, Σ' symbolically describe a spatial conjunction of pointers and acyclic list segments. In contrast, existing decision procedures combine list segments with conjunctions of equality and disequality predicates only. Moreover, the length information on list segments allows our techniques to prove properties where a tight interaction between program data and the shape of heap structures is needed.

The crux of our method lies in an interaction of the model-based approach to theory combination [13] and a so-called *match* function that derives logical implication between pairs of spatial conjunctions. Models of Π, called stacks, guide the process of showing that all heaps satisfying Σ also satisfy Σ'. The *match* function produces an assertion describing a set of stacks for which the current derivation is applicable. This assertion is used to prune the search space

and find more stacks for which the entailment has not been proved yet. Our method thus benefits from the efficiency of SMT solvers to maintain a logical representation of the search space already explored.

In summary, we present an efficient SMT-based decision procedure for separation logic with acyclic list segments with length. Our main contribution is the entailment checking algorithm for separation logic in combination with decidable theories, together with its formal proof of correctness.

Related work Our approach improves upon a previous separation logic proving method [22], which relied on paramodulation for equality reasoning [23] and provided improvements of several orders of magnitude on efficiency with respect to existing systems at the time. The current work extends this method, which only dealt with pure equalities, to support arbitrary theory expressions in both pure and spatial parts of the entailment. Our new *match* function generalises previous unfolding inference rules—in turn based on inferences from [3, 4]—and runs in linear time avoiding case reasoning as performed by most other systems. The logic context of an SMT solver, rather than literals in a clausal representation, maintains the explored search space. Doing so we remove a technical limitation from the approach in [22]: spatial reasoning no longer requires access to equality reasoning steps, and off-the-shelf SMT solvers become directly applicable.

Separation logic entailment checking in the fragment limited to list segments and pure equalities was shown to be decidable in polynomial time [10], and a tool exploiting this result has been developed [19]. Although we are mainly interested in reasoning about rich theory assertions describing stacks, exploration of this polynomial time result is an interesting direction for future work. In the opposite direction, work such as that from Botinčan et al. [6] and Chin et al. [9] develop techniques for dealing with more general user-specified predicates beyond simple list segments. The former work, moreover, also relies on SMT for pure reasoning. The cost of this increased expressivity, however, is that such procedures become incomplete. Our logic is more restrictive, allowing us to develop a more efficient, sound, terminating and complete procedure for entailment checking.

Piskac et al. [24] also developed a decision procedure for the list segment fragment. Their approach translates entailments to an intermediate logic which, given suitable axioms, is then decided by an SMT solver. The technique works as well for slightly more general structures, such as sorted list segments and doubly linked lists, but further generalisations probably require changes and extensions in the intermediate logic. We believe that generalisations to our approach are more straightforward, since to support other predicates we only need to define a suitable *subtract* operator, as we discuss for the case of linked list segments with length later in Section 4 of this paper.

Finally, Iosif et al. [20] have recently proved a decidability result for a large class of separation logic formulas with recursive predicate definitions. Their result, which without a doubt represents a major advance in the theory of separation logic, is based on a monadic second order logic encoding where formulas with a bounded tree width are known to be decidable. Although their fragment considered still has a few limitations—unlike our algorithm, their decidability re-

sult does not apply for structures with dangling data pointers—these theoretical results have opened up exciting directions for future research.

2 Illustration

To motivate our work, we illustrate how our algorithm discharges a verification condition produced in the analysis of a program. Consider the following C++ snippet that retrieves data associated with the k-th element of a linked list.

```
struct node { int data; node* next };
node* get(node* p, int k) { /* assume: ∃n. 0 ≤ k < n ∧ lseg(p, nil, n) */
  node* q = p;
  for (int i = 0; i < k; i++) q = q->next;
  return q->data;
}
```

The implementation is memory safe only if the value of k is less than the length of the list rooted at p, as made explicit by the assumption at the beginning of the function. The $\mathsf{lseg}(p, \mathsf{nil}, n)$ predicate denotes that, starting from the location p in the heap and following an acyclic chain of exactly n next-pointers, we reach the end of the list, i.e., nil. When the start/finish locations are equal, and thus necessarily $n = 0$, the list is empty and no nodes are allocated.

We remark that, due to the crucial mix of arithmetic and spatial reasoning involved—on how indices relate to the length of chains of dynamic pointers—the automated verification of even such simple code is often beyond the capabilities of existing program analysers. An analyser would symbolically execute the code, producing a series of verification conditions to be discharged. At some point, for example, the analyser needs to establish the validity of the entailment

$$\overbrace{i \simeq i' + 1}^{\Pi} \wedge \overbrace{\mathsf{lseg}(p, q', i') * \mathsf{next}(q', q) * \mathsf{lseg}(q, \mathsf{nil}, n - i' - 1)}^{\Sigma} \rightarrow \underbrace{\mathsf{lseg}(p, q, i) * \mathsf{lseg}(q, \mathsf{nil}, n - i)}_{\Sigma'},$$

explicating changes in the program state—respectively denoted by primed and regular variables—before and after the execution of each loop iteration. Note the use of '$\simeq$' for equality in the formal language, distinguished from '$=$' in the meta language. The star connective '$*$' states that memory cells allocated by the heap predicates are necessarily disjoint or *separated* from each other in memory; while $\mathsf{next}(q', q)$ represents a heap portion of exactly one node allocated at q' (the value of q before the loop execution) whose next pointer has the same value as q (after executing the loop).

Proving this entailment—which still involves a mix of arithmetic and spatial reasoning—shows that $\mathsf{lseg}(p, q, i) * \mathsf{lseg}(q, \mathsf{nil}, n - i)$ is a loop invariant. To this end, the algorithm performs the following key steps: First it enumerates pure models, assignments to program variables, that allow satisfying both Π and Σ in

the antecedent. For each pure model s, the algorithm attempts to (symbolically) prove that every heap h satisfying the antecedent, $s, h \models \Pi \wedge \Sigma$, also satisfies the consequence, $s, h \models \Sigma'$. The assignment is generalised as an assertion M pruning models of Π that lead to similar reasoning steps as with s. The entailment is valid if and only if all models of the antecedent are successfully considered.

So, we first build a constraint characterising the satisfiability of the spatial part of the antecedent. This constraint requires each spatial predicate in Σ to be sound, e.g. list lengths are non-negative, and each pair of predicates to be separated from each other. In particular, if two predicates start at the same heap location, necessarily one of them must be an empty heap with no allocated nodes. For our example entailment, the soundness of $\mathsf{lseg}(p, q', i')$ requires that the length of the list segment is non-negative, i.e. $0 \leq i'$, and the start/finish locations coincide if and only if the length of the list is zero, i.e. $p \simeq q' \leftrightarrow i' \simeq 0$. The soundness condition of $\mathsf{lseg}(q, \mathsf{nil}, n - i' - 1)$ is similarly determined.

	soundness of ...
$0 \leq i' \wedge (p \simeq q' \leftrightarrow i' \simeq 0)$	$\mathsf{lseg}(p, q', i')$
$0 \leq n - i' - 1 \wedge (q \simeq \mathsf{nil} \leftrightarrow n - i' - 1 \simeq 0)$	$\mathsf{lseg}(q, \mathsf{nil}, n - i' - 1)$

Additionally, say, for the pair of predicates $\mathsf{lseg}(p, q', i')$ and $\mathsf{lseg}(q, \mathsf{nil}, n - i' - 1)$ their separation condition is represented as $p \simeq q \rightarrow (p \simeq q' \vee q \simeq \mathsf{nil})$, i.e., if the start location p of the first predicate is equal to the start location q of the second predicate then either one of them must represent an empty segment. The separation condition for each pair of predicates in Σ is similarly computed.

	separation of ...
$p \simeq q' \rightarrow p \simeq q'$	$\mathsf{lseg}(p, q', i')$ and $\mathsf{next}(q', q)$
$p \simeq q \rightarrow (p \simeq q' \vee q \simeq \mathsf{nil})$	$\mathsf{lseg}(p, q', i')$ and $\mathsf{lseg}(q, \mathsf{nil}, n - i' - 1)$
$q' \simeq q \rightarrow q \simeq \mathsf{nil}$	$\mathsf{next}(q', q)$ and $\mathsf{lseg}(q, \mathsf{nil}, n - i' - 1)$

Finally, to make sure that nothing is allocated at the nil location we have to assert, say for $\mathsf{lseg}(p, q', i')$, that if the start location p is nil then necessarily the finish location q' is also nil. For the case of $\mathsf{next}(q', q)$ we simply assert that q' is not nil. We thus obtain three additional assertions.

	nil is not allocated by ...
$p \simeq \mathsf{nil} \rightarrow q' \simeq \mathsf{nil}$	$\mathsf{lseg}(p, q', i')$
$q' \not\simeq \mathsf{nil}$	$\mathsf{next}(q', q)$
$q \simeq \mathsf{nil} \rightarrow \mathsf{nil} \simeq \mathsf{nil}$	$\mathsf{lseg}(q, \mathsf{nil}, n - i' - 1)$

We refer to the conjunction of all above assertions as *well-formed*(Σ).

Crucially, these assertions do not contain spatial predicates any more, so an SMT solver is used to search for models of $\Pi \wedge$ *well-formed*(Σ). If no such model exists the entailment is vacuously true. In our example, however, the solver finds the model $s = \{p \mapsto 42, q' \mapsto 47, q \mapsto 29, i' \mapsto 1, i \mapsto 2, n \mapsto 3\}$. To show that,

with respect to this assignment s, every heap h model of Σ is also a model of Σ', we try to establish a *match* between Σ and Σ'. Specifically for each predicate in Σ' we seek a matching 'chain' of predicates in Σ such that the finish and start location of adjacent predicates is equal with respect to s.

So, we first search for a match for $\mathsf{lseg}(p, q, i) \in \Sigma'$ connecting p to q in i steps within the antecedent Σ. Trivially, since $s(p) = s(p)$, the chain must begin with $\mathsf{lseg}(p, q', i')$ leaving us yet to connect q' with q in $i - i'$ steps. This issues a new request to match $\mathsf{lseg}(q', q, i - i')$ against the remaining predicates from Σ, namely, $\mathsf{next}(q', q) * \mathsf{lseg}(q, \mathsf{nil}, n - i' - 1)$. Similarly, the chain must now continue with $\mathsf{next}(q', q)$ and a new request to match $\mathsf{lseg}(q, q, i - i' - 1)$ is issued. This time, however, $s \models i - i' - 1 \simeq 0$ so the match is completed. In the same vein, we search for a match for $\mathsf{lseg}(q, \mathsf{nil}, n - i) \in \Sigma'$ against the only remaining $\mathsf{lseg}(q, \mathsf{nil}, n - i' - 1)$ in Σ. Luckily, since $s \models n - i \simeq n - i' - 1$, both connect q to nil in the same number of steps and the match quickly succeeds. Since all predicates of Σ' are matched, and all predicates in Σ were used in a match, we conclude that Σ and Σ' have match exactly with respect to the current s.

The algorithm keeps track of the assertions required on s for the match to succeed, namely $M = (i - i' - 1 \simeq 0 \wedge n - i \simeq n - i' - 1)$. The matching proof obtained for this particular assignment s is thus generalised to all models satisfying M, and we may continue the enumeration of models for the antecedent excluding those where M is true.

A second call to the SMT solver reveals that $\Pi \wedge \mathit{well\text{-}formed}(\Sigma) \wedge \neg M$ is now unsatisfiable. Although our spatial reasoning procedure is unaware of this fact, the arithmetic capabilities of the SMT solver easily figure out that the hypothesis $\Pi = (i \simeq i' + 1)$ forces M to be always true. Since matching is possible for all models of the antecedent, we thus conclude that the entailment is valid.

3 Preliminaries

We write $f \colon X \to Y$ to denote a *function* with domain $X = \operatorname{dom} f$ and range Y; and $f \colon X \rightharpoonup Y$ to denote a *finite partial function* with $\operatorname{dom} f \subseteq X$. We write $f_1 * \cdots * f_n$ to simultaneously assert the disjointness of the domains of n functions, namely $\operatorname{dom} f_i \cap \operatorname{dom} f_j = \emptyset$ when $i \neq j$, and denote the, therefore, well defined function $f = f_1 \cup \cdots \cup f_n$. We sometimes describe functions by explicitly enumerating their elements; for example $f = \{a \mapsto b, b \mapsto c\}$ is the function such that $\operatorname{dom} f = \{a, b\}$, $f(a) = b$, and $f(b) = c$.

Satisfiability modulo theories We assume a first-order many-sorted language where each function symbol f of arity n has a signature $f \colon \tau_1 \times \cdots \times \tau_n \to \tau$, i.e. the symbol f takes n arguments of respective sorts τ_i and produces an expression of sort τ. A constant symbol is a 0-ary function symbol. Constant and function symbols are combined respecting their sorts to build syntactically valid *expressions*. We use $x \colon \tau$ to denote an expression x of sort τ. Each sort τ is associated with a set of values, for convenience also denoted τ. In particular we assume that booleans and integers, namely $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$ and $\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$,

are among the available sorts. We refer to a function symbol of boolean sort as a *predicate symbol*, and a boolean expression as a *formula*.

Some symbols have fixed predefined theory interpretations. For example the predicate $\simeq\colon \tau \times \tau \to \mathbb{B}$ tests equality between two expressions of the same sort; while theory symbols from the boolean domain, i.e. conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), truth ($\top$), falsity ($\bot$), entailment ($\to$), boolean equivalence ($\leftrightarrow$), and first-order quantifiers ($\forall$, $\exists$), have their standard interpretations. We similarly assume theory symbols for integer arithmetic with their usual interpretation and use nil as an alias for the integer constant 0.

Some function symbols are also left uninterpreted. A *variable*, in particular, is an uninterpreted constant symbol. Interpretations map uninterpreted symbols to values of the appropriate sort. We write $s(x)$ to denote the result of evaluating the expression x under the interpretation s. For example, if $s = \{n \mapsto 2\}$ then $s(1 + n) = 3$. A formula F is *satisfiable* if there is an s such that $s(F) = \mathit{true}$; in such case we also write $s \models F$ and say that s is a *model* of F. A formula is *valid* if it is satisfied by all interpretations. The job of an SMT solver is, given a formula F, to find a model such that $s \models F$ or prove that none exists.

Separation logic On top of the theories already supported by the SMT solver, we define *spatial symbols* to build expressions to describe properties about heaps. We thus introduce the spatial predicate symbols $\mathsf{emp}\colon \mathbb{B}$, $\mathsf{next}\colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$, $\mathsf{lseg}\colon \mathbb{Z}\times\mathbb{Z}\times\mathbb{Z} \to \mathbb{B}$, and $*\colon \mathbb{B}\times\mathbb{B} \to \mathbb{B}$ for, respectively, the empty heap, a points-to relation, an acyclic list segment with length, and the spatial conjunction. A *spatial formula* is one that may include spatial symbols, and a *pure formula* is one where no spatial symbols occur.

A *stack* is an interpretation for pure expressions, mapping uninterpreted symbols to suitable values. A *heap* is a partial finite map $h\colon \mathbb{Z} \rightharpoonup \mathbb{Z}$ that connects memory locations, represented as integers, and gives meaning to spatial symbols. Given a stack s, a heap h, and a spatial formula F we inductively define the spatial satisfaction relation $s, h \models F$ as $s, h \models \Pi$ if Π is pure and $s \models \Pi$, $s, h \models \mathsf{emp}$ if $h = \emptyset$, $s, h \models \mathsf{next}(x, y)$ if $h = \{s(x) \mapsto s(y)\}$, $s, h \models F_1 * F_2$ if $h = h_1 * h_2$ for some h_1 and h_2 such that $s, h_1 \models F_1$ and $s, h_2 \models F_2$. The acyclic list segment with length is inductively defined by

$$\begin{aligned}\mathsf{lseg}(x, z, n) = (x \simeq z \wedge n \simeq 0 \wedge \mathsf{emp})\\ \vee (x \not\simeq z \wedge n > 0 \wedge \exists y.\ \mathsf{next}(x, y) * \mathsf{lseg}(y, z, n - 1))\ .\end{aligned}$$

For example, given that $s = \{x \mapsto 3, y \mapsto 2, n \mapsto 1\}$ and $h = \{3 \mapsto 5, 5 \mapsto 2\}$, it follows that $s, h \models \mathsf{lseg}(x, y, n + 1)$. As with pure formulas, we say that a spatial formula F is *satisfiable* if there is a pair (s, h) such that $s, h \models F$; and *valid* if it is satisfied by every stack-heap pair. In particular the entailment $F \to G$ is valid if and only if every model of F is also a model of G. For a spatial formula F, we write $s \models F$ to denote that $s, h \models F$ for all heaps h.

We remark that this definition does not treat nil in any special way. To regain its expected behaviour, i.e. on a spatial formula F *nothing* may be allocated at the nil location, it is enough to consider $F * \mathsf{next}(\mathsf{nil}, \mathsf{nil})$ instead. Furthermore,

function $prove(\Pi \wedge \Sigma \rightarrow \Pi' \wedge \Sigma')$
1: $\Gamma := \Pi \wedge \mathit{well\text{-}formed}(\Sigma)$
2: $\Delta := \mathit{alloc}(\Sigma)$
3: **if satisfiable** $\Gamma \wedge \neg\Pi'$ **return** *invalid*
4: **while exists** s **such that** $s \models \Gamma$ **do**
5: $M := \mathit{match}(s, \Delta, \Sigma, \Sigma')$
6: **if** $s \not\models M$ **return** *invalid*
7: $\Gamma := \Gamma \wedge \neg M$
8: **return** *valid*

function $match(s, \Delta, \Sigma, \Sigma')$
9: **if exists** $S \in \Sigma$ **such that** $s \models \mathit{empty}(S)$
10: **return** $\mathit{empty}(S) \wedge \mathit{match}(s, \Delta, \Sigma \setminus S, \Sigma')$
11: **if exists** $S' \in \Sigma'$ **such that** $s \models \mathit{empty}(S')$
12: **return** $\mathit{empty}(S') \wedge \mathit{match}(s, \Delta, \Sigma, \Sigma' \setminus S')$
13: **if exists** $S \in \Sigma$ **and** $S' \in \Sigma'$ **such that** $s \not\models \mathit{separated}(S, S')$
14: $(S'', D) := \mathit{subtract}(\Delta, S', S)$
15: **if** $s \models \mathit{sound}(S'') \wedge D$ **return** $\neg\mathit{separated}(S, S') \wedge \mathit{sound}(S'') \wedge D \wedge \mathit{match}(s, \Delta, \Sigma \setminus S, (\Sigma' \setminus S') * S'')$
16: **if** $\Sigma = \emptyset$ **and** $\Sigma' = \emptyset$ **return** $\top$ **else return** $\bot$

Fig. 1. Model-driven entailment checker

although the language allows spatial conjunctions of arbitrary boolean formulas, we focus on the fragment where such conjuncts are restricted to spatial predicates. In the following when we say "a spatial conjunction" what we actually mean is "a spatial conjunction of spatial predicates". Also for convenience, a spatial conjunction $\Sigma = S_1 * \cdots * S_n$ is often treated in the meta level as a multi-set of boolean spatial predicates where $|\Sigma| = n$ is the number of conjuncts. We use set theory symbols, which are always to be interpreted as *multi*-set operations, to describe relations among spatial predicates and conjunctions. For example:

$$\mathsf{next}(y, z) \in \mathsf{lseg}(x, y) * \mathsf{next}(y, z) \qquad \mathsf{next}(x, y) * \mathsf{next}(x, y) \not\subseteq \mathsf{next}(x, y)$$
$$\mathsf{emp} * \mathsf{emp} * \mathsf{emp} \setminus \mathsf{emp} = \mathsf{emp} * \mathsf{emp}\ .$$

4 Decision procedure for list segments and theories

We begin this section describing the building blocks that, when put together as shown in the *prove* and *match* functions of Figure 1, constitute a decision procedure for entailment checking. The procedure works for entailments of the form $\Pi \wedge \Sigma \rightarrow \Pi' \wedge \Sigma'$, where both Π and Π' are pure formulas, with respect to any background theory supported by the SMT solver, while both Σ and Σ' are spatial conjunctions.

To abstract away the specific details of individual spatial predicates, we first define $\mathit{addr}(S)$, $\mathit{sound}(S)$, and $\mathit{empty}(S)$—respectively the *address*, *soundness*,

and *emptiness condition* of a spatial predicate S—as follows:

S	$addr(S)$	$sound(S)$	$empty(S)$
emp	nil	$\top$	$\top$
$\mathsf{next}(x, y)$	x	$\top$	$\bot$
$\mathsf{lseg}(x, y, n)$	x	$0 \leq n \wedge (x \simeq y \leftrightarrow n = 0)$	$x \simeq y$

The soundness condition is a formula that must be satisfied by any model of the spatial predicate, formally: if $s, h \models S$ then $s \models sound(S)$. If, furthermore, the emptiness condition is also true, then its corresponding heap model *must* be empty. Conversely, if the emptiness condition is false then the address of the predicate *must* necessarily occur in the domain of any heap satisfying the spatial predicate. Formally: given $s \models sound(S) \wedge empty(S)$ it follows $s, h \models S$ if and only if $h = \emptyset$; and if $s, h \models \neg empty(S) \wedge S$ then, necessarily, $s(addr(S)) \in \operatorname{dom} h$.

Separation We begin defining the notion of *separation* which is used, in particular, at lines 13 and 15 of the algorithm in Figure 1. Given any two spatial predicates S and S', the formula

$$separated(S, S') = addr(S) \simeq addr(S') \rightarrow empty(S) \vee empty(S')$$

states that two predicates are separated if either their addresses are distinct or one of the two predicates is empty. Otherwise, if both predicates are non-empty and share the same address, the formula $S * S'$ would not be satisfied. More formally, if $s, h \models S * S'$ then necessarily $s \models separated(S, S')$. We also say that two spatial predicates S and S' collide, with respect to the given stack s, if it is the case that $s \not\models separated(S, S')$.

Well-formedness The *well-formedness condition*, found at line 1 in Figure 1, is defined for a spatial conjunction $\Sigma = S_1 * \cdots * S_n$ as the pure formula

$$well\text{-}formed(\Sigma) = \bigwedge_{1 \leq i \leq n} sound(S_i) \wedge \bigwedge_{1 \leq i < j \leq n} separated(S_i, S_j) \ ,$$

which states that all predicates are sound and every pair is separated. The reader might want to revisit the example in Section 2, where the well-formedness of $\Sigma = \mathsf{lseg}(p, q', i') * \mathsf{next}(q', q) * \mathsf{lseg}(q, \mathsf{nil}, n - i' - 1)$ is computed. In fact, since nil is not special in our definition of the semantics, what we computed was the well-formedness of $\Sigma * \mathsf{next}(\mathsf{nil}, \mathsf{nil})$, and the last three assertions for the non-allocation of nil are just the separation conditions with respect to the added $\mathsf{next}(\mathsf{nil}, \mathsf{nil})$. The importance of the well-formedness condition comes from the fact that, as the next theorem states, it characterises the satisfiability of spatial conjunctions.

Theorem 1. *A spatial conjunction Σ is satisfiable if, and only if, the pure formula well-formed(Σ) is satisfiable.*

Allocation Given a stack s and a spatial conjunction $\Sigma = S_1 * \cdots * S_n$, the *allocated set* $alloc(\Sigma|s) = \{s(addr(S_i)) \mid s \not\models empty(S_i)\}$ is a set of locations necessarily allocated by any heap h satisfying Σ. That is, for all h such that $s, h \models \Sigma$ it follows that $alloc(\Sigma|s) \subseteq \operatorname{dom} h$.

The *allocation function*, found at line 2 in Figure 1, is defined without reference to any stack as

$$alloc(\Sigma) = \lambda x. \bigvee_{1 \leq i \leq n} \neg empty(S_i) \wedge x \simeq addr(S_i) \ ,$$

mapping a given variable x to a formula symbolically testing whether x should be allocated on heaps satisfying Σ. That is, if $\Delta = alloc(\Sigma)$ and s is any stack, we have that $s(x) \in alloc(\Sigma|s)$ if and only if $s \models \Delta(x)$. For instance, taking the same example Σ as before,

$$\Delta(x) = (p \not\simeq q' \wedge x = p) \vee (x \simeq q') \vee (q \not\simeq \mathsf{nil} \wedge x = q) \vee (x \simeq \mathsf{nil}) \ .$$

Thus x is considered allocated if it is equal to q' or nil; or if it is equal to the start location, p or q, of a non-empty list segment.

Subtraction We now proceed towards the introduction of the *subtraction operation*, occurring at line 14 in Figure 1, which lies at the core of our matching function. When trying to prove an entailment $s \models \Sigma \to \Sigma'$, we want to show that any heap model of Σ is also a model of Σ'. Thus, if we find a pair of colliding predicates $S \in \Sigma$ and $S' \in \Sigma'$, the portion of the heap that satisfies S must overlap with the portion of the heap satisfying S'. In fact, it is not hard to convince oneself—for the list segment predicates considered—that the heap model of S' should match exactly that of S plus some extra surplus.

Given two spatial predicates S, S', and an allocation function Δ, the subtraction operation $(S'', D) := subtract(\Delta, S', S)$ returns a pair where S'' is the remainder of subtracting S from S', and D is an additional side condition. Intuitively, if D is not satisfied, then there is a counterexample for the subtraction (c.f. Proposition 2 later). Specifically, for each pair of predicates we have:

S'	S	S''	D
$\mathsf{next}(x', z)$	$\mathsf{next}(x, y)$	emp	$y \simeq z$
$\mathsf{lseg}(x', z, n)$	$\mathsf{next}(x, y)$	$\mathsf{lseg}(y, z, n-1)$	$\top$
$\mathsf{next}(x', z)$	$\mathsf{lseg}(x, y, n)$	emp	$y \simeq z \wedge n \simeq 1$
$\mathsf{lseg}(x', z, n)$	$\mathsf{lseg}(x, y, m)$	$\mathsf{lseg}(y, z, n-m)$	$y \not\simeq z \to \Delta(z) \vee m \simeq 1$

Formalising our stated intuition, the following proposition states how if S'' is obtained by subtracting S from S' then, under suitable assumptions, the spatial predicate S' is equivalent to $S * S''$. The validity of this statement, as well as the following proposition, is easily verified by inspection of the relevant definitions.

Proposition 1. *Let Σ be a spatial conjunction and S, S' a pair of spatial predicates. Let $\Delta = alloc(\Sigma)$, let $(S'', D) = subtract(\Delta, S', S)$, and let s be a stack such that $s \models \neg separated(S, S') \wedge sound(S'') \wedge D$. Then the following claims hold.*

1. $s, h \models \Sigma * S * S'' \to \Sigma * S'$ *for every heap* h.
2. *if* $s, h \models \Sigma * S'$ *and* $s, h_1 \models S$ *for some* $h_1 \subseteq h$ *then* $s, h \models \Sigma * S * S''$.

Conversely, the following proposition states that if S and S' collide, but the subtraction is not successful, i.e. D is not satisfied, then it is possible to build a counterexample to the original entailment.

Proposition 2. *Let* Σ *be a spatial conjunction and* S, S' *a pair of spatial predicates. Let* $\Delta = alloc(\Sigma)$, *let* $(S'', D) = subtract(\Delta, S', S)$, *and let* s *be a stack such that* $s \models \neg separated(S, S')$. *If* $s \not\models sound(S'') \wedge D$ *then there is a* h_1 *such that* $s, h_1 \models S$, *but for all* h *such that* $h_1 \subseteq h$ *we have* $s, h \not\models \Sigma * S'$.

As an example suppose we want to determine the validity of $\Sigma * S \to S'$, where each $\Sigma = \mathsf{lseg}(y, z, m)$, $S = \mathsf{lseg}(x, y, n)$, and $S' = \mathsf{lseg}(x, z, n + m)$. We would then have that $\Delta = \lambda v.\ (y \not\simeq z \wedge v \simeq y)$, $S'' = \mathsf{lseg}(y, z, n + m - n)$, and $D = (y \not\simeq z \to \Delta(z) \vee n \simeq 1) = (y \not\simeq z \to (y \not\simeq z \wedge z \simeq y) \vee n \simeq 1)$. Assume a stack $s = \{x \mapsto 1, y \mapsto 2, z, \mapsto 3, n \mapsto 2, m \mapsto 1\}$. With respect to s, it is clear that S and S' collide, as they are both non-empty lists starting on the same location $s(x) = 1$. However, $s \not\models D$, since $s \models y \not\simeq z$ but $s \not\models \Delta(z)$, because z is not necessarily allocated in Σ, and $s(n) = 2 \neq 1$. Proposition 2 asserts the existence of a heap, in this case say $h_1 = \{1 \mapsto 3, 3 \mapsto 2\}$, such that $s, h_1 \models \mathsf{lseg}(x, y, n)$ but cannot be extended into a model of $\mathsf{lseg}(x, z, n + m)$ as this would introduce a cycle. In particular with the heap $h = h_1 * \{2 \mapsto 3\}$ we have $s, h \not\models \Sigma * S \to S'$, providing a counterexample for the original entailment. We end this section with the remark that, in order to generalise our method to other inductive predicates, it is enough to find a suitable *subtract* operator satisfying the conditions imposed by Propositions 1 and 2.

Matching and proving To finalise the description of our decision procedure for entailment checking we have only left to put all the ingredients together, as shown in Figure 1, into the *match* and *prove* functions.

The *match* function tries to establish whether $s \models \Sigma \to \Sigma'$, in a context where Δ specifies heap locations that must be allocated. The function proceeds by matching predicates in Σ with those in Σ', reducing their number of conjuncts as progress is made, and succeeding if eventually both Σ and Σ' become empty. Furthermore, when successful, the function returns an assertion M generalising the matching proof to all stacks that, like s, also satisfy M.

The function begins by inspecting Σ and Σ' to discard, at lines 10 and 12, any predicates that are empty with respect to s, recursively calling itself to verify the rest of the entailment. After removing all such empty predicates, if a pair of colliding predicates $S \in \Sigma$ and $S' \in \Sigma'$ is found, on line 14 we then proceed to compute $subtract(\Delta, S', S) = (S'', D)$. If the subtraction is successful, signalled by the fact that $s \models sound(S'') \wedge D$, we may replace S' with $S * S''$ in Σ', before removing S from both Σ and Σ' and proceeding with the next recursive call. Alternatively, we reach the bottom of the recursion at line 16, succeeding only if both Σ and Σ' have become empty. This behaviour is formalised in the following theorem, proved later in Section 5.

Theorem 2. *Given three spatial conjunctions $\hat{\Sigma}$, Σ, Σ', let $\Delta = alloc(\hat{\Sigma} * \Sigma)$, and let s be a stack such that $s \models$ well-formed$(\hat{\Sigma} * \Sigma)$. It follows that: 1) the function match$(s, \Delta, \Sigma, \Sigma')$ always terminates with a result M, 2) the execution requires $O(|\Sigma| + |\Sigma'|)$ recursive steps, 3) if $s \models M$ then the entailment $M \wedge \hat{\Sigma} * \Sigma \rightarrow \hat{\Sigma} * \Sigma'$ is valid, and 4) if $s \not\models M$ then $s \not\models \hat{\Sigma} * \Sigma \rightarrow \hat{\Sigma} * \Sigma'$.*

The main *prove* function, which determines whether $\Pi \wedge \Sigma \rightarrow \Pi' \wedge \Sigma'$ is valid, begins computing with the pure formula $\Gamma := \Pi \wedge$ *well-formed*(Σ) and the allocation function $\Delta :=$ *alloc*(Σ). An SMT solver is first used to test whether there are any models for $\Gamma \wedge \neg\Pi'$ since, if this is the case, then it is possible to build a counterexample that satisfies the antecedent but not the consequence of the entailment. Otherwise the function proceeds iteratively using the SMT solver to find models of Γ to guide the search for a proof or a counterexample. Given one such stack s, the *match* function is called to check the validity of the entailment with respect to s. If successful, *match* returns a formula M generalising the conditions in which the entailment is valid, so the search may continue for models where M does not hold. Iterations proceed until either all models have been checked or a counterexample is found in the process. Formally we state the following theorem, whose proof is given in Section 5.

Theorem 3. *Given an entailment $\Pi \wedge \Sigma \rightarrow \Pi' \wedge \Sigma'$ we have: i) the function prove$(\Pi \wedge \Sigma \rightarrow \Pi' \wedge \Sigma')$ always terminates, and ii) the return value corresponds to the validity of $\Pi \wedge \Sigma \rightarrow \Pi' \wedge \Sigma'$.*

5 Proofs of correctness

This section presents the main technical contribution of the paper, the proof of correctness of our entailment checking algorithm. The proof itself closely follows the structure of the previous section, filling in the technical details required to assert the statements of Theorem 1, on well-formedness, Theorem 2, on matching, and finally Theorem 3 on entailment checking.

Well-formedness Soundness of the well-formed condition *well-formed*(Σ), the first half of Theorem 1, is easily shown by noting that if a spatial conjunction Σ is satisfiable with respect to some stack and a heap, the formula *well-formed*(Σ) is also necessarily true with respect to the same stack.

Proposition 3. *Given $s, h \models \Sigma$ it follows $s \models$ well-formed(Σ).*

Proof. Let $\Sigma = S_1 * \cdots * S_n$. Since $s, h \models \Sigma$, there is a partition $h = h_1 * \cdots * h_n$ such that each $s, h_i \models S_i$. From the soundness definition it immediately follows that $s, h_i \models sound(S_i)$ for each predicate. For every pair S_i and S_j with $i < j$, if either $s \models empty(S_i)$ or $s \models empty(S_j)$, then trivially $s \models separated(S_i, S_j)$. Assume otherwise that $s \models \neg empty(S_i) \wedge \neg empty(S_j)$. It then follows that both $s(addr(S_i)) \in \operatorname{dom} h_i$ and $s(addr(S_j)) \in \operatorname{dom} h_j$. Since by construction h_i and h_j have disjoint domains, we have $s(addr(S_i)) \neq s(addr(S_j))$. This implies the fact that $s \models separated(S_i, S_j)$. □

For completeness of *well-formed*, the second half of Theorem 1, we prove a more general result. In particular if $s \models \mathit{well\text{-}formed}(\Sigma)$ and R is a set of *reserved* locations, disjoint from the necessarily allocated $\mathit{alloc}(\Sigma|s)$, then we show how to build a heap h such that $s, h \models \Sigma$ and $\operatorname{dom} h \cap R = \emptyset$.

Proposition 4. *Given a spatial conjunction Σ, a stack $s \models \mathit{well\text{-}formed}(\Sigma)$, and a finite set of locations R such that $\mathit{alloc}(\Sigma|s) \cap R = \emptyset$, there is a heap h such that $s, h \models \Sigma$ and $\operatorname{dom} h \cap R = \emptyset$.*

Proof. Let $\Sigma = S_1 * \cdots * S_n$. The proof is by induction on n and its base case, when $n = 0$, is trivially satisfied by $h = \emptyset$.

For $n > 1$, let $\Sigma' = \Sigma \setminus S_1 = S_2 * \cdots * S_n$ and let $R' = R \cup \mathit{alloc}(S_1|s)$. By construction the formula $\mathit{well\text{-}formed}(\Sigma) \rightarrow \mathit{well\text{-}formed}(\Sigma')$ is valid so, in particular, we also have $s \models \mathit{well\text{-}formed}(\Sigma')$. Furthermore, since $s \models \mathit{separated}(S_1, S_j)$ for all $2 \leq j \leq n$, it follows that $\mathit{alloc}(S_1|s) \cap \mathit{alloc}(\Sigma'|s) = \emptyset$ and, moreover, we also obtain that $\mathit{alloc}(\Sigma'|s) \cap R' = \emptyset$. Inductively applying the proposition on Σ' and the set R' we obtain a heap h' such that $s, h' \models \Sigma'$ and $\operatorname{dom} h' \cap R' = \emptyset$. Let

$$h_1 = \begin{cases} \emptyset & \text{if } S_1 = \mathsf{emp} \\ \{s(x) \mapsto s(y)\} & \text{if } S_1 = \mathsf{next}(x, y) \\ \{s(x) \mapsto \ell_1, \ell_1 \mapsto \ell_2, \ldots, \ell_{s(n)-1} \mapsto s(y)\} & \text{if } S_1 = \mathsf{lseg}(x, y, n) \end{cases}$$

where, if needed, the set of locations $\{\ell_1, \ldots, \ell_{s(n)-1}\} \cap (R \cup \operatorname{dom} h') = \emptyset$; since $R \cup \operatorname{dom} h'$ is finite but there are infinitely many locations, it is always possible to find suitable values. It is clear that $s, h_1 \models S_1$ and, furthermore, $\operatorname{dom} h_1 \cap \operatorname{dom} h'$ so $h = h_1 * h'$ is well defined. From these it follows that both $s, h \models \Sigma$ and $\operatorname{dom} h \cap R = \emptyset$. □

Theorem 1 follows as a corollary of Propositions 3 and 4.

Matching and proving The following proposition is the main ingredient required to establish the soundness and completeness of the *match* function of Figure 1. The proof, although quite long and rather technical, follows the intuitive description from Section 4 about the behaviour of *match*. Each of the main cases in the proof corresponds, respectively, to the conditions on lines 10 and 12, when discarding empty predicates, line 14, when a either a successful or unsuccessful subtraction is performed, and finally line 16, when the base case of the recursion is reached.

The first three cases are further divided each in two sub-cases, one for the situation when the recursive call is successful and a proof of validity is established, and one for the situation when a counterexample is built. The final case, the base of the recursion, is also divided into three sub-cases: when not all predicates in Σ' have been matched, when all predicates in Σ' were consumed but not all in Σ, and finally when both Σ and Σ' have become empty.

Proof (of Theorem 2). Termination of the function follows since, at each recursive call, the length of either Σ or Σ' is reduced. This also establishes the fact

that there are $O(|\Sigma| + |\Sigma'|)$ recursive calls. Now, given that the function does terminate, the proof is by induction on the recursive definition of *match*.

Note that, during the inductive proof, the spatial conjunction $\hat{\Sigma} * \Sigma$ always remains invariant. When a predicate S is removed from Σ, we implicitly add it to $\hat{\Sigma}$, keeping track of the already matched fragment from the original antecedent. This also keeps $\Delta = \mathit{alloc}(\hat{\Sigma} * \Sigma)$ always invariant between calls.

- Suppose we reach line 10, with a predicate $S \in \Sigma$ such that $s \models \mathit{empty}(S)$. Recursively let $M' = \mathit{match}(s, \Delta, \Sigma \setminus S, \Sigma')$ and $M = \mathit{empty}(S) \wedge M'$. Since $s \models \mathit{empty}(S)$ we have $s \models M$ if and only if $s \models M'$.
  - if $s \models M'$, by induction the entailment $M' \wedge (\hat{\Sigma} * S) * (\Sigma \setminus S) \rightarrow \hat{\Sigma} * \Sigma'$ is valid. Since $M \rightarrow M'$ then $M \wedge \hat{\Sigma} * \Sigma \rightarrow \hat{\Sigma} * \Sigma'$ is also valid.
  - if $s \not\models M'$, by induction $s \not\models (\hat{\Sigma} * S) * (\Sigma \setminus S) \rightarrow \hat{\Sigma} * \Sigma'$, which is exactly the same as $s \not\models \hat{\Sigma} * \Sigma \rightarrow \hat{\Sigma} * \Sigma'$.
- Suppose we reach line 12 with a predicate $S' \in \Sigma$ such that $s \models \mathit{empty}(S')$. Recursively let $M' = \mathit{match}(s, \Delta, \Sigma, \Sigma' \setminus S')$ and also $M = \mathit{empty}(S') \wedge M'$. Again $s \models M$ if and only if $s \models M'$.
  - if $s \models M'$, by induction $M' \wedge \hat{\Sigma} * \Sigma \rightarrow \hat{\Sigma} * (\Sigma' \setminus S')$ is valid. To prove that $M \wedge \hat{\Sigma} * \Sigma \rightarrow \hat{\Sigma} * \Sigma'$ is also valid, take any pair $s', h \models M \wedge \hat{\Sigma} * \Sigma$. From the inductive entailment we have $s', h \models \hat{\Sigma} * (\Sigma' \setminus S')$ and from the fact that $s' \models \mathit{empty}(S')$ also $s', \emptyset \models S'$. Thus $s', h \models \hat{\Sigma} * \Sigma'$.
  - if $s \not\models M'$, by induction there is a heap h such that $s, h \models \hat{\Sigma} * \Sigma$ but $s, h \not\models \hat{\Sigma} * (\Sigma' \setminus S')$. If it were the case that $s, h \models \hat{\Sigma} * \Sigma'$, from the fact that $s \models \mathit{empty}(S')$ it would follow that $s, h \models \hat{\Sigma} * (\Sigma' \setminus S')$, contradicting the information from the inductive step. Thus $s, h \not\models \hat{\Sigma} * \Sigma'$.
- Suppose we reach line 13, with two of predicates $S \in \Sigma$ and $S' \in \Sigma'$, such that $s \not\models \mathit{separated}(S, S')$. We compute $(S'', D) := \mathit{subtract}(\Delta, S', S)$, and further suppose that $s \models \mathit{sound}(S'') \wedge D$ so that we reach the next recursive call at line 15. Recursively let $M' = \mathit{match}(s, \Delta, (\Sigma \setminus S), (\Sigma' \setminus S') * S'')$ and $M = \neg \mathit{separated}(S, S') \wedge \mathit{sound}(S'') \wedge D \wedge M'$. As before we have $s \models M$ if and only if $s \models M'$.
  - if $s \models M'$, by induction $M' \wedge (\hat{\Sigma} * S) * (\Sigma \setminus S) \rightarrow (\hat{\Sigma} * S) * ((\Sigma' \setminus S') * S'')$ is valid. To prove that $M \wedge \hat{\Sigma} * \Sigma \rightarrow \hat{\Sigma} * \Sigma'$ is also valid, now take any pair $s', h \models M \wedge \hat{\Sigma} * \Sigma$. Since $\hat{\Sigma} * \Sigma = (\hat{\Sigma} * S) * (\Sigma \setminus S)$, from the inductive entailment after some rearrangement $s', h \models \hat{\Sigma} * (\Sigma' \setminus S') * (S * S'')$ and from Proposition 1 also $s', h \models \hat{\Sigma} * (\Sigma' \setminus S') * S'$. Thus $s', h \models \hat{\Sigma} * \Sigma'$.
  - if $s \not\models M'$, by induction after some rearrangement there is a heap h such that $s, h \models \hat{\Sigma} * \Sigma$ but $s, h \not\models \hat{\Sigma} * (\Sigma' \setminus S') * (S * S'')$. Partition the heap $h = h_1 * h_2$ such that $s, h_1 \models S$ and $s, h_2 \models \hat{\Sigma} * (\Sigma \setminus S)$. If it were the case that $s, h \models \hat{\Sigma} * \Sigma'$, from the second item on Proposition 1 it follows that $s, h \models \hat{\Sigma} * (\Sigma' \setminus S') * (S * S'')$, contradicting the inductive step. We therefore have that $s, h \not\models \hat{\Sigma} * \Sigma'$
- Suppose again we reach line 13, with two colliding predicates S and S'; we compute $(S'', D) := \mathit{subtract}(\Delta, S', S)$; but this time $s \not\models \mathit{sound}(S'') \wedge D$ so we reach line 16 with non-empty Σ and Σ', returning $M = \bot$. Since $s \not\models M$ we have to show that $s \not\models \hat{\Sigma} * \Sigma \rightarrow \hat{\Sigma} * \Sigma'$.

From Proposition 2 there is a heap h_1 such that $s, h_1 \models S$ and for any extension h, i.e. $h_1 \subseteq h$, we have $s, h \not\models \hat{\Sigma} * (\Sigma' \setminus S') * S'$. Applying Proposition 4 with $R = \operatorname{dom} h_1$, it is possible to obtain another heap h_2 such that $s, h_2 \models \hat{\Sigma} * (\Sigma \setminus S)$ and $\operatorname{dom} h_1 \cap \operatorname{dom} h_2 = \emptyset$. Let $h = h_1 * h_2$, from this it follows that $s, h \models \hat{\Sigma} * \Sigma$, and we already knew that $s, h \not\models \hat{\Sigma} * \Sigma'$.

– Finally suppose that we reach line 16, with no remaining pairs of colliding predicates in Σ and Σ'. We may find ourselves in several situations:
  - $\Sigma' \neq \emptyset$, so there is a $S' \in \Sigma'$ with $s \not\models \mathit{empty}(S')$, but it does not collide with any $S \in \Sigma$, so the function returns $M = \bot$ and we have to prove that $s \not\models \hat{\Sigma} * \Sigma \to \hat{\Sigma} * \Sigma'$. If S collides with some predicate in $\hat{\Sigma}$, then the consequence is immediately unsatisfiable and from Proposition 4 we obtain a model for $\hat{\Sigma} * \Sigma$. Otherwise let $R = \{s(\mathit{addr}(S'))\}$, since S' does not collide with anything in $\hat{\Sigma} * \Sigma$, we have $R \cap \mathit{alloc}(\hat{\Sigma} * \Sigma | s) = \emptyset$ and again from Proposition 4 there is a h such that $s, h \models \hat{\Sigma} * \Sigma$ and $s(\mathit{addr}(S')) \notin \operatorname{dom} h$. Since $s(\mathit{addr}(S'))$ must be included, by necessity, on any model of $\hat{\Sigma} * \Sigma'$, it follows as we wanted that $s, h \not\models \hat{\Sigma} * \Sigma'$.
  - $\Sigma' = \emptyset$ but $\Sigma \neq \emptyset$, so there is a $S \in \Sigma$ with $s \not\models \mathit{empty}(S)$, the function returns $M = \bot$ and thus we have to prove that $s \not\models \hat{\Sigma} * \Sigma \to \hat{\Sigma}$. From Proposition 4 with $R = \emptyset$ there is a h such that $s, h \models \hat{\Sigma} * \Sigma$. Partition the heap $h = h_1 * h_2$ such that $s, h_1 \models \hat{\Sigma}$ and $s, h_2 \models \Sigma$. Since there is a non-empty S in Σ it must be the case that $h_2 \neq \emptyset$ and $h_1 \subset h$ is a strict subset. Because all our considered spatial predicates are precise, it therefore follows that $s, h \not\models \hat{\Sigma}$.
  - Both $\Sigma' = \emptyset$ and $\Sigma = \emptyset$, so the function returns $M = \top$. In this final case it is trivial that $s \models M$ and $M \wedge \hat{\Sigma} \to \hat{\Sigma}$ is valid. □

We are now ready to prove the termination and correctness of the main *prove* function as stated earlier in Theorem 3.

Proof (of Theorem 3). Termination is established since each iteration of the loop at line 4 strictly reduces the number satisfying models of Γ. Since there is only a finite number of distinct formulas that may be built by conjunctions of $\mathit{empty}(S)$, $\mathit{sound}(S)$, $\neg\mathit{separated}(S, S')$ and the side condition of $\mathit{subtract}(\Delta, S', S)$—the building blocks for the return value M of *match*—all combinations will be exhausted at some point.

For correctness we first note that, starting from line 1, it is established that the formula $\Gamma \to \Pi \wedge \mathit{well\text{-}formed}(\Sigma)$ is valid and, since later only more conjuncts are appended to Γ, this invariant is maintained throughout the execution.

If the formula $\Gamma \wedge \neg\Pi'$ in line 3 is satisfiable, then there is a stack s such that $s \models \Gamma$ but $s \not\models \Pi'$. From Proposition 4 there is a heap h such that $s, h \models \Pi \wedge \Sigma$ but, since it already fails on the pure part, $s, h \not\models \Pi' \wedge \Sigma'$ and the program reports that the entailment is invalid. Otherwise, if $\Gamma \wedge \neg\Pi'$ is unsatisfiable, it follows that $\Pi \wedge \Sigma \to \Pi'$ is valid. In order to show this take any $s', h \models \Pi \wedge \Sigma$, from Proposition 3 we have that $s' \models \Pi \wedge \mathit{well\text{-}formed}(\Sigma)$. It therefore must be the case that $s' \models \Pi'$ or s' would be a model of the unsatisfiable $\Gamma \wedge \neg\Pi'$.

To finalise we now prove that line 4 at the base of the loop always satisfies the invariants that if $\Gamma \wedge \Sigma \to \Sigma'$ is valid then also $\Pi \wedge \Sigma \to \Sigma'$ is. Just before

entering the loop we have $\Gamma = \Pi \wedge \mathit{well\text{-}formed}(\Sigma)$. Assuming $\Gamma \wedge \Sigma \rightarrow \Sigma'$ is valid take any $s', h \models \Pi \wedge \Sigma$, from Proposition 3 it follows that $s' \models \mathit{well\text{-}formed}(\Sigma)$ and therefore, from our assumption, $s', h \models \Pi' \wedge \Sigma'$.

If we enter the code of the loop we have $s \models \Gamma$ and $M = \mathit{match}(s, \Delta, \Sigma, \Sigma')$. If $s \not\models M$ from Theorem 2 there is a heap h such that $s, h \models \Pi \wedge \Sigma$ but, however, $s, h \not\models \Sigma'$, providing as required a counterexample for the entailment. Alternatively, if $s \models M$, from $\Gamma \wedge \neg M \wedge \Sigma \rightarrow \Sigma'$ we have to prove $\Pi \wedge \Sigma \rightarrow \Sigma'$. Take any $s', h \models \Pi \wedge \Sigma$, if $s', h \models M$ then again from Theorem 2 the formula $M \wedge \Sigma \rightarrow \Sigma'$ is valid, and $s', h \models \Sigma'$. Otherwise, if $s', h \not\models M$, from our previous assumption it would also follow that $s', h \models \Sigma'$.

We reach the final line if Γ becomes unsatisfiable and, since $\Gamma \wedge \Sigma \rightarrow \Sigma'$ would then be trivially valid, we prove as desired the validity of $\Pi \wedge \Sigma \rightarrow \Sigma'$. □

6 Experiments

We implemented our entailment checking algorithm in a tool called Asterix using Z3 as the pure theory back-end. Due to the current lack of realistic benchmarks making use of such theory features, we only report the running times of our new implementation against already published benchmarks from [22].

These are benchmarks with a significant number of repeated spatial atoms in the entailment, generated by "cloning" multiple copies of verification conditions obtained when running Smallfoot [5] against its own benchmark suite. They are particularly difficult for the unfolding implemented in slp [22] and the match function in Asterix. We observe a significant improvement, since our match function collects constraints that are potentially useful for other applications of match and relies on the efficiency of a highly optimised SMT solver.

Copies	Smallfoot	slp	Asterix
1	0.01	0.11	0.17
2	0.07	0.06	0.19
3	1.03	0.08	0.23
4	9.53	0.13	0.26
5	55.85	0.38	0.31
6	245.69	2.37	0.39
7	(64%)	20.83	0.54
8	(15%)	212.17	0.85
9	—	—	1.49
10	—	—	2.81

Acknowledgements This research was supported in part by the ERC project 308125 VeriSynth.

References

1. C. Barrett and C. Tinelli. CVC3. In *CAV*, pages 298–302, 2007.

2. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.
3. J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, number 3328 in LNCS, pages 97–109, 2004.
4. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.
5. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2006.
6. M. Botincan, M. J. Parkinson, and W. Schulte. Separation logic verification of c programs with an SMT solver. *Electr. Notes Theor. Comput. Sci.*, 254:5–23, 2009.
7. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4SMT solver. In *CAV*, pages 299–303, 2008.
8. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.
9. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.
10. B. Cook, C. Haase, J. Ouaknine, M. J. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*, pages 235–249, 2011.
11. 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.
12. P. Cousot, R. Cousot, and L. Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In *FOSSACS*, pages 456–472, 2011.
13. L. de Moura and N. Bjørner. Model-based theory combination. *Electron. Notes Theor. Comput. Sci.*, 198(2), 2008.
14. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
15. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3), 2005.
16. D. Distefano and M. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, pages 213–226, 2008.
17. R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, pages 161–177, 2009.
18. B. Dutertre and L. D. Moura. The Yices SMT solver. Technical report, Computer Science Laboratory, SRI International, 2006.
19. C. Haase, S. Ishtiaq, J. Ouaknine, and M. Parkinson. SeLoger: A tool for graph-based reasoning in separation logic. In *CAV*, 2013.
20. R. Iosif, A. Rogalewicz, and J. Simacek. The tree width of separation logic with recursive definitions. In *CADE*, 01 2013.
21. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, pages 229–240, 2008.
22. J. A. Navarro Pérez and A. Rybalchenko. Separation Logic + Superposition Calculus = Heap Theorem Prover. In *PLDI*, pages 556–566, 2011.
23. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier, 2001.
24. R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using SMT. In *CAV*, 2013.
25. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.