Separation Logic Modulo Theories

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

[1] University College London
[2] Technische Universität München

Abstract. Logical reasoning about program data often requires dealing with heap structures as well as scalar data types. Recent advances in Satisfiability Modular Theory (SMT) already offer efficient procedures for dealing with scalars, yet they lack any support for dealing with heap structures. In this paper, we present an approach that integrates Separation Logic—a prominent logic for reasoning about list segments on the heap—and SMT. We follow a model-based approach that communicates aliasing among heap cells between the SMT solver and the Separation Logic reasoning part. An experimental evaluation using the Z3 solver indicates that our approach can effectively put to work the advances in SMT for dealing with heap structures. This is the first decision procedure for the combination of separation logic with SMT theories.

1 Introduction

Satisfiability Modulo Theory (SMT) solvers play an important role for the construction of abstract interpretation tools [11, 12]. They can efficiently deal with relevant logical theories of various scalar data types, e.g., fixed length bit-vectors and numbers, as well as uninterpreted functions and arrays [1, 7, 14, 15, 19]. However, dealing with programs that manipulate heap-allocated data structures using pointers exposes limitations of today's SMT solvers.

For example, SMT does not support separation logic—a promising logic for dealing with programs that manipulate the heap following a certain discipline [24]. Advances in the construction of such a solver could directly boost a wide range of separation logic based verifiers: manual/tool assisted proof development [18, 20, 25], extended static checking [5, 17], and automatic inference of heap shapes [2, 8, 16, 26].

In this paper we present a method for extending an STM solver with separation logic with list segment predicate [3], which is a frequently used instance of separation logic used by the majority of existing tools. Our method decides entailments of the form $\Pi \wedge \Sigma \rightarrow \Pi' \wedge \Sigma'$. Here, Π and Π' are *arbitrary* theory assertions supported by SMT, while Σ and Σ' are spatial conjunctions of pointer predicates $\mathsf{next}(x, y)$ and list segment predicates $\mathsf{lseg}(x, y)$. Symbols occurring in the spatial conjunctions can also occur in Π and Π'.

The crux of our method lies in an interaction of the model based approach to combination of theories [13] and a so-called *match* function that we propose for establishing logical implication between a pair of spatial conjunctions. We use

models of Π, which we call stacks, to guide the process of showing that every heap that satisfies Σ also satisfies Σ'. In return, the match function collects an assertion that describes a set of stacks for which the current derivation is also applicable. This assertion is then used to take those stacks into account for which we have not proved the entailment yet. As a result, our method can benefit from the efficiency offered by SMT for maintaining a logical context keeping track of stacks for which the entailment is already proved.

In summary, we present (to the best of our knowledge) the first SMT based decision procedure for separation logic with list segments. Our main contribution is the entailment checking algorithm for separation logic combined with decidable theories, together with its correctness proof. Furthermore we provide an implementation of the algorithm using Z3 for theory reasoning, and an evaluation on micro-benchmarks.

The paper is organised as follows. A run of the algorithm is illustrated in Section 2. We give preliminary definitions in Section 3. Our method is described in Section 4. All proofs are presented in Section 5. We present an experimental evaluation in Section 6. Conclusions are finally presented in Section 7.

Related work Our method is directly inspired by a theorem prover for separation logic with list segments [23] based on paramodulation techniques [22] to deal with equality reasoning. An approach that turned out quite advantageous compared to SmallFoot-based proof systems previously developed.

While [23] only deals with equalities, the work in this paper supports arbitrary SMT theory expressions in the entailment. Theory extensions of paramodulation are still an open problem—even state-of-the-art first order provers deliver poor performance on problems with linear arithmetic—so it is not evident how to extend [23] with theory reasoning. Similarly, it is unclear how to extend SmallFoot or jStar to obtain a decision procedure with rich theory reasoning.

Our *match* function can be seen as a generalisation of the unfolding inferences, geared towards interaction with the logical context of an SMT solver, rather than literals in a clausal representation of the entailment problem. Last but not least, on that previous work the combination with paramodulation is given by a quite complex inference system, at a level of detail which would not accessible through a black-box SMT prover. The original proof system for list segments [3, 4] gives a starting point to the design of our *match* function. However, while the proof system needs to branch and perform case reasoning during proof search, the *match* function is a deterministic, linear pass over the spatial conjuncts.

Recently, entailment between separation logic formulas where Π and Π' are conjunctions of (dis-)equalities was shown to be decidable in polynomial time [10]. While we are primarily interested reasoning about rich theory assertions describing stacks, exploration of this polynomial time result is an interesting direction for future work. Regarding an Nelson-Oppen combination of decision procedures [21], we see an algorithm following this combination approach as an interesting and difficult question for the future work. A direct application of such theory combination does not work, since it requires a satisfiability checker for sets of (possibly negated) spatial conjunctions. The interplay of conjunction,

negation and spatial conjunction is likely to turn this into a PSPACE problem. In contrast, the spatial reasoning in our approach has linear complexity, thus shifting the computational complexity to the SMT prover instead.

Chin et al. [9] present a fold/unfold mechanism to deal with user-specified well-founded recursive predicates. Due to such a general setting, it does not provide completeness. Our logic is more restrictive, allowing to develop a complete decision procedure. Similarly, Botinčan et al. [6] rely on a SmallFoot based proof system which, although does not guarantee completeness on the fragment we consider, is able to deal with user provided inference and rewriting rules.

2 Illustration

In this section we illustrate our algorithm using a high-level description and a simple example. To this end we prove the validity of the entailment:

$$\underbrace{c < e}_{\Pi} \wedge \underbrace{\mathsf{lseg}(a,b) * \mathsf{lseg}(a,c) * \mathsf{next}(c,d) * \mathsf{lseg}(d,e)}_{\Sigma} \rightarrow \underbrace{\top}_{\Pi'} \wedge \underbrace{\mathsf{lseg}(b,c) * \mathsf{lseg}(c,e)}_{\Sigma'} .$$

Abstractly, the algorithm performs the following key steps. It symbolically enumerates models that satisfy Π and yield a satisfiable heap part for Σ in the antecedent. For each such assignment s the algorithm attempts to (symbolically) prove that each heap h satisfying the antecedent, i.e., $s, h \models \Pi \wedge \Sigma$ also satisfies the consequence, i.e., $s, h \models \Pi' \wedge \Sigma'$. Finally, we generalise the assignment s and use the corresponding assertion to prune further models of Π that would lead to similar reasoning steps as s. The entailment is valid if and only all models of the pure parts are successfully considered.

For our example we begin with the construction of the constraint that guarantees the satisfiability of the heap part of the antecedent. This constraint requires that each pair of spatial predicates in Σ is not colliding, i.e., if two predicates start from the same heap location then one of them represents an empty heap. A list segment, say $\mathsf{lseg}(a,b)$, represents an empty heap if its start and end locations are equal, i.e., if $a \simeq b$. A points-to predicates, say $\mathsf{next}(c,d)$, always represents a non-empty heap. For the predicates $\mathsf{lseg}(a,b)$ and $\mathsf{lseg}(d,e)$ the absence of collision is represented as $a \simeq d \rightarrow a \simeq b \vee d \simeq e$, i.e., if the start location a of the first predicate is equal to the start location d of the second predicate then either of the predicates represents an empty heap. The remaining pairs of predicates produce the following non-collision assertions.

$a \simeq a \rightarrow a \simeq b \vee a \simeq c$	$\mathsf{lseg}(a,b)$ and $\mathsf{next}(a,c)$
$a \simeq c \rightarrow a \simeq b \vee \bot$	$\mathsf{lseg}(a,b)$ and $\mathsf{next}(c,d)$
$a \simeq d \rightarrow a \simeq b \vee d \simeq e$	$\mathsf{lseg}(a,b)$ and $\mathsf{lseg}(d,e)$
$a \simeq c \rightarrow a \simeq c \vee \bot$	$\mathsf{lseg}(a,c)$ and $\mathsf{next}(c,d)$
$a \simeq d \rightarrow a \simeq c \vee d \simeq e$	$\mathsf{lseg}(a,c)$ and $\mathsf{lseg}(d,e)$
$c \simeq d \rightarrow \bot \vee d \simeq e$	$\mathsf{next}(c,d)$ and $\mathsf{lseg}(d,e)$

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

Next, we use an SMT solver to find a model for $\Pi \wedge \textit{well-formed}(\Sigma)$. If no such model exists the entailment is vacuously true. For our example, however, the solver finds the model $s = \{a \mapsto 0, b \mapsto 0, c \mapsto 0, d \mapsto 1, e \mapsto 1\}$.

We then symbolically show that for every heap h model of Σ is also a model of Σ'. We do this by showing that Σ and Σ' are matching, i.e., for each predicate in Σ' there is a corresponding 'chain' of predicates in Σ. The chain condition requires adjacent predicates to have a location in common, namely, the finish location of a predicate is equal to the start location of the next with respect to s.

Since matching only needs to deal with predicates representing non-empty heaps, we first normalise Σ and Σ' by removing spatial predicates that are empty in the given model s, i.e., we remove each list segment predicate whose start and finish locations are equal with respect to s. From Σ we remove $\mathsf{lseg}(a, b)$ since $s(a) = s(b) = 0$, and from Σ' we cannot remove anything.

Now we attempt to find a match for $\mathsf{lseg}(b, c) \in \Sigma'$ in the normalised antecedent $\mathsf{lseg}(a, c) * \mathsf{next}(c, d) * \mathsf{lseg}(d, e)$. The chain should start with $\mathsf{lseg}(a, c)$ since $s(a) = s(b)$. Since $\mathsf{lseg}(a, c)$ finishes at the same location as $\mathsf{lseg}(b, c)$ in every model, we are done with the matching for $\mathsf{lseg}(b, c)$. Since $\mathsf{lseg}(a, c)$ was used to construct a chain, we cannot consider it in the remaining matching steps (but only for the same model s). Next we compute matching for $\mathsf{lseg}(c, e) \in \Sigma'$ using the remaining predicates $\mathsf{next}(c, d) * \mathsf{lseg}(d, e)$ from Σ. We begin the chain using $\mathsf{next}(c, d)$ since it has the same start location as $\mathsf{lseg}(c, e)$. Since the finish location of $\mathsf{next}(c, d)$ is not equal to e with respect to s we still need to connect d and e. We perform this connection by an additional matching request that requires to match $\mathsf{lseg}(d, e)$ using the remaining predicates from Σ, i.e., using only $\mathsf{lseg}(d, e)$. Fortunately, this matching request can be trivially satisfied. Since all predicates of Σ' are matched, and all predicates in Σ were used for matching, we conclude that Σ and Σ' exactly match with respect to the current s.

The algorithm notices that from the model s only the assertion $a \simeq b$ was necessary to perform the matching. Hence, the model s is generalised to the assertion $U = (a \simeq b)$. We continue the enumeration of pure models for the antecedent, excluding those where $a \simeq b$. The SMT solver reports that $\Pi \wedge \textit{well-formed}(\Sigma) \wedge \neg U$ is not satisfiable. Hence we 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; while $f \colon X \rightharpoonup Y$ is a *partial function* with $\operatorname{dom} f \subseteq X$. We write $f_1 * \cdots * f_n$ to simultaneously denote the union $f_1 \cup \cdots \cup f_n$ of n functions, and assert that their domains are pairwise disjoint, i.e. $\operatorname{dom} h_i \cap \operatorname{dom} h_j = \emptyset$ when $i \neq j$. Given two functions $f \colon Y \to Z$ and $g \colon X \to Y$, we write $f \circ g$ to denote their composition, i.e. $(f \circ g)(x) = f(g(x))$ for every $x \in \operatorname{dom} g$. We sometimes write functions explicitly by enumerating their elements, for example $f = \{a \mapsto b, b \mapsto c\}$ is the function with $\operatorname{dom} f = \{a, b\}$ and such that $f(a) = b$ and $f(b) = c$.

Syntax of separation logic We assume a sorted language with both theory and uninterpreted symbols. Each function symbol f has an arity n and a signa-

ture $f\colon \tau_1 \times \cdots \times \tau_n \to \tau$, taking n arguments of respective sorts τ_i and returning an expression of sort τ. A constant symbol is a 0-ary function symbol. A *variable* is an uninterpreted constant symbol, and Var denotes the set of all variables in the language. Constant and function symbols are combined as usual, respecting their sorts, to build syntactically valid *expressions*. We use $x\colon \tau$ to denote an expression x of sort τ, and $\mathcal{L}$ to denote the set of all expressions in the language.

We assume that, among the available sorts, there are Int and Bool for, respectively, integer and boolean expressions. We refer to a function symbol of boolean range as a *predicate symbol*, and a boolean expression as a *formula*. We also assume the existence of a built-in predicate $\simeq\colon \tau \times \tau \to \mathsf{Bool}$ for testing equality between two expressions of the same sort; as well as standard theory symbols from the boolean domain, that is: conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), truth ($\top$), falsity ($\bot$), implication ($\to$), bi-implication ($\leftrightarrow$) and first order quantifiers ($\forall$, $\exists$). Theory symbols for arithmetic may also be present, and we use nil as an alias for the integer constant 0.

Additionally, we also define *spatial symbols* to build expressions that describe properties about memory heaps. We have the spatial predicate symbols $\mathsf{emp}\colon \mathsf{Bool}$, $\mathsf{next}\colon \mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}$ and $\mathsf{lseg}\colon \mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}$ for, respectively, the empty heap, a points to relation, and acyclic-list segments; their semantics are described in the following section. Furthermore, we also have the symbol for *spatial conjunction* $*\colon \mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}$. A formula or an expression is said to be *pure* if it contains no spatial symbols.

Although in principle one can write spatial conjunctions of arbitrary boolean formulas, in our context we only deal with the case where each conjunct is a spatial predicate. So when we say a "spatial conjunction" what we actually mean is a "spatial conjunction of spatial predicates". Furthermore, at the meta-level, we treat a spatial conjunction $\Sigma = S_1 * \cdots * S_n$ as a multi-set of boolean spatial predicates, and write $|\Sigma| = n$ to denote the number of predicates in the conjunction. In particular we use set theory symbols to describe relations between spatial predicates and spatial conjunctions, which are always to be interpreted as *multi-set* operations. For example:

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

Semantics of separation logic Each sort τ is associated with a set of values, which we also denote by τ, usually according to their background theories; e.g. $\mathsf{Int} = \{\ldots, -1, 0, 1, \ldots\}$, and $\mathsf{Bool} = \{\bot, \top\}$. We use $\mathsf{Val} = \tau_1 \uplus \cdots \uplus \tau_n$ to denote the disjoint union of all values for all sorts in the language.

A *stack* is a function $s\colon \mathsf{Var} \to \mathsf{Val}$ mapping variables to values in their respective sorts, i.e. for a variable $v\colon \tau$ we have $s(v) \in \tau$. The domain of s is naturally extender over arbitrary pure expressions in $\mathcal{L}$ using an appropriate interpretation for their theory symbols, e.g. $s(1 + 2) = 3$. In our context, a *heap* corresponds to a partial function $h\colon \mathsf{Int} \rightharpoonup \mathsf{Val}$ mapping memory locations, represented as integers, to values.

```
1: function prove(Π ∧ Σ → Π′ ∧ Σ′)
2:    Γ := Π ∧ well-formed(Σ)
3:    while exists s ⊨ Γ do
4:       U := match(s, Σ, Σ, Σ′)
5:       if s ⊭ Π′ ∧ U then return invalid
6:       Γ := Γ ∧ ¬(Π′ ∧ U)
7:    return valid

8: function match(s, Σ̂, Σ, Σ′)
9:    if exists S ∈ Σ such that s ⊨ empty(S)
10:      return empty(S) ∧ match(s, Σ̂, Σ \ S, Σ′)
11:   if exists S′ ∈ Σ′ such that s ⊨ empty(S′)
12:      return empty(S′) ∧ match(s, Σ̂, Σ, Σ′ \ S′)
13:   if exists S ∈ Σ, S′ ∈ Σ′ such that s ⊨ match-step(Σ̂, S, S′)
14:      return match-step(Σ̂, S, S′) ∧ match(s, Σ̂, Σ \ S, (Σ′ \ S′) ∗ residue(S, S′))
15:   else
16:      return (Σ ≡ ∅) ∧ (Σ′ ≡ ∅)
```

Fig. 1. Model driven entailment checker

Given a stack s, a heap h, and a formula F we inductively define the satisfaction relation of separation logic, denoted $s, h \models F$, as:

$$
\begin{array}{ll}
s, h \models \Pi & \text{if } \Pi \text{ is pure and } s(\Pi) = \top, \\
s, h \models \mathsf{emp} & \text{if } h = \emptyset, \\
s, h \models \mathsf{next}(x, y) & \text{if } h = \{s(x) \mapsto s(y)\}, \\
s, h \models F_1 * F_2 & \text{if } h = h_1 * h_2 \text{ for some } h_1 \text{ and } h_2 \\
 & \qquad \text{such that } s, h_1 \models F_1 \text{ and } s, h_2 \models F_2.
\end{array}
$$

Semantics for the acyclic list segment is introduced through the inductive definition $\mathsf{lseg}(x, z) \equiv (x \simeq z \wedge \mathsf{emp}) \vee (x \not\simeq z \wedge \exists y.\, \mathsf{next}(x, y) * \mathsf{lseg}(y, z))$. As an example consider $\{x \mapsto 1, y \mapsto 2\}, \{1 \mapsto 3, 3 \mapsto 2\} \models \mathsf{lseg}(x, y)$.

When $s, h \models F$ we say that the interpretation (s, h) is a *model* of the formula F. A formula is *satisfiable* if it admits at least one model, and *valid* if it is satisfied by all possible interpretations. Note, in particular, that an entailment $F \to G$ is valid if every model of F is also a model of G. Finally, for a formula F we write $s \models F$ if it is the case that, for every heap h, we have that $s, h \models F$ holds.

Note that nil is not treated in any special way by this logic. If one wants nil to regain its expected behaviour, i.e. *nothing* can be allocated at the nil address, it is enough to consider $\mathsf{next}(\mathsf{nil}, 0) * F$, where F is an arbitrary formula.

4 Decision procedure for list segments and SMT theories

In this section we define and describe the building blocks that, when put together as shown in the *prove* and *match* procedures of Figure 1, constitute a decision

procedure for entailment checking. The procedure works for entailments of the form $\Pi \wedge \Sigma \to \Pi' \wedge \Sigma'$, where both Π and Π' are pure formulas, with respect to any background theory supported by the SMT solver, and both Σ and Σ' are spatial conjunctions.

To abstract away the specifics of a spatial predicate S, we first define $addr(S)$ and $empty(S)$—respectively the *address* and the *emptiness condition* of a given spatial predicate—as follows:

S	$addr(S)$	$empty(S)$
emp	—	$\top$
$\mathsf{next}(x, y)$	x	$\bot$
$\mathsf{lseg}(x, y)$	x	$x \simeq y$

Intuitively, if the emptiness condition is true with respect to a stack-model s, the portion of the heap-model that corresponds to S *must* be empty. Alternatively, if the emptiness condition is false with respect to s, the value associated with its address *must* occur in the domain of any heap satisfying the spatial predicate. Formally: given $s \models empty(S)$ for a stack s, we have $s, h \models S$ if, and only if, the heap $h = \emptyset$; and if $s, h \models \neg empty(S) \wedge S$ then, necessarily, $s(addr(S)) \in \operatorname{dom} h$.

Well-formedness Before introducing the *well-formed* condition, occurring at line 2 of the algorithm in Figure 1, we first define the notion of *collision* between spatial predicates. Given any two spatial predicates S and S', the formula

$$collide(S, S') = \neg empty(S) \wedge \neg empty(S') \wedge addr(S) \simeq addr(S') \ .$$

states that two predicates collide if, with respect to a stack-model, they are both non-empty and share the same address. This would cause a problem if both S and S' occur together in a spatial conjunction, since they would assert that the same address is allocated at two disjoint—separated—portions of the heap.

Given a spatial conjunction $\Sigma = S_1 * \cdots * S_n$, the *well-formedness condition* is defined as the pure formula

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

stating that no pair of predicates in the spatial conjunction collide. As an example consider the spatial conjunction

$$\Sigma = \underbrace{\mathsf{next}(x, y)}_{S_1} * \underbrace{\mathsf{lseg}(x, z)}_{S_2} * \underbrace{\mathsf{next}(w, z)}_{S_3}$$

we obtain

$$\begin{aligned}
collide(S_1, S_2) &= (\top \wedge x \not\simeq z \wedge x \simeq x) = (x \not\simeq z) \\
collide(S_1, S_3) &= (\top \wedge \top \wedge x \simeq w) = (x \simeq w) \\
collide(S_2, S_3) &= (x \not\simeq z \wedge \top \wedge x \simeq w) = (x \not\simeq z \wedge x \simeq w) \\
well\text{-}formed(\Sigma) &= \neg(x \not\simeq z) \wedge \neg(x \simeq w) \wedge \neg(x \not\simeq z \wedge x \simeq w) = (x \simeq z \vee x \not\simeq w) \ .
\end{aligned}$$

That is, the formula is well-formed only when $x \simeq z$, so that the second predicate is empty, and $x \not\simeq w$, so that the first and third do not collide. In general, the well-formedness condition is quite important since, 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.*

Matching step We now proceed towards the introduction of the *match-step* condition, used at line 14 in Figure 1, which lies at the core of our matching procedure. For this we first define, given a spatial conjunction $\Sigma = S_1 * \cdots * S_n$ and an expression x, the *allocation condition*

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

which holds, with respect to a stack-model s, when a corresponding heap-model h for Σ would necessarily have to include $s(x)$ in its domain. Continuing from our previous example we have that

$$\mathit{alloc}(\Sigma, z) = (\top \wedge z \simeq x) \vee (x \not\simeq z \wedge z \simeq x) \vee (\top \wedge z \simeq w) = (z \simeq x \vee z \simeq w) \ .$$

That is, the value of z must be allocated in the heap if either $z \simeq x$, so it is needed to satisfy $\mathsf{next}(x, y)$, or $z \simeq w$ and it is needed to satisfy $\mathsf{next}(w, z)$. If otherwise the allocation condition is false, although it may occur, there is no actual need for z to be allocated in the domain of the heap.

Now, 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 $S' \in \Sigma'$, then portion of the heap that satisfies S should overlap with the portion of the heap that satisfies 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.

In the following definitions *residue* gives the precise value of the extra surplus, while *enclosed* specifies additional conditions which are necessary so that the model of S doesn't leak outside the model of S'.

S'	S	$\mathit{residue}(S, S')$	$\mathit{enclosed}(\Sigma, S, S')$
$\mathsf{next}(x', z)$	$\mathsf{next}(x, y)$	emp	$y \simeq z$
$\mathsf{lseg}(x', z)$	$\mathsf{next}(x, y)$	$\mathsf{lseg}(y, z)$	$\top$
$\mathsf{next}(x', z)$	$\mathsf{lseg}(x, y)$	emp	$\bot$
$\mathsf{lseg}(x', z)$	$\mathsf{lseg}(x, y)$	$\mathsf{lseg}(y, z)$	$y \not\simeq z \to \mathit{alloc}(\Sigma, z)$

The *matching step condition* is the formula

$$\mathit{match\text{-}step}(\Sigma, S, S') = \mathit{collide}(S, S') \wedge \mathit{enclosed}(\Sigma, S, S') \ .$$

To formalise our stated intuition, the following proposition articulates how the residue that is computed between two colliding predicates is indeed satisfied by the remaining heap surplus. The validity of this statement, as in the case of the subsequent two propositions, can be easily verified by inspection of the relevant definitions.

Proposition 1. *Given two spatial predicates S, S', a stack $s \models \mathit{collide}(S, S')$ and a heap h such that $s, h \models S'$, if there is a partition $h = h_1 * h_2$ for which $s, h_1 \models S$, it necessarily follows that $s, h_2 \models \mathit{residue}(S, S')$.*

Moreover, for any stack satisfying the matching step condition, we are free to replace S' in Σ' with the matched expression $S * \mathit{residue}(S, S')$. Formally we state the following proposition.

Proposition 2. *Given a stack $s \models \mathit{match\text{-}step}(\Sigma, S, S')$, where S and S' are spatial predicates, and S occurs in the spatial conjunction Σ, for any spatial conjunction Σ' containing S' we have that*

$$s \models (\Sigma' \setminus S') * S * \mathit{residue}(S, S') \to \Sigma'$$

Finally, we state that the enclosing condition is complete in the sense that, if it were not satisfied by a stack s, then one could build a counterexample for the matching $S * \mathit{residue}(S, S') \to S'$.

Proposition 3. *Given two spatial predicates S, S', a spatial conjunction Σ that contains S, a stack s and a two-part heap $h = h_1 * h_2$ such that $s, h_1 * h_2 \models \Sigma$ and $s, h_2 \models S * \mathit{residue}(S, S')$, if $s \models \mathit{collide}(S, S') \wedge \neg \mathit{enclosed}(\Sigma, S, S')$, then there is a h_2' such that $s, h_1 * h_2' \models \Sigma$ but $s, h_2' \not\models S * \mathit{residue}(S, S') \to S'$.*

As an example consider the case where $S = \mathsf{lseg}(x, y)$ and $S' = \mathsf{lseg}(x', z)$, such that $\mathit{residue}(S, S') = \mathsf{lseg}(y, z)$. Take some stack $s \models \mathit{collide}(S, S')$ and the heap $h_2 = \{s(x) \mapsto s(y), s(y) \mapsto s(z)\}$ as a model of $\mathsf{lseg}(x, y) * \mathsf{lseg}(y, z)$. From $s \models \neg \mathit{enclosed}(\Sigma, S, S')$ it follows that $s(x) \neq s(y)$ and the address $s(z)$ does not need to be allocated anywhere in $h = h_1 * h_2$. This allows us to patch and let $h_2' = \{s(x) \mapsto s(z), s(z) \mapsto s(y), s(y) \mapsto s(z)\}$, which is still a model of the pair $\mathsf{lseg}(x, y) * \mathsf{lseg}(y, z)$ but—due to the introduced cycle—not of $\mathsf{lseg}(x', z)$.

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 $s \models \hat{\Sigma} \to (\hat{\Sigma} \setminus \Sigma) * \Sigma'$. Initially called with $\hat{\Sigma}$ set to Σ, at the top level this is in fact equivalent to checking the validity of $s \models \Sigma \to \Sigma'$. During the execution process $\hat{\Sigma}$ will retain its initial value, Σ and Σ' carry the portions of the entailment that are left to match, while $\hat{\Sigma} \setminus \Sigma$ is the fragment already matched. As the function progresses, the conjunctions Σ and Σ' will become shorter, while the matched portion $\hat{\Sigma} \setminus \Sigma$ grows. If successful both Σ and Σ' will become empty, yielding at the end the trivial entailment $s \models \hat{\Sigma} \to \hat{\Sigma}$.

The function begins by inspecting Σ and Σ' to discard, at lines 10 and 12, any empty predicates with respect to s, and recursively calling itself to verify the rest of the entailment. After removing all such empty predicates, if a valid matching step is found, the predicate S' occurring in Σ' is replaced with $S * \mathit{residue}(S, S')$, so that S—which now occurs both in Σ and Σ'—can be moved to the matched part of the entailment in the recursive call at line 14.

If the function is successful, after reaching the bottom of the recursion at line 16 with both Σ and Σ' becoming empty, the return value collects a conjunction of all assumptions made on the values of stack. This allows to generalise the proof which works not only for the particular stack s, but for any stack satisfying the same assumptions. Otherwise, if the bottom of the recursion is reached with some portions still left to match, the function returns an unsatisfiable formula signalling the existence of a counterexample for the entailment. This behaviour is formalised in the following theorem, proved later in Section 5.

Theorem 2. *Given a pair of spatial conjunctions Σ, Σ' and a stack s such that $s \models \mathit{well\text{-}formed}(\hat{\Sigma})$, we have that:*

- *the procedure $\mathit{match}(s, \Sigma, \Sigma, \Sigma')$ always terminates with a result U,*
- *the execution requires $O(n)$ recursive steps, where $n = |\Sigma| + |\Sigma'|$.*
- *if $s \models U$ then the entailment $U \wedge \Sigma \to \Sigma'$ is valid, and*
- *if $s \not\models U$ then $s \not\models \Sigma \to \Sigma'$.*

The main *prove* function, which checks whether $\Pi \wedge \Sigma \to \Pi' \wedge \Sigma'$ is valid, begins with the pure formula $\Gamma := \Pi \wedge \mathit{well\text{-}formed}(\Sigma)$. An SMT solver iteratively finds models for Γ, which become candidate stack models 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 U generalising the conditions in which the entailment is valid, so the search may continue for stacks where U does not hold. The iterations proceed until either all possible stacks have been discarded, or a counterexample is found in the process. It is important to stress that the function does not enumerate all concrete models but, rather, the equivalence classes returned by *match*. Formally we state the following theorem, whose proof is given in Section 5.

Theorem 3. *Given two pure formulas Π, Π', and two spatial formulas Σ, Σ', we have that:*

- *the procedure $\mathit{prove}(\Pi \wedge \Sigma \to \Pi' \wedge \Sigma')$ always terminates, and*
- *the return value corresponds to the validity of $\Pi \wedge \Sigma \to \Pi' \wedge \Sigma'$.*

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 $\mathit{well\text{-}formed}(\Sigma)$, the first half of Theorem 1, can be easily shown by noting that if a spatial conjunction Σ is satisfiable with respect to some stack and a heap, the formula $\mathit{well\text{-}formed}(\Sigma)$ is also necessarily true with respect to the same stack.

Proposition 4. *Given a spatial conjunction Σ, a stack s, and a heap h, if we have $s, h \models \Sigma$, then also $s \models \mathit{well\text{-}formed}(\Sigma)$.*

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 $s, h_i \models S_i$. Given a pair of predicates S_i and S_j with $i < j$, if either $s \models \mathit{empty}(S_i)$ or $s \models \mathit{empty}(S_j)$, then trivially $s \models \neg \mathit{collide}(S_i, S_j)$.

Assume otherwise that $s \models \neg \mathit{empty}(S_i) \wedge \neg \mathit{empty}(S_j)$. It follows that both $s(\mathit{addr}(S_i)) \in \operatorname{dom} h_i$ and $s(\mathit{addr}(S_j)) \in \operatorname{dom} h_j$. Since by construction h_i and h_j have disjoint domains, we have $s(\mathit{addr}(S_i)) \neq s(\mathit{addr}(S_j))$. This implies the fact that $s \models \neg \mathit{collide}(S_i, S_j)$. □

For completeness of the well-formed condition $\mathit{well\text{-}formed}(\Sigma)$, the second half of Theorem 1, we prove a slightly more general result. In particular we show that if a stack $s \models \mathit{well\text{-}formed}(\Sigma)$ then it is possible to build a heap h such that $s, h \models \Sigma$. Furthermore, we show that such h is *conservative* in the sense that it only allocates addresses which are strictly necessary.

Proposition 5. *Given a spatial conjunction $\Sigma = S_1 * \cdots * S_n$ and a stack s such that $s \models \mathit{well\text{-}formed}(\Sigma)$, there is a heap h for which $s, h \models \Sigma$ and, furthermore, the domain $\operatorname{dom} h = \{\mathit{addr}(S_i) \mid s \models \neg \mathit{empty}(S_i)\}$.*

Proof. Consider the heap $h = h_1 * \cdots * h_n$ where each h_i is defined as follows:

- if $s \models \mathit{empty}(S_i)$ then $h_i = \emptyset$; otherwise
- if $s \models \neg \mathit{empty}(S_i)$ it follows that $S_i = \mathsf{next}(x, y)$ or $S_i = \mathsf{lseg}(x, y)$, in either case let $h_i = \{s(x) \mapsto s(y)\}$.

By construction $s, h_i \models S_i$ and, furthermore, if $s \models \neg \mathit{empty}(S_i, S_j)$ it follows that $\operatorname{dom} h_i = \{s(\mathit{addr}(S_i))\}$. From this we easily get as desired that the domain of the heap $\operatorname{dom} h = \{\mathit{addr}(S_i) \mid s \models \neg \mathit{empty}(S_i)\}$. Now, to prove that $s, h \models \Sigma$, we have only left to show that for any pair S_i, S_j with $i \neq j$ the domains or their respective heaplets are disjoint, i.e. $\operatorname{dom} h_i \cap \operatorname{dom} h_j = \emptyset$.

If either $s \models \mathit{empty}(S_i)$ or $s \models \mathit{empty}(S_j)$ the result is trivial. Otherwise assume that $s \models \neg \mathit{empty}(S_i) \wedge \neg \mathit{empty}(S_j)$. Since $s \models \mathit{well\text{-}formed}(\Sigma)$, and in particular also $s \models \neg \mathit{collide}(S_i, S_j)$, it follows that $s \not\models \mathit{addr}(S_i) \simeq \mathit{addr}(S_j)$. Namely the address values $s(\mathit{addr}(S_i)) \neq s(\mathit{addr}(S_j))$ and, thus, the domains of h_i and h_j are disjoint. □

Theorem 1 follows immediately as a corollary of Propositions 4 and 5.

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

Each case is further divided 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 last case, the base of the recursion, is divided into four sub-cases: the successful case when the matching is completed, the case in which all of Σ' is consumed but there are predicates in Σ left to match, the case in which there is a collision but the enclosure condition is not met, and finally the case in which there is no collision at all.

Proposition 6. *Given three spatial formulas $\hat{\Sigma}$, Σ, Σ', and a stack s such that $\Sigma \subseteq \hat{\Sigma}$, and $s \models$ well-formed($\hat{\Sigma}$); let U be the pure formula returned by match$(s, \hat{\Sigma}, \Sigma, \Sigma')$.*

- *If $s \models U$ then $U \wedge \hat{\Sigma} \to (\hat{\Sigma} \setminus \Sigma) * \Sigma'$ is valid and, otherwise*
- *if $s \not\models U$ there is a h such that $s, h \not\models \hat{\Sigma} \to (\hat{\Sigma} \setminus \Sigma) * \Sigma'$.*

Proof. The proof goes by induction, following the recursive definition of the *match* function.

- Suppose we reach line 10, with a predicate $S \in \Sigma$ such that $s \models \mathit{empty}(S)$. Recursively let $U' = \mathit{match}(s, \hat{\Sigma}, \Sigma \setminus S, \Sigma')$ and $U = \mathit{empty}(S) \wedge U'$. Since $s \models \mathit{empty}(S)$ it follows $s \models U \leftrightarrow U'$.
  - if $s \models U$, we want to show that $U \wedge \hat{\Sigma} \to (\hat{\Sigma} \setminus \Sigma) * \Sigma'$ is valid, so take any model $s', h \models U \wedge \hat{\Sigma}$. By induction we know the formula $U \wedge \hat{\Sigma} \to R$ is valid, where $R = (\hat{\Sigma} \setminus (\Sigma \setminus S)) * \Sigma' = (\hat{\Sigma} \setminus \Sigma) * S * \Sigma'$. It follows therefore follows that $s', h \models (\hat{\Sigma} \setminus \Sigma) * S * \Sigma'$. Since $s \models \mathit{empty}(S)$, there is nothing allocated in h for S and, thus, $s', h \models (\hat{\Sigma} \setminus \Sigma) * \Sigma'$.
  - if $s \not\models U'$, by induction there is a heap h such that $s, h \models \hat{\Sigma}$ but, at the same time, $s, h \not\models (\hat{\Sigma} \setminus \Sigma) * S * \Sigma'$. Again, since $s, \emptyset \models S$, it must be the case that $s, h \not\models (\hat{\Sigma} \setminus \Sigma) * \Sigma'$. (Otherwise you get a contradiction.)
- Suppose we reach line 12 with a predicate $S' \in \Sigma$ such that $s \models \mathit{empty}(S')$. Recursively let $U' = \mathit{match}(s, \hat{\Sigma}, \Sigma, \Sigma' \setminus S')$ and $U = \mathit{empty}(S') \wedge U'$. Again we have $s \models U \leftrightarrow U'$.
  - if $s \models U'$, we want to show that $U' \wedge \hat{\Sigma} \to (\hat{\Sigma} \setminus \Sigma) * \Sigma'$ is valid, so take any model $s', h \models U \wedge \hat{\Sigma}$. By induction we know $U \wedge \hat{\Sigma} \to (\hat{\Sigma} \setminus \Sigma) * (\Sigma' \setminus S')$ is valid and, thus, we also get that $s', h \models (\hat{\Sigma} \setminus \Sigma) * (\Sigma' \setminus S')$. Again, from $s' \models \mathit{empty}(S')$ and $s', \emptyset \models S'$ it follows $s', h \models (\hat{\Sigma} \setminus \Sigma) * (\Sigma' \setminus S') * S'$ or, equivalently, $s', h \models (\hat{\Sigma} \setminus \Sigma) * \Sigma'$.
  - if $s \not\models U'$, by induction there is a heap h such that $s, h \models \hat{\Sigma}$ but, at the same time, $s, h \not\models (\hat{\Sigma} \setminus \Sigma) * (\Sigma' \setminus S')$. Similarly $s, \emptyset \models S'$, so it must be the case that $s, h \not\models (\hat{\Sigma} \setminus \Sigma) * (\Sigma' \setminus S') * S'$ or, equivalently, $s, h \not\models (\hat{\Sigma} \setminus \Sigma) * \Sigma'$.
- Suppose we reach line 14, with two of predicates $S \in \Sigma$ and $S' \in \Sigma'$, such that the stack $s \models \mathit{match\text{-}step}(\hat{\Sigma}, S, S')$. Let $S'' = \mathit{residue}(S, S')$, recursively obtain $U' = \mathit{match}(s, \hat{\Sigma}, \Sigma \setminus S, (\Sigma' \setminus S') * S'')$ and let $U = \mathit{match\text{-}step}(S) \wedge U'$. As before we have $s \models U \leftrightarrow U'$.
  - if $s \models U$, we want to show that $U \wedge \hat{\Sigma} \to (\hat{\Sigma} \setminus \Sigma) * \Sigma'$ is valid. That is, any model $s', h \models U \wedge \hat{\Sigma}$ is also a model of $(\hat{\Sigma} \setminus \Sigma) * \Sigma'$. By induction we have that $U' \wedge \hat{\Sigma} \to R$ is valid, where the formula

$$R = (\hat{\Sigma} \setminus (\Sigma \setminus S)) * (\Sigma' \setminus S') * S'' = (\hat{\Sigma} \setminus \Sigma) * (\Sigma' \setminus S') * S * S'' \ .$$

Since $s', h \models U' \wedge \hat{\Sigma}$ it follows that $s', h \models R$. By Proposition 2, since $s' \models \mathit{match\text{-}step}(\hat{\Sigma}, S, S')$, we obtain that $s', h \models (\hat{\Sigma} \setminus \Sigma) * (\Sigma' \setminus S') * S'$ or, equivalently, $s', h \models (\hat{\Sigma} \setminus \Sigma) * \Sigma'$.

- if $s \not\models U$, by induction, there exists a heap h such that $s, h \models \hat{\Sigma}$ but, however, $s, h \not\models (\hat{\Sigma} \setminus \Sigma) * (\Sigma' \setminus S') * S * S''$. Partition $h = h_1 * h_2$ such that $s, h_1 \models \hat{\Sigma} \setminus S$ and $s, h_2 \models S$. Now note that, regardless of the value of S, letting $h'_2 = \{s(x) \mapsto s(y)\}$ and $h' = h_1 * h'_2$ we have that both $s, h'_2 \models S$ and $s, h' \models \hat{\Sigma}$. We claim that $s, h' \not\models (\hat{\Sigma}, \setminus \Sigma) * \Sigma'$.
  Assume by contradiction that $s, h' \models (\hat{\Sigma} \setminus \Sigma) * \Sigma'$, and partition now $h' = h_3 * h_4$ such that $s, h_3 \models (\hat{\Sigma} \setminus \Sigma) * (\Sigma' \setminus S')$ and $s, h_4 \models S'$. Because S and S' collide, it follows that $\operatorname{dom} h'_2 = \{s(\mathit{addr}(S))\} \subseteq \operatorname{dom} h_4$ and $h_4 = h'_2 * h_5$ for some remainder h_5. Then, by Proposition 1, $s, h_4 \models S * S''$ and $s, h_5 \models S''$. But $h = h_1 * h_2 = h_3 * h_2 * h_5$ would make a model of $(\hat{\Sigma} * \Sigma) * (\Sigma' \setminus S') * S * S''$, contradicting our inductive hypothesis.

– Suppose we reach line 16. We can find ourselves in several situations:

- $\Sigma' = \emptyset$, $\Sigma = \emptyset$, and the function returns $U = \top$. In this case it is trivial that $s \models U$ and $U \wedge \hat{\Sigma} \to (\hat{\Sigma} \setminus \emptyset) * \emptyset$ is valid.
- $\Sigma' = \emptyset$, there is a $S \in \Sigma$, and the function returns $U = \bot$. In this case $s \not\models U$, so we need to find a counterexample for the entailment. From Proposition 5 there is a heap h such that $s, h \models \hat{\Sigma}$. Partition $h = h_1 * h_2$ such that $s, h_1 \models (\hat{\Sigma} \setminus \Sigma)$ and $s, h_2 \models \Sigma$. Since S occurs in Σ, and at this point $s \not\models \mathit{empty}(S)$, it is necessarily the case that $s(\mathit{addr}(S)) \in \operatorname{dom} h_2$. In particular $h_2 \neq \emptyset$, and because $h = h_1 * h_2$, we obtain $s, h \not\models (\hat{\Sigma} \setminus \Sigma)$. Furthermore, since $\Sigma' = \emptyset$, this is equivalent to $s, h \not\models (\hat{\Sigma} \setminus \Sigma) * \Sigma'$.
- There is a $S' \in \Sigma'$, a $S \in \Sigma$ such that $s \models \mathit{collide}(S, S')$, and the function returns $U = \bot$. Since we did not end up on line 14, it must be the case that $s \not\models \mathit{enclosed}(\hat{\Sigma}, S, S')$. By Property 5 there is a heap h such that $s, h \models \hat{\Sigma}$. Partition $h = h_1 * h_2$ such that $s, h_1 \models (\hat{\Sigma} \setminus S)$ and $s, h_2 \models S$. Let $h'_2 = \{s(x) \mapsto s(y)\}$ and $h' = h_1 * h'_2$; since $s \models \neg \mathit{empty}(S)$ we have that $s, h'_2 \models S$ and $s, h' \models \hat{\Sigma}$.
  If it turns out that $s, h' \not\models (\hat{\Sigma} \setminus \Sigma) * \Sigma'$ we are done. Assume otherwise that $s, h' \models (\hat{\Sigma} \setminus \Sigma) * \Sigma'$ and partition the heap $h' = h_3 * h_4$ such that $s, h_3 \models (\hat{\Sigma} \setminus \Sigma) * (\Sigma' \setminus S')$ and $s, h_4 \models S'$. Since the predicates S, S' collide and are non-empty, it follows that the address $\{s(\mathit{addr}(S))\} = \operatorname{dom} h'_2 \subseteq \operatorname{dom} h_4$ and, therefore, $h_4 = h'_2 * h_5$ for some remainder h_5. By Proposition 1 it follows that $s, h_4 \models S * S''$ and $s, h_5 \models S''$. Since $h' = h_3 * h'_2 * h_5$ it follows then that $s, h_3 * h_5 \models (\hat{\Sigma} \setminus S)$. By Proposition 3 there is a h_6 such that $s, h_3 * h_5 * h_6 \models \hat{\Sigma}$ but $s, h_5 * h_6 \not\models S'$. However, since $s, h_3 \models (\hat{\Sigma} \setminus \Sigma) * (\Sigma' \setminus S')$, it follows that $s, h_3 * h_5 * h_6 \not\models (\hat{\Sigma} \setminus \Sigma) * \Sigma'$. The heap $h_3 * h_5 * h_6$ is a counterexample for the entailment.
- There is some $S' \in \Sigma'$ and $s \not\models \mathit{collide}(S, S')$ for all $S \in \Sigma$, thus the function returns $U = \bot$. By Property 5 there is a heap h such that $s, h \models \hat{\Sigma}$. Partition $h = h_1 * h_2$ into two parts such that $s, h_1 \models (\hat{\Sigma} \setminus \Sigma)$ and $s, h_2 \models \Sigma$. Since S' does not collide with any predicate in Σ, it follows that $s(\mathit{addr}(S')) \notin \operatorname{dom} h_2$, in particular $s, h_2 \not\models \Sigma'$. From this it follows that $s, h_1 * h_2 \not\models (\hat{\Sigma} \setminus \Sigma) * \Sigma'$. □

The correctness of the *match* procedure, formally stated previously in Theorem 2, follows as a corollary of this proposition for the case when $\hat{\Sigma} = \Sigma$. Termination of the procedure can also be easily verified since, at the recursive calls in lines 10 and 14 the size of the third argument decreases and, when it stays the same at the recursive call in line 12, the size of the fourth argument decreases. This same termination argument also shows that the number of recursive steps is in fact linear in the size of Σ and Σ'.

Finally we are ready to prove the termination and correctness of the main *prove* procedure as stated earlier in Theorem 3. Specifically, we'll show that the procedure returns valid if, and only if, the entailment $\Pi \wedge \Sigma \rightarrow \Pi' \wedge \Sigma'$ supplied as argument is indeed valid.

Proof (of Theorem 3). Termination can be established since at each iteration of the loop at line 3, the number satisfying models of Γ is being strictly reduced. Since there is only a finite number of formulas that can be built by combinations of $\mathit{empty}(S)$ and $\mathit{match\text{-}step}(\hat{\Sigma}, S, S')$—the building blocks for U—all suitable combinations should be exhausted at some point.

For correctness we now prove that line 3 at the base of the loop always satisfies the invariants:

1. $\Gamma \rightarrow \Pi \wedge \mathit{well\text{-}formed}(\Sigma)$, and
2. if $\Gamma \wedge \Sigma \rightarrow \Pi' \wedge \Sigma'$ is valid then also $\Pi \wedge \Sigma \rightarrow \Pi' \wedge \Sigma'$ is.

The first invariant can be easily verified by inspecting the code and noting that at the beginning $\Gamma = \Pi \wedge \mathit{well\text{-}formed}(\Sigma)$, and later only more conjuncts are appended to Γ.

For the second invariant, right before entering the loop we have that $\Gamma = \Pi \wedge \mathit{well\text{-}formed}(\Sigma)$. So, assuming that $\Pi \wedge \mathit{well\text{-}formed}(\Sigma) \wedge \Sigma \rightarrow \Pi' \wedge \Sigma'$ is valid, take any $s', h \models \Pi \wedge \Sigma$, from Proposition 4 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 that $s \models \Gamma$ and start by letting $U = \mathit{match}(s, \Sigma, \Sigma, \Sigma')$. If $s \not\models \Pi' \wedge U$, then either we have that $s \not\models \Pi'$—from Proposition 5 there is a heap h such that $s, h \models \Pi \wedge \Sigma$ but $s, h \not\models \Pi'$—or $s \not\models U$—in which case from Theorem 2 there is a h such that $s, h \models \Pi \wedge \Sigma$ but $s, h \not\models \Sigma'$. In either case the entailment is invalid and the procedure correctly reports this.

Alternatively, if $s \models \Pi' \wedge U$, from $\Gamma \wedge \neg(\Pi' \wedge U) \wedge \Sigma \rightarrow \Pi' \wedge \Sigma'$ we have to prove that $\Pi \wedge \Sigma \rightarrow \Pi' \wedge \Sigma'$. Take any $s', h \models \Pi \wedge \Sigma$, if $s', h \models \Pi' \wedge U$ then from Theorem 2 the formula $U \wedge \Sigma \rightarrow \Sigma'$ is valid, and $s', h \models \Pi' \wedge \Sigma'$. Otherwise, if $s', h \not\models \Pi' \wedge U$, from our assumption we have as well $s', h \models \Pi' \wedge \Sigma'$. □

6 Experiments

We implemented our entailment checking algorithm in a tool called Asterix using Z3 as the theory back-end for testing the satisfaction of pure formulas and evaluating expressions against pure stack-models. The tool already accepts arbitrary theory expressions and assertions as part of the entailment formula.

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

Table 1. Running times in seconds while checking 'clones' of SmallFoot examples.

However, due to the current lack of realistic application benchmarks making use of such theory features, we only report the running times of this new implementation against already published benchmarks from [23].

Table 1 shows experiments that have a significant number of repeated spatial atoms in the entailment. They are particularly difficult for the unfolding implemented in slp and the match function in Asterix. Since our match function collects constraints that can potentially be useful for other applications of match, we observe a significant improvement.

7 Conclusion

We have presented a method for extending an SMT solver with separation logic using the list segment predicate. Our method decides entailments of the form $\Pi \wedge \Sigma \rightarrow \Pi' \wedge \Sigma'$, whose pure and spatial components may freely use arbitrary theory assertions and theory expressions, as long as they are supported by the back-end SMT solver. Furthermore, we provide a formal proof of correctness of the algorithm, as well as a experimental results with an implementation using Z3 as the theory solver.

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*, pages 178–192, 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*, pages 238–252, 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, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.
17. D. Distefano and M. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, pages 213–226, 2008.
18. R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, pages 161–177, 2009.
19. B. Dutertre and L. D. Moura. The Yices SMT solver. Technical report, Computer Science Laboratory, SRI International, 2006.
20. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, pages 229–240, 2008.
21. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.
22. 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.
23. J. A. N. Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, pages 556–566, 2011.
24. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
25. H. Yang. An example of local reasoning in bi pointer logic: the schorr-waite graph marking algorithm. In *SPACE workshop*, 2001.
26. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008.