A Decidable Fragment of Separation Logic

Seminar: Automated Reasoning

Florian Sextl[1]

[1]Technical University of Munich, sextl@in.tum.de

Abstract. This paper deals with a fragment of separation logic defined by Berdine et al. in [1]. It focuses on decidability of entailments and is limited to simple linked lists. With the restrictions provided by the fragment, a powerful rule is defined: *UnrollCollapse*. This rule allows arguing about lists in an efficient and trivially decidable manner. Based on these findings, a proof system is introduced, on top of which a decision procedure is defined. With, inter alia, this procedure, the fragment is proven to be decidable.

1. Introduction

Low-level imperative programs written in system programming languages are key to present day digital infrastructure. Nearly all system programming, i.e., programming of operating systems, device drivers and the like, is done at a rather low level. Low-level programming provides simple optimization possibilities and direct control over memory. Though, this kind of direct control has to be handled with great care as even small faults can lead to severe problems in the affected system. Thus, especially low-level programs used in critical infrastructure benefit from formal verification that proves them correct. Even though formal verification allows for this kind of safety by design, it is rather seldomly used in real-world projects due to its high complexity. To tackle this problem, several (semi-)automated reasoning methods have been introduced by the research community. One method meant to verify properties about the memory usage of low-level programs is separation logic. Separation logic is an extension of the Hoare logic first introduced by John C. Reynolds in [5]. It is one of the most useful logic systems to argue about low-level imperative programs in regard to concurrent memory safety [4], inter alia. Despite this extension, basic separation logic focuses on shared mutable data structures on the heap.

2. A Decidable Fragment of Separation Logic

In the following, a decidable fragment of the original basic separation logic is introduced and investigated. As such, it uses a subset of operators from separation logic and limits their use to focus on arguing about lists. This fragment was originally introduced by Berdine et al. in [1].

2.1. A Fragment of Separation Logic

The original paper about separation logic introduced an artificial low-level programming language with dedicated commands for memory allocation and deallocation as a basis for the logic itself. The commands ensure several structural properties, such as no memory leaks when executed on a valid start state. In addition, pointer arithmetic is prohibited for the fragment of separation logic explored in this work.

Separation logic formulæ comprise two parts: the so-called pure formula part Π that consist of predicates from a subset of propositional logic and the so-called spatial formula part Σ. In general, both parts argue about values from a specific value set V which contains a special symbol for a dangling pointer, namely *nil*. These values can be used as both addresses into the memory as well as real computational values. The memory model derived from the artificial programming language used with separation logic consists of a stack s that maps variables to heap addresses (denoted as $[\![v]\!]s = a$ where $v \in V$) and a heap h that maps finitely many L-values (i.e., values that are not *nil*) to arbitrary other values (cf. fig. 1). This mapping is also called the *points-to* relation and is a relation between a memory cell address and its content. Accordingly, a variable used as a left-hand side with this relation encodes a memory cell address, which is stored on the stack. With the points-to relation, reachability of a memory cell can be decided by computing the transitive closure of all points-to pairs within the heap. This fact enables precise arguing about the heap space.

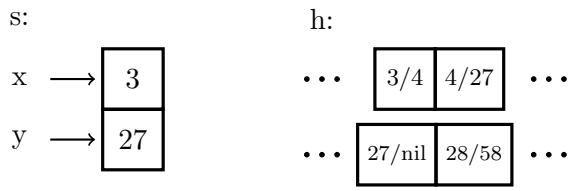
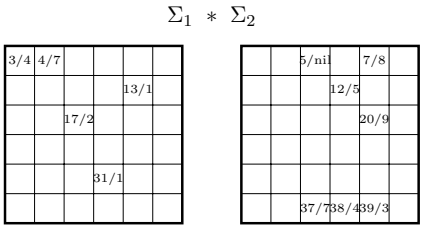


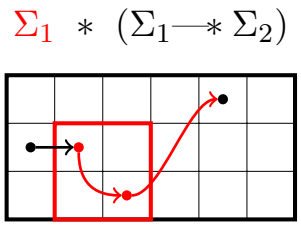
Figure 1: The memory model consists of a stack s and a heap h. The heap cells are described in the format *address/value*.

Together, the heap and the stack form a program state. Formulæ are then interpreted as predicates on these states. Thereby, the pure parts argue about equalities and inequalities of expressions, which are comprised of either variables or *nil*. For variables, equality is defined on their respective address according to the stack, i.e., for x, y variables $x = y \Leftrightarrow [\![x]\!]s = [\![y]\!]s$. These simple pure formulæ can also be negated or combined by logical conjunction with the neutral element *true*. In contrast to this, spatial formulæ

comprise simple points-to facts (written as $E_1 \mapsto E_2$) of expressions as well as *ls* structures, which are described below. These simple spatial formulæ can be combined by the separating conjunction ($*$) for heaps that have non-overlapping domains (cf. fig. 2(a)). The according neutral element is *emp* - the empty heap. Another operator on spatial formulæ is the so called separating implication ($-\!\!*$). Even though it is not used in the fragment, it is necessary to formulate specific properties and argue about meta-logic context. The separating implication $\Sigma_1 -\!\!* \Sigma_2$ denotes a subheap that, if combined with a heap that satisfies Σ_1, would satisfy Σ_2. With this definition, every spatial formula can be extended trivially to $\Sigma \equiv \Sigma_1 * (\Sigma_1 -\!\!* \Sigma)$ for an arbitrary, non-contradicting Σ_1 (cf. fig. 2(b)).



(a) A separating conjunction of two spatial formulæ Σ_1 and Σ_2 and fitting non-overlapping heap models. The heap cells are again labeled in the format *address/value*.



(b) A separating implication with the missing extension Σ_1. The arrows encode points-to relations that are necessary to satisfy Σ_2.

Figure 2: Formulæ about heap space.

Although *ls* structures are handled as simple spatial formulæ, they can be defined using other spatial formula components. To put it simple, *ls* structures represent linked lists on a low level. As such, they can be formulated as a series of points-to facts where each cell value is the address of the next cell. Linked lists are the core to the reasoning for which the fragment described in this work is meant. In general, *ls* structures could contain arbitrary values, yet in the following only lists without content are considered as this simplification does not prohibit a valid extension to lists with arbitrary content. The *ls* structure is defined by two expressions, of which the first evaluates to the address of the first cell in the list, whereas the second is the content of the last cell in the list (cf. fig. 3). This content may be another address or the dangling pointer. With this, the shortest possible list is the empty list $ls(E_1, E_2)$ where $E_1 = E_2$ for a given state. This list is equivalent to the empty stack *emp*. Although this notation could also be used to describe a circular list, this case is not allowed within the fragment. Circular lists would break unambiguousness within a spatial formula and are, therefore, only allowed by the combination of two distinct heap parts $ls(x, y) * y \mapsto x$. Any *ls* structure that ends with a dangling pointer is considered a full list, whereas a valid pointer at the end denotes a list segment. This also means, that there can not be any dangling pointer within a list. On another note, two non-overlapping lists are encoded as $ls(x, nil) * ls(y, nil)$ and two lists that share a common tail are encoded as $ls(x, z) * ls(y, z) * ls(z, nil)$. This kind

of explicit sharing of memory cells is the only allowed way of memory sharing for this fragment.

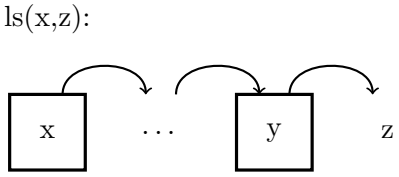


Figure 3: A simple linked list from the cell with address x to (possibly) another cell with address y that points to the next element z[1]. The arrows are again points-to relations encoded in the cells' values.

Formulæ can be evaluated under a certain state; this is then called a satisfaction and written as $s, h \models \Pi \mid \Sigma$. A satisfaction takes a state as a memory model and evaluates whether the formula is satisfied by this specific model. This property can be checked in linear time in correspondence to the combined size of the state and the formula. To achieve this, the formula has to be checked step by step until either a subformula is not satisfied for this state or no subformula remains to be checked. Both pure subformulæ and points-to facts can be checked directly with the state, whereas *ls* structures have to be followed through. A concrete decision algorithm would therefore first check the simple parts and then go through all *ls* structures destructively. This means, that each points-to relation is deleted from the heap model after it has satisfied a sublist. With this, circular lists and implicit sharing can be detected.

Based on these satisfactions, the more important entailments can be defined as a relation between two formulæ $\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'$. This relation is valid if and only if for all states s, h that satisfy $s, h \models \Pi \mid \Sigma$ (the antecedent) the satisfaction $s, h \models \Pi' \mid \Sigma'$ (the consequent) is also valid. Entailments come with a handful of simple properties that can be deduced directly from their definition. First of all, entailments are reflexive, even up to equality (e.g. $(x = y \wedge E = F) \mid x \mapsto E \vdash y \mapsto F$). On another note, inequality to *nil* can be implied by every valid points-to fact (e.g. $x \mapsto E \vdash x \neq nil \mid x \mapsto E$). Furthermore, points-to facts combined by the separating conjunction imply inequality of their addresses (e.g. $x \mapsto E * y \mapsto F \vdash x \neq y \mid x \mapsto E * y \mapsto F$). The definition of *ls* structures can also be trivially entailed (e.g. $x = y \mid emp \vdash ls(x, y)$ and $x \neq y \mid x \mapsto y \vdash ls(x, y)$ for lists of length zero and one respectively).

Entailments can be used to rewrite a formula to a better suited form before or after one command step in the whole separation logic. Hence, the fragment's focus on automation of entailment decision can be used as part of a bigger (semi-) automated proof procedure.

2.2. UnrollCollapse

One important property ensured by the fragment is *precision*. A formula is a precise predicate if for any given state, no more than one subheap satisfies it. That means, that

[1]This element may also not exist, i.e., it may also be a *nil* address.

for a satisfied precise predicate there is an unambiguous heap part which it corresponds to. Every points-to fact is by definition precise, as a heap storage can map cell addresses to values only surjectively. If the points-to fact is satisfied, there is exactly one memory cell with the corresponding address and value. Similarly, a separating conjunction of two precise formulæ is itself precise by definition. Precision of *ls* structures is slightly more complex as lists could be arbitrarily long and this length is not denoted directly in the *ls* structure. Yet, as precision is defined for a given state with a fixed heap, the length of a list can be determined directly from the points-to relation of this heap. With this information, every *ls* structure can be decomposed into a series of points-to facts connected via separating conjunctions. Hence, *ls* structures are also precise. To summarize, all spatial formulæ that can be validly built within the fragment are precise by default.

The validity of entailments can be decided naïvely by checking all satisfying models for the antecedent against the consequent. The number of these models depends directly on the concrete antecedent formula. Although formulæ consisting solely of points-to facts and separating conjunctions can only be satisfied by models with a heap of the same size as the formula[2], *ls* structures can be satisfied by arbitrarily long chains of pointers. Based on this observation, it is clear that formulæ without *ls* structures can be satisfied by only a finite number of models modulo substitution. Due to this, an entailment without *ls* structures in the antecedent can be decided in finite time, whereas an antecedent with *ls* structures introduces possibly infinitely many models that have to be checked and, accordingly, requires a different approach.

The idea behind the approach presented in the following is to abstract an arbitrary list by two basic cases: first the empty list and second the list with two elements as a representation for all longer lists[3]. This intuitive description can be formulated as the following proposition from [1]:

Proposition 1. *The following rule is sound:*

UnrollCollapse

$$\frac{\begin{array}{c}(\Pi \wedge E_1 = E_2) \mid \Sigma \quad \vdash \quad \Pi' \mid \Sigma' \\ (\Pi \wedge E_1 \neq E_2 \wedge x \neq E_2) \mid (E_1 \mapsto x * x \mapsto E_2 * \Sigma) \quad \vdash \quad \Pi' \mid \Sigma'\end{array}}{\Pi \mid ls(E_1, E_2) * \Sigma \quad \vdash \quad \Pi' \mid \Sigma'} \; x \notin fv(\Pi, E_1, E_2, \Sigma, \Pi', \Sigma')$$

In this context, *fv* denotes the set of variables occurring in the given formula parts. The *UnrollCollapse* rule encapsulates the idea that the inductive definition of a list[4] can be abstracted to non-inductive base cases. As an example, *UnrollCollapse* allows to prove

$$(a \neq c \wedge b \neq c) \mid a \mapsto b * ls(b, c) \quad \vdash \quad a \neq c \mid ls(a, c)$$

by proving

$$(a \neq c \wedge b \neq c \wedge b = c) \mid a \mapsto b \quad \vdash \quad a \neq c \mid ls(a, c)$$

[2]This follows directly from precision.

[3]A list of length one can be treated as a special case of a list of length two.

[4]The list is assumed to be a whole list not a mere segment and, therefore, ends in a dangling pointer.

and

$$(a \neq c \wedge b \neq c \wedge x \neq c) \mid (a \mapsto b * b \mapsto x * x \mapsto c) \quad \vdash \quad a \neq c \mid ls(a, c)$$

The antecedent of the first entailment contains a contradiction ($b \neq c \wedge b = c$). Thus, there exists no state that would satisfy this formula and, consequently, all satisfying models for the antecedent (none in this case) satisfy also the consequent. With this, the first entailment holds by definition. The second entailment can also be shown to hold as the points-to facts form an acyclic path from a to c and are, as such, semantically equivalent to the *ls* structure $ls(a, c)$.

In general, the *UnrollCollapse* rule enables more sophisticated decision procedures than naïve model checking. Before such a procedure can be found, the proposed soundness of *UnrollCollapse* needs to be proven.

Proof. For soundness the proposition has to be valid for true premises. For this reason, the premises are assumed to be valid:

$$(\Pi \wedge E_1 = E_2) \mid \Sigma \quad \vdash \quad \Pi' \mid \Sigma' \tag{1}$$

$$(\Pi \wedge E_1 \neq E_2 \wedge x \neq E_2) \mid (E_1 \mapsto x * x \mapsto E_2 * \Sigma) \quad \vdash \quad \Pi' \mid \Sigma' \tag{2}$$

for $x \notin fv(\Pi, E_1, E_2, \Sigma, \Pi', \Sigma')$. To prove the conclusion, the entailment is split and investigated for a fixed but arbitrary state s, h. That means, that on one hand the antecedent becomes another assumption $s, h \models \Pi \mid ls(E_1, E_2) * \Sigma$, whereas on the other hand the consequent becomes the new goal $s, h \models \Pi' \mid \Sigma'$. To prove this goal, a case distinction about the equality of E_1 and E_2 is necessary:

Case $[\![E_1]\!]s = [\![E_2]\!]s$:
In this case, assumption (1) holds and the goal can be directly concluded.

Case $[\![E_1]\!]s \neq [\![E_2]\!]s$:
In this case, h can be divided into two non-empty parts of which one models the list $ls(E_1, E_2)$ whereas the other models Σ: $h = h_{ls} * h_\Sigma$. With these two subheaps a new variable x can be introduced to formulate two new satisfactions: $s', h_{ls} \models E_1 \mapsto x * ls(x, E_2)$ and $s', h_\Sigma \models (\Pi \wedge E_1 \neq E_2) \mid \Sigma$ where $s' = [s|x \to l]$ for some address l.

For the next step, a new intuition about lists is required; i.e., a valid entailment with a *ls* structure of length two implies that the consequent is insensitive to the actual model of the list. This intuition is formalized in the following lemma (modified from [1]):

Lemma 1.

$$\begin{aligned} &\textit{If} && \Pi \mid ls^2(E_1, E_2) * \Sigma \quad \vdash \quad \Pi' \mid \Sigma' && (3)\\ &\textit{and} && s, h \models (\Pi \wedge E_1 \neq E_2 \wedge [\![E_2]\!]s \notin dom\,(h)) \mid \Sigma && (4)\\ &\textit{then} && s, h \models \Pi' \mid (ls(E_1, E_2) \mathrel{-\!\!*} \Sigma') \end{aligned}$$

The proof for this lemma is omitted here due to its high complexity but can be found in [1] Appendix A.4. Formula (3) denotes the valid entailment with the list of

length two ($ls^2(E_1, E_2)$). On the other hand, formula (4) describes a model s, h that satisfies the antecedent without the original list ($\Pi \mid \Sigma$) but with an other non-empty list ($E_1 \neq E_2 \wedge [\![E_2]\!]s \notin dom\,(h)$). If both (3) and (4) are valid, the model s, h satisfies also the consequent of (3) ($\Pi' \mid \Sigma'$) minus a fitting list model ($ls(E_1, E_2) \text{—}\!* \Sigma'$).

Now, lemma 1 can be applied to prove the remaining goal by instantiating its premises. Premise (3) requires a *ls* structure of length two. In general, a *ls* structure of length two can be encoded as $E_1 \mapsto E_2 * E_2 \mapsto E_3$ if all used expressions are not equal to each other[5]. Hence, assumption (2) satisfies the premise (3). Then again, the second premise (4) is satisfied by the model s', h_Σ, because $s', h_\Sigma \models (\Pi \wedge E_1 \neq E_2) \mid \Sigma$ holds as deduced before and $[\![E_2]\!]s \notin dom\,(h)$ holds as E_2 has to be a dangling pointer by definition. In general, a dangling pointer means that the expression evaluates to no valid address in the domain of the heap h and thereby also no valid address in any subheap like h_Σ. With both premises satisfied, the conclusion $s', h_\Sigma \models \Pi' \mid (ls(E_1, E_2) \text{—}\!* \Sigma')$ of lemma 1 holds. Together with $s', h_{ls} \models E_1 \mapsto x * ls(x, E_2) \equiv ls(E_1, E_2)$ the combined satisfaction $s, h \models \Pi' \mid \Sigma'$ holds by the definition of $\text{—}\!*$. This satisfaction is exactly the goal that had to be proven. □

In addition to soundness, the *UnrollCollapse* rule needs to be shown helpful for deciding validity of entailments, as well. The aforementioned intuition, that all entailments without a list are decidable, can be formalized as the following lemma:

Lemma 2. *For fixed* $\Pi, \Sigma, \Pi', \Sigma'$ *such that no subformula of* Σ *is of form* $ls(E_1, E_2)$*, checking* $\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'$ *is decidable.*

The proof for this lemma can be sketched out as the intuition about points-to facts and separating conjunctions from before. A similar proof sketch can be found in [1], p. 102, whereas the complete proof can be found in Appendix A.2 in the same source.

The lemma can then be used to prove the following corollary:

Corollary 1 (Validity Decidable)**.** *For fixed* $\Pi, \Sigma, \Pi', \Sigma'$*, checking* $\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'$ *is decidable.*

Proof. By applying *UnrollCollapse* repeatedly, any subformula of Σ containing a *ls* structure can be rewritten to a set of entailments which do not contain any *ls* structure. Due to lemma 2 all entailments which are part of the fragment are decidable. □

2.3. Proof System and Decision Procedure

In the following, *UnrollCollapse* is used as the core of a proof system. This system comprises several rules, which encode specific proof steps. The main proof idea is to first enrich the antecedent of an entailment with information that is sound to conclude from the formula alone and afterwards simplify both the antecedent and the consequent in parallel until an axiom is applicable. With this idea, all rules can be seen as functions that return a proof for their conclusion when given proofs for their premises. Rules

[5]The inequality of E_1 and E_2 follows directly from the separating conjunction.

without a premise are axioms and can be seen as constant proofs. *UnrollCollapse* is the only rule in this system with more than one premise. This means, that all other rules are mere rewriting rules that do not split up the computation of the proof. For simplicity, the complete rule system is omitted here but can be found in Appendix A.

As all rules except *UnrollCollapse* perform rather simple rewriting operations that can be proven sound directly from the definition of the fragment[6], soundness can be concluded for the whole proof theory. Soundness means in this case that on the one hand all entailments which can be derived from the system[7] are valid and on the other hand a tree of rule applications from an entailment at the root to axioms at the leafs is a proof for this entailment and can be found if and only if the entailment is valid.

On another note, the rule system is claimed to be complete for the whole fragment of separation logic by the original paper [1]. The authors implicitly assume this property and provide no explicit proof or even intuition on why this holds. Completeness of the proof system is thus also assumed in this work and can only be intuitively deduced from the set of rules itself. Although a thorough reasoning about the property of completeness is omitted here, it can be concluded that the aforementioned proof tree can be built for all valid entailments in the fragment. This is equivalent to the proposition that there exists no proof tree for all invalid entailments.

Based on this rule system, a simple recursive decision procedure $\texttt{PS}(g)$ can be formulated that either fails or returns a proof for the given goal entailment g. It is depicted as algorithm 1.

Algorithm 1 Decision Procedure

$Rs \leftarrow \{(r, p) \mid r \text{ a rule}, p \text{ a predicate}\}$ ▷ The rules with application conditions.

function $\texttt{PS}(g)$
  if $\exists (r, p) \in Rs.$ UnifiesWithConclusion$(g, r) \wedge p(g)$ **then**
    $p_0, \ldots, p_n \leftarrow$ premisesOf(r)
    return $r(\texttt{PS}(p_0), \ldots, \texttt{PS}(p_n))$
  else
    Fail

The algorithm picks in each step one rule in a nondeterministic manner, such that the current goal can be unified with the rule's conclusion. In addition, some rules require certain side conditions that have to be met for the rule to be applicable. If a rule exists, for which both the unification succeeds and the predicate holds for the goal, its premises are set as new goals and the procedure is called recursively. The recursive calls then either fail or return a proof for their subgoal. Afterwards, the rule is called with the proven premises and returns the proof for the original goal. Otherwise, if the rule is an axiom, there are no premises and the rule itself is returned as a proof. If more than one

[6] In addition, *UnrollCollapse* was proven sound in proposition 1.

[7] Deriving of entailments from the rules is similar to deriving a word from a formal grammar. By starting from an axiom and applying rules in a specific order, any valid entailment can be formulated modulo substitution of variable names and changes in the order.

rule would be applicable, their order is inconsequential as the rules are defined to be orthogonal in how they transform the formula.

The predicates appended to the rules range from simple assertions about not adding subformulæ twice to complex structural conditions. The most important structural predicate is the normal form for formulæ from the separation logic fragment. A formula from the fragment is in normal form if and only if it is maximally explicit, i.e., if its pure formula part consists of only pairwise inequalities of all variables and inequalities of all variables with *nil*. In addition, the spatial formula part is required to consist of only points-to facts from variables to expressions combined by separating conjunction. For example, the formula

$$x \neq y \mid x \mapsto y * y \mapsto nil$$

can be enriched to normal form as such:

$$(x \neq y \wedge x \neq nil \wedge y \neq nil) \mid x \mapsto y * y \mapsto nil$$

As *UnrollCollapse* allows to remove *ls* structures from the antecedent of an entailment, every antecedent can be enriched to normal form. The soundness of this step depends solely on the soundness of the rule system that is used for the enrichment, which was shown before. On top of this, the normal form is particularly useful as it allows for parallel simplifications in both the antecedent and the consequent. Hence, the proof system's rules can be divided into those that enrich the antecedent to become normal form and those that use the normal form to simplify both antecedent and consequent leading to applicability of an axiom in the long run. This corresponds to the basic idea the proof system is built upon.

On another note, the normal form can be used to argue about termination of the algorithm. Termination of `PS` can be shown with regard to the size of entailments. The size of an entailment is defined as a triple of the number of *ls* structures, the number of missing inequalities in the antecedent's pure part and the length of the whole entailment (defined as the number of simple subformulæ). If the antecedent of an entailment is in normal form, its size is minimal in regard to the first two measurements. The termination of the algorithm can therefore be proven trivially by showing that all rules decrease the lexicographical size of the entailment, that means have a lesser size for their premises as for their conclusion. Thereby, the enriching rules reduce the first two measurements and the simplification rules reduce the last measurement. In the long run, all three measurements are reduced to a minimum if the entailment is in fact valid. Due to the rule system being complete, termination can be concluded for all entailments.

Besides termination, a decision procedure needs also to fail if and only if the entailment is invalid. In the case of the `PS` algorithm, this means that the computation gets stuck when no fitting rule is found. In this case a so called *Bad Model* [1] can be constructed, which then serves as a countermodel for the entailment and shows its invalidity. A *Bad Model* for a formula is a state that is explicit in its mapping of variables to addresses via the stack and that satisfies the formula. There exists a *Bad Model* for every formula in normal form. As the procedure can only be stuck if the antecedent is already in normal

form[8], there exists a *Bad Model* for the antecedent of a stuck goal entailment. It can then be proven by case distinction on the consequent that there also exists a *Bad Model* for the antecedent that does not satisfy the consequent. The concrete proof is omitted here for simplicity as well as the corresponding proof for invalidity preservation by the proof system. Both can be found in [1] in their entirety and are directly based on the soundness and completeness of the rule system. Nonetheless, it can be concluded that `PS` fails if and only if it has found a disproof of the goal, i.e., the goal is invalid.

In conclusion, `PS` terminates for all entailments and has then either found a sound proof due to the soundness of the used rules or an implicit disproof for which a *Bad Model* can be constructed as a countermodel. With this properties, the algorithm can be concluded to be a valid decision procedure for the fragment of separation logic this work focuses on.

2.4. Examples

In the following, two simple examples are explained to show how the decision procedure can be used. Unchanged subformulæ are mostly omitted in each step for readability purposes. The rules used are given in brackets and refer to those in Appendix A. The first example is a valid entailment.

$$x \neq y \mid x \mapsto y * y \mapsto nil \quad \vdash \quad true \mid ls(x, nil)$$

The algorithm first enriches the antecedent to normal form. As the spatial formula part of the antecedent consist only of points-to facts, this can be achieved by adding inequalities about x and y (twice NILNOTLVAL), which follow directly from the points-to facts.

$$\Rightarrow (x \neq y \land x \neq nil \land y \neq nil) \mid \cdots \quad \vdash \quad \cdots$$

With this done, it is possible to apply a rule that simplifies *ls* structures in the consequent element by element (twice NONEMPTY*ls*).

$$\Rightarrow \cdots \mid y \mapsto nil \quad \vdash \quad true \mid ls(y, nil)$$

$$\Rightarrow \cdots \mid emp \quad \vdash \quad true \mid ls(nil, nil)$$

After simplifying the *ls* structure to the empty list ($ls(nil, nil)$[9]), it rewrites it to the empty heap (EMPTY*ls*) and proves the resulting entailment with an axiom (TAUTOLOGY).

$$\Rightarrow \cdots \mid emp \quad \vdash \quad true \mid emp$$

$$\Rightarrow valid \ \square$$

On the contrary, the second example is invalid as the separating conjunction ($x \mapsto nil * y \mapsto nil$) in the antecedent prohibits the equality ($x = y$) in the consequent.

$$true \mid x \mapsto nil * y \mapsto nil \quad \vdash \quad x = y \mid y \mapsto nil$$

[8] Otherwise, there exists a rule which enriches the formula as described above and is applicable.
[9] This would not be a valid list in the fragment but is considered acceptable for a subgoal.

The algorithm starts again with enriching the antecedent to normal form. For this it again adds inequalities about all used variables (*Partial, twice nilNotLval).

$$\Rightarrow (x \neq y \wedge x \neq nil \wedge y \neq nil) \mid \cdots \vdash \cdots$$

After this it removes the spatial duplicates $(y \mapsto nil)$ from both sides of the entailment (Frame). This is a sound step, as the same spatial formula part always entails itself and does not interfere with any other formula part.

$$\Rightarrow (x \neq y \wedge x \neq nil \wedge y \neq nil) \mid x \mapsto nil \vdash x = y \mid emp$$

The now resulting subgoal can not be unified with any rule and so the procedure gets stuck and fails. From this point a simple *Bad Model* can be found that satisfies the antecedent but contradicts the consequent. One such *Bad Model* is the following:

$$s = [x \to 5, y \to 23], h = [5 \to nil | 23 \to nil]$$

Due to failure preservation, the original goal is concluded to be invalid with the *Bad Model* as a disproof.

3. Conclusion

This paper summarized the concept and accomplishments of the fragment of separation logic introduced by Berdine et al. in [1]. The fragment focuses on enabling decidability for entailments with linked-lists. Due to the restraints of the fragment, arguing about the inductively defined *ls* structures can be broken down to arguing about two non-inductive cases, i.e., the empty list and the list with two elements. Based on this accomplishment, a sound and complete proof system was defined. This system became the core to the simple decision procedure `PS`.

In conclusion, the introduced fragment was proven to be decidable and provides a pattern on how more expressive fragments can be made decidable. In addition, it introduced a very simple decision procedure that runs extremely deterministic, as its proof tree is only nested on applications of *UnrollCollapse* and does not need to backtrack in case of getting stuck. Decision procedures with this characteristic are extremely useful and can be taken advantage of within bigger proof systems. As such, the fragment became the basis for the Smallfoot verification tool's core symbolic execution mechanism [2, 3]. Due to this, the fragment can be seen as an important step to arguing about separation logic in general.

References

[1] J. Berdine, C. Calcagno, and P. W. O'Hearn. A Decidable Fragment of Separation Logic. In K. Lodaya and M. Mahajan, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 3328 of *Lecture Notes in Computer Science*, pages 97–109, Berlin, Heidelberg, 2004. Springer.

[2] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic Execution with Separation Logic. In K. Yi, editor, *Programming Languages and Systems*, pages 52–68. Springer, Berlin, Heidelberg, 2005.

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, pages 115–137. Springer, Berlin, Heidelberg, 2006.

[4] P. W. O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019.

[5] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17th annual IEEE symposium on logic in computer science*, pages 55–74. IEEE Comput. Soc, 22-25 July 2002.

Appendix

A. Proof system

In the following the proof system's rules are depicted as they are formulated in [1]. The rules in appendix A.2 and A.3 as well as *UnrollCollapse* are meant to bring the antecedent in normal form, whereas the rules in appendix A.5 and A.6 simplify both the antecedent and the consequent to match one of the axioms from appendix A.1 in the long run.

A.1. Axioms

Tautology

$$\frac{}{\Pi \mid emp \vdash true \mid emp}$$

Contradiction

$$\frac{}{\Pi \wedge E \neq E \mid \Sigma \vdash \Pi' \mid \Sigma'}$$

A.2. Removing of equalities

Substitution

$$\frac{\Pi[E/x] \mid \Sigma[E/x] \vdash \Pi'[E/x] \mid \Sigma'[E/x]}{\Pi \wedge x = E \mid \Sigma \vdash \Pi' \mid \Sigma}$$

=ReflexiveL

$$\frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge E = E \mid \Sigma \vdash \Pi' \mid \Sigma'}$$

A.3. Introduction of inequalities

nilNotLval

$$\frac{\Pi \wedge E_1 \neq nil \mid E_1 \mapsto E_2 * \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid E_1 \mapsto E_2 * \Sigma \vdash \Pi' \mid \Sigma'}$$

*Partial

$$\frac{\Pi \wedge E_1 \neq E_3 \mid E_1 \mapsto E_2 * E_3 \mapsto E_4 * \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid E_1 \mapsto E_2 * E_3 \mapsto E_4 * \Sigma \vdash \Pi' \mid \Sigma'}$$

A.4. UnrollCollapse

UnrollCollapse

$$\frac{\Pi \wedge E_1 = E_2 \mid \Sigma \vdash \Pi' \mid \Sigma' \quad (\Pi \wedge E_1 \neq E_2 \wedge x \neq E_2) \mid E_1 \mapsto x * x \mapsto E_2 * \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid ls(E_1, E_2) * \Sigma \vdash \Pi' \mid \Sigma'} \; x \notin fv(\Pi, E_1, E_2, \Sigma, \Pi', \Sigma')$$

A.5. Simplifications that require no normal form

$$\text{=ReflexiveR}\;\frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \wedge E = E \mid \Sigma'} \qquad \text{Hypothesis}\;\frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge P \mid \Sigma \vdash \Pi' \wedge P \mid \Sigma'} \qquad \text{Empty}ls\;\frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \mid ls(E, E) * \Sigma'}$$

A.6. Simplifications that require normal form

$$\text{Frame}\;\frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid S * \Sigma \vdash \Pi' \mid S * \Sigma'} \qquad \text{NonEmpty}ls\;\frac{\Pi \wedge E_1 \neq E_3 \mid \Sigma \vdash \Pi' \mid ls(E_2, E_3) * \Sigma'}{\Pi \wedge E_1 \neq E_3 \mid E_1 \mapsto E_2 * \Sigma \vdash \Pi' \mid ls(E_1, E_3) * \Sigma'}$$