Size Does Matter: Two Certified Abstractions to Disprove Entailment in Intuitionistic and Classical Separation Logic

Clément Hurlin, François Bobot, Alexander Summers

Size Does Matter: Two Certified Abstractions to Disprove Entailment in Intuitionistic and Classical Separation Logic

Clément Hurlin
INRIA Sophia Antipolis – Méditerranée and University of Twente
clement.hurlin@inria.fr

François Bobot
LRI, Univ Paris-Sud and INRIA Saclay – Île-de-France
bobot@lri.fr

Alexander J. Summers
Imperial College London
alexander.j.summers@imperial.ac.uk

ABSTRACT

We describe an algorithm to disprove entailment between separation logic formulas. We abstract models of formulas by their size and check whether two formulas have models whose sizes are compatible. Given two formulas A and B that do not have compatible models, we can conclude that $A \not\vdash B$. We provide two different abstractions (of different precision) of models. Our algorithm is of interest wherever entailment checking is performed (such as in program verifiers).

1. INTRODUCTION

Separation logic [19] has recently made a dent in the verification of pointer manipulating programs. Its successes include simple pointer programs with parallelism [3], C programs [16], and object-oriented programs [9, 11]. Formulas in separation logic typically describe parts of the heap: in the context of program verifiers, method *preconditions* describe the heap space required by methods to execute correctly, and method *postconditions* describe the heap space passed back to the caller when the method returns. Separation logic comes in two flavours: *intuitionistic* separation logic is used to verify garbage-collected programs [18, 11] while *classical* separation logic is used to verify programs in which memory deallocation is performed manually [3, 16]. Contrary to intuitionistic separation logic, classical separation logic does not admit weakening, allowing one to reason about memory leaks.

We propose a novel algorithm to disprove entailment between formulas. We abstract formulas by the sizes of their possible models and use comparisons of these sizes to disprove entailment between formulas. Intuitively, to disprove an entailment $A \vdash B$, it suffices to show that there exists a model (i.e., a heap) of A whose size is smaller than the size of all models of B. We give two different ways of calculating sizes of models, which have differing complexities and precisions.

Our algorithm is of interest whenever entailment checking is performed, for example in program verifiers. In this particular context, since our algorithm's complexity is low, it can be used to quickly show that a method is not provable, perhaps because programmer-supplied specifications are incorrect.

IWACO '09, July 6 2009, Genova, Italy

2. BACKGROUND

We work with permission accounting separation logic based on that of Bornat et al. [5]. In particular, we reuse their structure of models to abstract over the permission models. A permission model $\mathscr{M}$ consists of:

1. A commutative semigroup with total binary operator $+_{\mathscr{M}}$.
2. A minimal permission, m_0, that satisfies the following property: $(\forall m)(m_0 +_{\mathscr{M}} m = m)$.
3. A maximal permission, m_W.
4. A total order on permissions, $\leq_{\mathscr{M}}$.
5. A subset of permissions which are defined to be *splittable*.
6. A pair of functions on permissions, $\mathsf{split1}$ and $\mathsf{split2}$, which define the two parts into which a splittable permission may be split. These functions must satisfy the following property: $(\forall m)(\mathsf{split1}(m) +_{\mathscr{M}} \mathsf{split2}(m) = m)$.

The maximal permission m_W represents full (and exclusive) access to a location, whereas the minimal permission represents no access. Permissions in between these two extremes will allow read access but not write access to a location. Some (in some models, all) permissions are *splittable*: they can be divided into two smaller (in the sense of permission ordering) permissions. In particular, the maximal permission (which denotes exclusive access) may be split into smaller permissions, none of which allow full access to the location. However, permissions may also be *combined* (by addition): in this way it is possible for a full permission to be regained, if all partial permissions are combined together.

Later, we define heaps that only contain *valid* permissions. Valid permissions are permissions that can be obtained by splitting the maximal permission. As a corollary, any valid permission m satisfies the following inequality: $m_0 <_{\mathscr{M}} m \leq_{\mathscr{M}} m_W$. Our model, however, includes permissions that are bigger than m_W, because we need to sum permissions above m_W. That is why $+_{\mathscr{M}}$ is a total operator.

Our paper's results have been established for two concrete permissions models: Boyland's fractional permissions [6, 5] and the counting model of Bornat et al. [5].

In the fractional permission model, permissions are rational numbers. The binary operator and the ordering are the usual $+$ and $\leq$ on rationals, the minimal permission is 0 and the maximal permission is 1. All permissions are splittable, and splitting simply divides permissions in half: $\mathsf{split1}(m) = \mathsf{split2}(m) = \frac{m}{2}$.

In the counting model, permissions were originally represented by integers, but we re-encode them in a different manner to allow the use of a standard order. We encode permissions as a pair

(n,i), where n is a natural number (in fact, only 0 or 1), and i is an integer. Intuitively, the first of the pair indicates whether this permission can be *split*, to give out further (read) permissions, while the second indicates a number of read permissions (positive to represent read permissions which have been obtained, and negative to represent those given out). The minimal permission is $(0,0)$, and the maximal permission is $(1,0)$. Permissions are added by pairwise addition of their components, and ordered by the lexicographic ordering. A permission is splittable if its first component is 1. The split functions are defined by: $\mathsf{split1}((1,i)) = (1,i-1)$ and $\mathsf{split2}((1,i)) = (0,1)$.

We map Bornat's counting permissions onto our pairs model, by the following rules (in which n represents a positive integer):

$$0 \mapsto (1,0) \qquad n \mapsto (1,-n) \qquad -n \mapsto (0,n)$$

We study both intuitionistic and classical separation logic [15]. Both flavours contains pure (heap independent) and spatial (heap dependent) formulas:

n	$\in$	$\mathbb{N}$	
a,v	$\in$	$n \mid n+n \mid n-n \mid \ldots$	addresses and values
m			abstract permissions
Π	$::=$	$a=a \mid a \neq a \mid \mathtt{true} \mid \ldots$	pure formulas
Σ^I	$::=$	$a \overset{m}{\hookrightarrow} v$	intuitionistic spatial formulas
Σ^C	$::=$	$\mathtt{emp} \mid a \overset{m}{\mapsto} v$	classical spatial formulas

In intuitionistic separation logic, the atomic spatial formula is the *harpoon* $a \overset{m}{\hookrightarrow} v$ while in classical separation logic, the atomic spatial formula is the *points-to* predicate $a \overset{m}{\mapsto} v$. The harpoon $a \overset{m}{\hookrightarrow} v$ has a dual meaning. Firstly, $a \overset{m}{\hookrightarrow} v$ asserts that the heap contains a cell at address a with content v. Secondly, $a \overset{m}{\hookrightarrow} v$ asserts permission m to the cell at address a. If $m = m_W$, $a \overset{m}{\hookrightarrow} v$ asserts write and read authorization to the cell at address a, otherwise it asserts *readonly* authorization. The points-to predicate $a \overset{m}{\mapsto} v$ has the same meaning as the harpoon but it enforces that the heap contains *only* the cell at address a. Intuitionistic (respectively classical) separation logic is obtained by taking Σ to be Σ^I (respectively Σ^C) below:

$$A,B \quad ::= \quad \Pi \mid \Sigma \mid A \star A \mid A \wedge A \mid A \vee A \quad \text{formulas}$$

In $A \star B$, $\star$ is the *separating conjunction*. $A \star B$ represents a heap consisting of two *separate* subheaps A and B. $A \wedge B$ is the usual logical conjunction, which represents two different views of a heap. We do not include the standard implication $\Rightarrow$ because this simplifies the situation by avoiding bunched contexts in the proof system [17]. The fact that most separation-logic-based program verifiers do not include a standard implication [3, 16, 11] supports our choice. In addition, we use a language without variables, but describe how variables interfere with our algorithm in Sec. 7.

Because our algorithms are defined in Coq [10], we represent heaps as lists (which are well supported by Coq's standard library). A list entry is a triplet of an address, a valid permission, and a value:

c	$::=$	(a,m,v)	heap cells
h	$::=$	$[] \mid c :: h$	heaps

We define a projection operator to extract a value from a cell: $\mathsf{val}(a,m,v) = v$ and we write $h[a]$ to denote h's set of heap cells whose address are a:

$$\begin{aligned} [][a] &= \emptyset \\ ((a,m,v) :: h)[a'] &= \begin{cases} \{(a,m,v)\} \cup h[a] & \text{if } a = a' \\ h[a] & \text{otherwise} \end{cases} \end{aligned}$$

We write $h(a)$ to denote the sum of permissions to a occurring in h:

$$\begin{aligned} [](a) &= m_0 \\ ((a,m,v) :: h)(a') &= \begin{cases} m +_{\mathscr{M}} h(a) & \text{if } a = a' \\ h(a) & \text{otherwise} \end{cases} \end{aligned}$$

Conjunction of two heaps is simply list concatenation (written $h@h'$) and compatibility of heaps is standard:

$$h \# h' \triangleq (\forall a) \left(\begin{array}{l} h(a) +_{\mathscr{M}} h'(a) \leq m_W \text{ and} \\ (\forall c \in h[a], \forall c' \in h'[a])(\mathsf{val}(c) = \mathsf{val}(c')) \end{array} \right)$$

The semantics of formulas is standard. In particular, the semantics of pure formulas is left unaxiomatized:

h	$\models$	iff	Π	$\mathsf{oracle}(\Pi)$
h	$\models$	iff	$\mathtt{emp}$	$h = []$
h	$\models$	iff	$a \overset{m}{\hookrightarrow} v$	$(\exists h')(h = (a,m,v)@h')$
h	$\models$	iff	$a \overset{m}{\mapsto} v$	$h = (a,m,v)$
h	$\models$	iff	$A \star B$	$(\exists h_1, h_2)(h_1 \# h_2, h = h_1 @ h_2,$
				$h_1 \models$ iff A, and $h_2 \models B)$
h	$\models$	iff	$A \wedge B$	$h \models A$ and $h \models B$
h	$\models$	iff	$A \vee B$	$h \models A$ or $h \models B$

We write $A \vdash B$ to denote that A entails B. Appendix A shows $\vdash$'s definition, i.e., our proof system. It is driven by natural deduction rules that are common to the logic of bunched implications [17] and linear logic [20]. The only non-standard rules are (Splitting) and (Merging) [13] that lift permission splitting to formulas. The proof system is sound w.r.t. to the semantics above:

THEOREM 1 (SOUNDNESS OF THE PROOF SYSTEM). *If $A \vdash B$, then for all h, $h \models A$ implies $h \models B$.*

Because we use it later, we spell out Thm. 1's contraposition:

THEOREM 2 (CONTRAPOSITION OF THM. 1). *If there exists h such that $h \models A$ and $h \not\models B$, then $A \not\vdash B$.*

3. PRELIMINARIES

We abstract models of separation logic formulas (i.e., heaps) by their size. We use two abstractions of different precisions: the first abstraction is a *whole heap* abstraction while the second abstraction is a *per cell* abstraction. For the whole heap abstraction, the size of a heap is simply the sum of all permissions occurring in this heap:

$$\mathsf{size}_{\mathscr{M}}([]) = m_0 \qquad \mathsf{size}_{\mathscr{M}}((a,m,v) :: h) = m +_{\mathscr{M}} \mathsf{size}_{\mathscr{M}}(h)$$

For the per cell abstraction, the size of a heap is a *permission table* i.e., the corresponding heap without values:

pt_c	$::=$	(a,m)	permission table cells
pt	$::=$	$[] \mid pt_c :: pt$	permission tables

$$\mathsf{size}_{\mathscr{P}}([]) = [] \qquad \mathsf{size}_{\mathscr{P}}((a,m,v) :: h) = (a,m) :: \mathsf{size}_{\mathscr{P}}(h)$$

As for heaps, we write $pt(a)$ to denote the sum of permissions to a occurring in pt. Sizes are equipped with an order. For the whole heap abstraction, we use the order on permissions $\leq_{\mathscr{M}}$ while for the per cell abstraction, we use an order on permission tables $\leq_{\mathscr{P}}$:

$$pt \leq_{\mathscr{P}} pt' \triangleq (\forall a)(pt(a) \leq pt'(a))$$

Sizes are equipped with minimum and maximum functions. For the whole heap abstraction, we use standard definitions according to the corresponding order on permissions $\leq_{\mathscr{M}}$:

$$\mathsf{smin}_{\mathscr{M}}(m,m') = \begin{cases} m & \text{iff } m \leq_{\mathscr{M}} m' \\ m' & \text{otherwise} \end{cases}$$

$$\mathsf{smax}_{\mathscr{M}}(m,m') = \begin{cases} m' & \text{iff} \quad m \leq_{\mathscr{M}} m' \\ m & \text{otherwise} \end{cases}$$

For the per cell abstraction, we use definitions that merge permission tables point-wise:

$$\mathsf{smin}_{\mathscr{P}}(pt,pt') \triangleq \{pt'' \mid (\forall a)(pt''(a) = \mathsf{smin}_{\mathscr{M}}(pt(a),pt'(a)))\}$$
$$\mathsf{smax}_{\mathscr{P}}(pt,pt') \triangleq \{pt'' \mid (\forall a)(pt''(a) = \mathsf{smax}_{\mathscr{M}}(pt(a),pt'(a)))\}$$

>From now on, we subscript functions when we speak about a concrete abstraction while we use functions without subscripts when we describe properties of both abstractions.

4. THE DISPROVING ALGORITHM FOR INTUITIONISTIC SEPARATION LOGIC

To disprove entailment between formulas in intuitionistic separation logic, we search for models with bounded sizes. Formally, we use min and max functions that satisfy the following property:

THEOREM 3 (PROPERTY OF min AND max). *If $h \models A$, then there exist h_s, h_r such that $h = h_s @ h_r$ and $h_s \models A$ and $\mathsf{min}(A) \leq \mathsf{size}(h_s) \leq \mathsf{max}(A)$.*

Intuitively, Thm. 3 states that, given a satisfiable formula A (i.e., there exists h such that $h \models A$), there exists a "small" model h_s of A whose size is in between $\mathsf{min}(A)$ and $\mathsf{max}(A)$ (the possibility of a superfluous heap portion h_r reflects the semantics of intuitionistic separation logic). Then, we use min and max to disprove entailment as follows:

THEOREM 4 (min/max IS A DISPROVING ALGORITHM). *If A is satisfiable and $\mathsf{min}(B) \leq \mathsf{max}(A)$ does* not *hold, then $A \not\vdash B$.*

PROOF. Because A is satisfiable, there exists h such that $h \models A$. Then, by Thm. 3, it follows that there exists h_s such that $h_s \models A$ and $\mathsf{size}(h_s) \leq \mathsf{max}(A)$.

Now, suppose $h_s \models B$. Then, by Thm. 3, it follows that $\mathsf{min}(B) \leq \mathsf{size}(h_s)$. From $\mathsf{size}(h_s) \leq \mathsf{max}(A)$, by transitivity, it follows that $\mathsf{min}(B) \leq \mathsf{max}(A)$. This contradicts, however, the hypothesis that $\mathsf{min}(B) \leq \mathsf{max}(A)$ does not hold. Hence, $h_s \not\models B$.

From $h_s \models A$ and $h_s \not\models B$, by Thm. 2, it follows that $A \not\vdash B$. □

The reader might wonder why we formulate Thm. 4 with a negation on the $\leq$ operator instead of using a $<$ operator. The reason is that, for permission tables, $\neg(pt \leq_{\mathscr{P}} pt')$ is *not* equivalent to $pt' <_{\mathscr{P}} pt$ (where $<_{\mathscr{P}}$ would be defined in a way similar to $\leq_{\mathscr{P}}$).

In the remainder of this section, we show definitions of min and max for the whole heap abstraction and the per cell abstraction.

4.1 Whole Heap Abstraction

min and max are defined as follows:

$$\begin{aligned} \mathsf{min}(\Pi) &= m_0 \\ \mathsf{min}(a \overset{m}{\hookrightarrow} v) &= m \\ \mathsf{min}(A \star B) &= \mathsf{min}(A) +_{\mathscr{M}} \mathsf{min}(B) \\ \mathsf{min}(A \wedge B) &= \mathsf{smax}_{\mathscr{M}}(\mathsf{min}(A), \mathsf{min}(B)) \\ \mathsf{min}(A \vee B) &= \mathsf{smin}_{\mathscr{M}}(\mathsf{min}(A), \mathsf{min}(B)) \end{aligned}$$

$$\begin{aligned} \mathsf{max}(\Pi) &= m_0 \\ \mathsf{max}(a \overset{m}{\hookrightarrow} v) &= m \\ \mathsf{max}(A \star B) &= \mathsf{max}(A) +_{\mathscr{M}} \mathsf{max}(B) \\ \mathsf{max}(A \wedge B) &= \mathsf{max}(A) +_{\mathscr{M}} \mathsf{max}(B) \\ \mathsf{max}(A \vee B) &= \mathsf{smax}_{\mathscr{M}}(\mathsf{max}(A), \mathsf{max}(B)) \end{aligned}$$

4.2 Per Cell Abstraction

min and max are defined as follows:

$$\begin{aligned} \mathsf{min}(\Pi) &= [] \\ \mathsf{min}(a \overset{m}{\hookrightarrow} v) &= (a,m) :: [] \\ \mathsf{min}(A \star B) &= \mathsf{min}(A) @ \mathsf{min}(B) \\ \mathsf{min}(A \wedge B) &= \mathsf{smax}_{\mathscr{P}}(\mathsf{min}(A), \mathsf{min}(B)) \\ \mathsf{min}(A \vee B) &= \mathsf{smin}_{\mathscr{P}}(\mathsf{min}(A), \mathsf{min}(B)) \end{aligned}$$

$$\begin{aligned} \mathsf{max}(\Pi) &= [] \\ \mathsf{max}(a \overset{m}{\hookrightarrow} v) &= (a,m) :: [] \\ \mathsf{max}(A \star B) &= \mathsf{max}(A) @ \mathsf{max}(B) \\ \mathsf{max}(A \wedge B) &= \mathsf{smax}_{\mathscr{P}}(\mathsf{max}(A), \mathsf{max}(B)) \\ \mathsf{max}(A \vee B) &= \mathsf{smax}_{\mathscr{P}}(\mathsf{max}(A), \mathsf{max}(B)) \end{aligned}$$

Although max's definitions are very similar for the two abstractions, there is one case where they differ: in case $\wedge$ of max. In analogy with the per cell abstraction, one could expect $\mathsf{max}(A \wedge B)$ to be $\mathsf{smax}_{\mathscr{M}}(\mathsf{max}(A), \mathsf{max}(B))$ in the whole heap abstraction. To see why this is unsound, however, one can consider a formula where the right and left hand sides of a $\wedge$ represent separate parts of the heap such as $42 \overset{1}{\hookrightarrow} \_ \wedge 47 \overset{1}{\hookrightarrow} \_$ (where _ denotes irrelevant values).

5. THE DISPROVING ALGORITHM FOR CLASSICAL SEPARATION LOGIC

To disprove entailment between formulas in classical separation logic, we pursue a slightly different goal than for intuitionistic separation logic: instead of exhibiting *one* model whose size is bounded, we search for bounds on the size of *all* models.

Because we allow pure formulas, which do not constrain the size of heaps modelling them, we cannot always find an upper-bound on the size of heaps. Hence we add a distinguished "maximal" size that we write ∞ (both for the whole heap and for the per cell abstraction). The meaning of ∞ is axiomatized as follows:

$$(\forall m)(m \leq_{\mathscr{M}} \infty) \qquad (\forall pt)(pt \leq_{\mathscr{P}} \infty)$$

THEOREM 5 (PROPERTY OF min AND max). *If $h \models A$, then $\mathsf{min}(A) \leq \mathsf{size}(h) \leq \mathsf{max}(A)$.*

Then, we use min and max to disprove entailment as follows:

THEOREM 6 (min/max IS A DISPROVING ALGORITHM). *If A is satisfiable and $\mathsf{min}(B) \leq \mathsf{max}(A)$ does* not *hold, then $A \not\vdash B$.*

PROOF. Similar to the proof of Thm. 4. □

In the remainder of this section, we give definitions of min and max for the whole heap abstraction and the per cell abstraction. For min, we only show its definition in case of a formula `emp`; for other cases min's definition is similar to the intuitionistic case (see Sec. 4.1 and 4.2).

5.1 Whole Heap Abstraction

min and max are defined as follows:

$$\mathsf{min}(\mathtt{emp}) = m_0$$

$$\begin{aligned} \mathsf{max}(\Pi) &= \infty \\ \mathsf{max}(\mathtt{emp}) &= m_0 \\ \mathsf{max}(a \overset{m}{\mapsto} v) &= m \\ \mathsf{max}(A \star B) &= \mathsf{max}(A) +_{\mathscr{M}} \mathsf{max}(B) \\ \mathsf{max}(A \wedge B) &= \mathsf{smin}_{\mathscr{M}}(\mathsf{max}(A), \mathsf{max}(B)) \\ \mathsf{max}(A \vee B) &= \mathsf{smax}_{\mathscr{M}}(\mathsf{max}(A), \mathsf{max}(B)) \end{aligned}$$

As an example, consider the following computations of the maximum and minimum of two formulas (for the fractional permissions model):

$$\begin{array}{rl} & \mathsf{max}((42 \overset{1}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_) \vee \\ & \qquad ((42 \overset{1}{\mapsto} \_ \star 47 \overset{3/4}{\mapsto} \_) \wedge (47 \overset{1/2}{\mapsto} \_ \star 47 \overset{1/4}{\mapsto} \_ \star 42 \overset{1}{\mapsto} \_))) \\ = & \mathsf{max}_{\mathscr{M}}(1 +_{\mathscr{M}} 1/2, \mathsf{min}(\mathsf{max}(42 \overset{1}{\mapsto} \_ \star 47 \overset{3/4}{\mapsto} \_), \\ & \qquad \mathsf{max}(47 \overset{1/2}{\mapsto} \_ \star 47 \overset{1/4}{\mapsto} \_ \star 42 \overset{1}{\mapsto} \_))) \\ = & \mathsf{smax}_{\mathscr{M}}(\tfrac{3}{2}, \mathsf{smin}_{\mathscr{M}}(\tfrac{7}{4}, \tfrac{7}{4})) = \tfrac{7}{4} \end{array}$$

$$\mathsf{min}(42 \overset{1}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_) = 1 +_{\mathscr{M}} \tfrac{1}{2} +_{\mathscr{M}} \tfrac{1}{2} = 2$$

Now, because $2 \leq_{\mathscr{M}} 7/4$ does not hold, our algorithm deduces:

$$(42 \overset{1}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_) \vee ((42 \overset{1}{\mapsto} \_ \star 47 \overset{3/4}{\mapsto} \_) \wedge (47 \overset{1/2}{\mapsto} \_ \star 47 \overset{1/4}{\mapsto} \_ \star 42 \overset{1}{\mapsto} \_)) \not\vdash 42 \overset{1}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_$$

We now compare max's definition for $\star$ and $\wedge$ in both semantics (similar remarks will apply for the per cell abstraction).

- In the classical semantics, we have $\mathsf{max}(A \star B) = \mathsf{max}(A) +_{\mathscr{M}} \mathsf{max}(B)$, while we have $\mathsf{max}(A \wedge B) = \mathsf{smin}_{\mathscr{M}}(\mathsf{max}(A), \mathsf{max}(B))$. This reflects the intuition that, *(1)* in $A \star B$, A and B represents *separate* heaps while *(2)* in $A \wedge B$, A and B represents different *views* of the *same* heap.
- In the intuitionistic semantics, however, we have $\mathsf{max}(A \star B) = \mathsf{max}(A \wedge B) = \mathsf{max}(A) +_{\mathscr{M}} \mathsf{max}(B)$. This reflects that, both in $A \star B$ *and* in $A \wedge B$, A and B must (for $\star$) or may (for $\wedge$) represent separate parts of the heap.

5.2 Per Cell Abstraction

min and max are defined as follows:

$$\mathsf{min}(\mathtt{emp}) = []$$

$$\begin{array}{rcl} \mathsf{max}(\Pi) & = & \infty \\ \mathsf{max}(\mathtt{emp}) & = & [] \\ \mathsf{max}(a \overset{m}{\mapsto} v) & = & (a, m) :: [] \\ \mathsf{max}(A \star B) & = & \begin{cases} \mathsf{max}(A) @ \mathsf{max}(B) & \text{if } \mathsf{max}(A) \neq \infty \text{ and } \mathsf{max}(B) \neq \infty \\ \infty & \text{otherwise} \end{cases} \\ \mathsf{max}(A \wedge B) & = & \mathsf{smin}_{\mathscr{P}}(\mathsf{max}(A), \mathsf{max}(B)) \\ \mathsf{max}(A \vee B) & = & \mathsf{smax}_{\mathscr{P}}(\mathsf{max}(A), \mathsf{max}(B)) \end{array}$$

In this case, the definitions of max are similar in both abstractions.

5.3 On ∞

We elaborate here on the consequence of our use of ∞ above for the whole heap abstraction (similar comments apply to the per cell abstraction).

The $\star$ and $\vee$ connectives have a different behavior w.r.t. ∞ than $\wedge$. To see why, consider the following properties of $+_{\mathscr{M}}$, $\mathsf{smin}_{\mathscr{M}}$, and $\mathsf{smax}_{\mathscr{M}}$ w.r.t. ∞:

$$(\forall m)(m +_{\mathscr{M}} \infty = \infty +_{\mathscr{M}} m = \infty)$$
$$(\forall m)(\mathsf{smin}_{\mathscr{M}}(m, \infty) = \mathsf{smin}_{\mathscr{M}}(\infty, m) = m)$$
$$(\forall m)(\mathsf{smax}_{\mathscr{M}}(m, \infty) = \mathsf{smax}_{\mathscr{M}}(\infty, m) = \infty)$$

Because of the properties above, both the $\star$ and $\vee$ operators "spread" ∞: for $op \in \{\star, \vee\}$, $\mathsf{max}(A\ op\ B) = \infty$ if $\mathsf{max}(A) = \infty$ or $\mathsf{max}(B) = \infty$. The $\wedge$ operator, however, stops ∞ from spreading: $\mathsf{max}(A \wedge B) \neq \infty$ if $\mathsf{max}(A) \neq \infty$ or $\mathsf{max}(B) \neq \infty$.

Because the variants of separation logic used in program verifiers [3, 11] impose pure formulas and spatial formulas to be $\wedge$-conjoined at the top level, max would never return ∞ in these variants.

6. PRECISION

Ideally, min and max's definitions need to be as tight as possible, while retaining soundness (Thms. 3 and 5). We exemplify this for both flavours of the logic. Our explanations focus solely on the whole heap abstraction but our remarks apply to both abstractions.

- In the intuitionistic semantics, we have $\mathsf{max}(A \wedge B) = \mathsf{max}(A) + \mathsf{max}(B)$. At first sight, this might appear too loose, but the semantics of the harpoon requires this definition for soundness. To see why, one can consider a formula in which the right and left hand sides of a $\wedge$ represent separate parts of the heap, such as $42 \overset{1}{\hookrightarrow} \_ \wedge 47 \overset{1}{\hookrightarrow} \_$.
- In the classical semantics, we have $\mathsf{max}(A \wedge B) = \mathsf{smin}_{\mathscr{M}}(\mathsf{max}(A), \mathsf{max}(B))$. This definition is tight: if $\mathsf{max}(A) < \mathsf{max}(B)$, by $\wedge$'s semantics, it follows that any model of B whose size is greater than $\mathsf{max}(A)$ cannot be a model of $A \wedge B$ (and conversely). In the classical semantics, we have $\mathsf{max}(A \vee B) = \mathsf{smax}_{\mathscr{M}}(\mathsf{max}(A), \mathsf{max}(B))$. This definition is also tight because, if $\mathsf{max}(A) < \mathsf{max}(B)$, by $\vee$'s semantics, it follows that a model of B can be a model of $A \vee B$ (and conversely).

7. EXTENSION TO OTHER OPERATORS OF SEPARATION LOGIC

In this section, we review extensions of our algorithm for which our results are partial. Our algorithm has been extended to deal with the magic wand solely for the classical semantics (Sec. 7.1), soundness of min and max for quantifiers has been checked only on pen and paper (Sec. 7.2), and variables have been integrated only with the whole heap abstraction (Sec. 7.3).

7.1 The magic wand $\mathrel{-\!\!\star}$

The magic wand $\mathrel{-\!\!\star}$ matches the resource conjunction $\star$, in the sense that the modus ponens law is satisfied: $A \star (A \mathrel{-\!\!\star} B)$ implies B. The magic wand's semantics quantifies over models of the wand's left-hand side:

$$h \models A \mathrel{-\!\!\star} B \quad \text{iff} \quad (\forall h')((h' \models A \text{ and } h' \# h) \text{ implies } h' @ h \models B)$$

In the classical semantics, we have mechanically verified that our algorithm can be extended to deal with the magic wand. Because min and max are now mutually recursive, min, like max, can return ∞. For the whole heap abstraction, definitions of min and max from Sec. 5.1 are completed as follows (see Appendix B for the per cell approximation):

$$\mathsf{min}(A \mathrel{-\!\!\star} B) = \begin{cases} \mathsf{min}(B) -_{\mathscr{M}} \mathsf{max}(A) & \text{if } \mathsf{min}(B) \neq \infty \text{ and } \mathsf{max}(A) \neq \infty \\ m_0 & \text{otherwise} \end{cases}$$

$$\mathsf{max}(A \mathrel{-\!\!\star} B) = \begin{cases} \mathsf{max}(B) -_{\mathscr{M}} \mathsf{min}(A) & \text{if } \mathsf{max}(B) \neq \infty \text{ and } \mathsf{min}(A) \neq \infty \\ \infty & \text{otherwise} \end{cases}$$

For these definitions to be sound, we need to assume that formulas appearing on the left-hand side of the magic wand are satisfiable. While this is harmful from a theoretical point of view, the literature so far on program verification that uses the magic wand in examples [13, 14] suggests this is a plausible assumption for program verifiers.

We believe this definition could be adapted for the intuitionistic semantics, but it is a non-trivial task. In a nutshell, the universal quantification in the magic wand's semantics and the way Thm. 3 is formulated (i.e., with an existential quantifier) prevent a simple proof by induction, and we leave this for future work.

7.2 Quantification

We distinguish between quantification on values and permissions. We add the following formulas to Sec. 2's language:

$$A,B \quad ::= \quad \ldots \mid \exists v.A \mid \exists m.A \mid \forall v.A \mid \forall m.A$$

The semantics is standard and we omit the corresponding proof rules which are also standard. To define min and max, we introduce a "very small permission" $\varepsilon \in \mathcal{M}$. The meaning of ε is axiomatized by: $(\forall m)(m <_{\mathcal{M}} m +_{\mathcal{M}} \varepsilon)$

For both abstractions min and max are defined as follows:

$$\begin{array}{ll} \mathsf{min}(\exists v.A) = \mathsf{min}(A) & \mathsf{min}(\exists m.A) = \mathsf{min}(A[\varepsilon/m]) \\ \mathsf{max}(\exists v.A) = \mathsf{max}(A) & \mathsf{max}(\exists m.A) = \mathsf{max}(A[m_W/m]) \end{array}$$

$$\begin{array}{ll} \mathsf{min}(\forall v.A) = \mathsf{min}(A) & \mathsf{min}(\forall m.A) = \mathsf{min}(A[m_W/m]) \\ \mathsf{max}(\forall v.A) = \mathsf{max}(A) & \mathsf{max}(\forall m.A) = \mathsf{max}(A[m_W/m]) \end{array}$$

7.3 Variables

In real-world flavours of separation logic, variables are used. For the whole heap abstraction, variables (including quantification over variables) do not create any extra difficulties for our work. For the per cell abstraction, it is unclear at this stage how variables can be handled elegantly, and we leave this for future work.

8. COMPLEXITY

The complexity of our techniques is linear for the whole heap abstraction, and $O(n(\log n))$ for the per cell abstraction (where n is the input formula's size). To our knowledge, there is no complexity result for entailment checking in separation logic. We believe the complexity of our disproving algorithm is low enough to be useful when entailment must be checked quickly.

This is supported by the fact that integrating both the magic wand and quantifiers do not increase our algorithm's complexity, while they typically increase by orders of magnitude the complexity of model-checking or validity-checking [8, 7].

9. SOUNDNESS

All theorems from Sec. 2, 4, and 5 have been mechanically checked with Coq [10]. Addition of the magic wand (Sec. 7.1) for the classical semantics has also been mechanically checked. Proofs about the proof system (Thms. 1 and 2); properties of permissions, heaps, permissions tables etc. are 3090 lines long. The proofs of Thms 3, 4, 5 and 6 are 620 lines long. The proofs have been engineered so that certified implementations of the two abstractions could be extracted. Proof scripts are available online [1].

Proofs for quantifiers (Sec. 7.2) have been checked only on pen and paper.

10. RELATED WORK AND CONCLUSION

Related Work.

In the context of program verifiers, checking entailment between separation logic formulas has been studied for while languages [4] and object-oriented languages [11, 9]. In these fragments, the magic wand $-\!\!\star$ is omitted and special predicates for describing data structures are present.

Properties (decidability, undecidability, and complexity) both for model checking and for validity have been studied [8, 2, 7]. To our knowledge, disproving entailment has only been studied in a fully-fledged tableaux procedure [12]. The cited work's algorithm for checking entailment can generate either proofs or counter-models, whereas our algorithm focuses on disproving entailment and does not generate counter-models.

Conclusion.

We have presented new techniques for disproving entailment between separation logic formulas, which we believe to be particularly relevant in the context of automated provers for separation logic. Because our algorithm's complexity is low, it is of interest wherever entailment checking needs to be checked quickly. We have provided two different techniques (of different precision), each of which can be applied to both the intuitionistic and classical flavours of separation logic. We provide mechanical proofs of the soundness of our techniques [1].

Acknowledgments.

We thank Didier Galmiche and Marieke Huisman for their fruitful comments about this paper. Hurlin was supported in part by IST-FET-2005-015905 Mobius project and ANR-06-SETIN-010 ParSec project.

11. REFERENCES

[1] Coq proof scripts: http://tinyurl.com/aez9sj.

[2] J. Berdine, C. Calcagno, P. W. O'Hearn. A decidable fragment of separation logic. In K. Lodaya, M. Mahajan, eds., *Foundations of Software Technology and Theoretical Computer Science*, vol. 3821 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[3] J. Berdine, C. Calcagno, P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, W.-P. de Roever, eds., *Formal Methods for Components and Objects*, vol. 4111 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[4] J. Berdine, C. Calcagno, P. W. O'Hearn. Symbolic execution with separation logic. In K. Yi, ed., *Asian Programming Languages and Systems Symposium*, vol. 3780 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[5] R. Bornat, P. W. O'Hearn, C. Calcagno, M. Parkinson. Permission accounting in separation logic. In J. Palsberg, M. Abadi, eds., *Principles of Programming Languages*. ACM Press, 2005.

[6] J. Boyland. Checking interference with fractional permissions. In R. Cousot, ed., *Static Analysis Symposium*, vol. 2694 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[7] R. Brochenin, S. Demri, E. Lozes. On the almighty wand. In M. Kaminski, S. Martini, eds., *Computer Science Logic*. Springer-Verlag, 2008.

[8] C. Calcagno, H. Yang, P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data

$$\frac{}{A \vdash A}\ (\text{Id}) \qquad \frac{A \vdash B}{A, C \vdash B}\ (\text{Weak}) \qquad \frac{\mathsf{oracle}(\Pi)}{\mathtt{true} \vdash \Pi}\ (\text{Oracle})$$

$$\frac{A_1 \vdash B_1 \qquad A_2 \vdash B_2}{A_1, A_2 \vdash B_1 \star B_2}\ (\star \text{ Intro}) \qquad \frac{A \vdash B_1 \star B_2 \qquad C, B_1, B_2 \vdash D}{A, C \vdash D}\ (\star \text{ Elim})$$

$$\frac{A, B_1 \vdash B_2}{A \vdash B_1 \mathrel{-\!\!\star} B_2}\ (\mathrel{-\!\!\star} \text{ Intro}) \qquad \frac{A \vdash C_1 \mathrel{-\!\!\star} C_2 \qquad B \vdash C_1}{A, B \vdash C_2}\ (\mathrel{-\!\!\star} \text{ Elim})$$

$$\frac{A \vdash B_1 \qquad A \vdash B_2}{A \vdash B_1 \wedge B_2}\ (\wedge \text{ Intro}) \qquad \frac{A \vdash B_1 \wedge B_2}{A \vdash B_1}\ (\wedge \text{ Elim 1}) \qquad \frac{A \vdash B_1 \wedge B_2}{A \vdash B_2}\ (\wedge \text{ Elim 2})$$

$$\frac{A \vdash B_1}{A \vdash B_1 \vee B_2}\ (\vee \text{ Intro 1}) \qquad \frac{A \vdash B_2}{A \vdash B_1 \vee B_2}\ (\vee \text{ Intro 2})$$

$$\frac{\vee \text{ does not occur in } A}{A \vdash \mathsf{split1}(A) \star \mathsf{split2}(A)}\ (\text{Splitting}) \qquad \frac{\vee \text{ does not occur in } A}{\mathsf{split1}(A) \star \mathsf{split2}(A) \vdash A}\ (\text{Merging})$$

Figure 1: Proof Sytem

structures. In R. Hariharan, M., V. Vinay, eds., *Foundations of Software Technology and Theoretical Computer Science*, vol. 2245 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[9] W. Chin, C. David, H. Nguyen, S. Qin. Enhancing modular OO verification with separation logic. In G. C. Necula, P. Wadler, eds., *Principles of Programming Languages*. ACM Press, 2008.

[10] Coq development team. The Coq proof assistant reference manual V8.0. Technical Report 255, INRIA, France, 2004. http://coq.inria.fr/doc/main.html.

[11] D. DiStefano, M. Parkinson. jStar: Towards practical verification for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, vol. 43. ACM Press, 2008.

[12] D. Galmiche, D. Mery. Tableaux and resource graphs for separation logic. *Journal of Logic and Computation*, 2008.

[13] C. Haack, C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In J. Meseguer, G. Rosu, eds., *Algebraic Methodology and Software Technology*, vol. 5140 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[14] C. Haack, C. Hurlin. Resource usage protocols for iterators. *Journal of Object Technology*, 2009. To appear.

[15] S. Ishtiaq, P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, 2001.

[16] B. Jacobs, F. Piessens. The verifast program verifier. Technical Report CW520, Katholieke Universiteit Leuven, 2008.

[17] P. W. O'Hearn, D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2), 1999.

[18] M. Parkinson. Local reasoning for Java. Technical Report UCAM-CL-TR-654, University of Cambridge, 2005.

[19] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*. IEEE Press, 2002.

[20] P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, 1993.

APPENDIX

A. PROOF SYSTEM

Fig. 1 shows our proof system (where weakening is only included for the intuitionistic flavour).

In the proof rules (Splitting) and (Merging), we use split1 and split2 (defined below) to lift permission splitting to formula splitting. The conditions on the proof rules for splitting and merging, ensure that no disjunctions occur within the formulas concerned. This is because splitting and merging is unsound for disjunctions (and formulas with the magic wand). As we only split formulas without these connectives, we need not define those cases for the definition of splitting formulas:

$$\begin{array}{lcl}
\mathsf{split1}(\Pi) & = & \Pi \\
\mathsf{split1}(\mathtt{emp}) & = & \mathtt{emp} \\
\mathsf{split1}(a \overset{m}{\hookrightarrow} v) & = & a \overset{\mathsf{split1}(m)}{\hookrightarrow} v \\
\mathsf{split1}(a \overset{m}{\mapsto} v) & = & a \overset{\mathsf{split1}(m)}{\mapsto} v \\
\mathsf{split1}(A \star B) & = & \mathsf{split1}(A) \star \mathsf{split1}(B) \\
\mathsf{split1}(A \wedge B) & = & \mathsf{split1}(A) \wedge \mathsf{split1}(B)
\end{array}$$

$$\begin{array}{lcl}
\mathsf{split2}(\Pi) & = & \Pi \\
\mathsf{split2}(\mathtt{emp}) & = & \mathtt{emp} \\
\mathsf{split2}(a \overset{m}{\hookrightarrow} v) & = & a \overset{\mathsf{split2}(m)}{\hookrightarrow} v \\
\mathsf{split2}(a \overset{m}{\mapsto} v) & = & a \overset{\mathsf{split2}(m)}{\mapsto} v \\
\mathsf{split2}(A \star B) & = & \mathsf{split2}(A) \star \mathsf{split2}(B) \\
\mathsf{split2}(A \wedge B) & = & \mathsf{split2}(A) \wedge \mathsf{split2}(B)
\end{array}$$

B. OMITTED CASES

The definitions of min and max from Sec. 5.2 are completed as follows (where $-_{\mathscr{P}}$ is defined by applying the unary $-_{\mathscr{M}}$ operator point-wisely to permission tables):

$$\mathsf{min}(A \mathrel{-\!\!\star} B) = \begin{cases} \mathsf{min}(B) @ -_{\mathscr{P}}(\mathsf{max}(A)) & \text{if } \mathsf{min}(B) \neq \infty \text{ and } \mathsf{max}(A) \neq \infty \\ [] & \text{otherwise} \end{cases}$$

$$\mathsf{max}(A \mathrel{-\!\!\star} B) = \begin{cases} \mathsf{max}(B) @ -_{\mathscr{P}}(\mathsf{min}(A)) & \text{if } \mathsf{max}(B) \neq \infty \text{ and } \mathsf{min}(A) \neq \infty \\ \infty & \text{otherwise} \end{cases}$$