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

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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IWACO ’09, July 6 2009, Genova, Italy
Copyright 2009 ACM 978-1-60558-546-8/09/07 ...$10.00.

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 \( \mathcal{M} \) consists of:

1. A commutative semigroup with total binary operator \(+_{\mathcal{M}}\).
2. A minimal permission, \( m_{0} \), that satisfies the following property: \((\forall m)(m_{0} +_{\mathcal{M}} m = m)\).
3. A maximal permission, \( m_{W} \).
4. A total order on permissions, \( \leq_{\mathcal{M}} \).
5. A subset of permissions which are defined to be splittable.
6. A pair of functions on permissions, \( \text{split}1 \) and \( \text{split}2 \), which define the two parts into which a splittable permission may be split. These functions must satisfy the following property: \((\forall m)(\text{split}1(m) +_{\mathcal{M}} \text{split}2(m) = m)\).

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 satisfies the following inequality: \( m_{0} \leq_{\mathcal{M}} m \leq_{\mathcal{M}} m_{W} \). Our model, however, includes permissions that are bigger than \( m_{W} \), because we need to sum permissions above \( m_{W} \). That is why \(+_{\mathcal{M}}\) is a total order.

Our paper’s results have been established for two concrete permission 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: \(\text{split}1(m) = \text{split}2(m) = \frac{m}{2}\).

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...
A shows a split1. The split functions are defined by:

\[ (i_1, i_2) \mapsto 1 \quad \text{if} \quad i_1 < i_2 \]

\[ (i_1, i_2) \mapsto -1 \quad \text{if} \quad i_1 > i_2 \]

\[ (i_1, i_2) \mapsto 0 \quad \text{otherwise} \]

We map Bornat’s counting permissions onto our pairs model, by the following rules (where \( n \) represents a positive integer):

\[ 0 \mapsto (1, 0) \quad n \mapsto (1, -n) \quad -n \mapsto (0, n) \]

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

\[
\begin{align*}
  n & \in \mathbb{N} \\
  a, v & \in \{ n \mid n + n, n - n \} \quad \text{addresses and values} \\
  m & \in \text{abstract permissions} \\
  \Pi & := a = a \mid a \neq a \mid \text{true} \quad \ldots \quad \text{pure formulas} \\
  \Sigma^I & := a \mapsto v \quad \text{intuitionistic spatial formulas} \\
  \Sigma^C & := \text{emp} \mid a \mapsto v \quad \text{classical spatial formulas}
\end{align*}
\]

In intuitionistic separation logic, the atomic spatial formula is the harpoon \( a \mapsto v \) while in classical separation logic, the atomic spatial formula is the points-to predicate \( a \mapsto v \). The harpoon \( a \mapsto v \) has a dual meaning. Firstly, \( a \mapsto v \) asserts that the heap contains a cell at address \( a \) with content \( v \). Secondly, \( a \mapsto v \) asserts permission \( m \) to the cell at address \( a \). If \( m = m_w \), \( a \mapsto v \) asserts write and read authorization to the cell at address \( a \), otherwise it asserts read-only authorization. The points-to predicate \( a \mapsto v \) 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 \( \Sigma \) to be \( \Sigma^I \) (respectively \( \Sigma^C \)) below:

\[
A, B := \Pi \mid \Sigma \mid A \oplus A \mid A \land A \mid A \lor A \quad \text{formulas}
\]

In \( A \oplus B \), \( \oplus \) is the separating conjunction. \( A \land B \) represents a heap consisting of two separate subheaps \( A \) and \( B \). \( A \lor B \) 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 Sect. 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:

\[
c := (a, m, v) \quad \text{heap cells} \\
h := \emptyset \quad \text{h \_ heaps}
\]

We define a projection operator to extract a value from a cell:

\[
\text{val}(a, m, v) = v \\
\text{h}[a] = \{ (a, m, v) \} \cup h[a] \quad \text{if} \quad a = a' \\
\text{h}[a] \quad \text{otherwise}
\]

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

\[
|a| = m_0 \\
(a, m, v) \mapsto h(a)' = \begin{cases} m + \h[A] & \text{if} \quad a = a' \\
\ h[\ a] & \text{otherwise}
\end{cases}
\]

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

\[
h \| h' \iff (\forall a)(h(a) + \h[A] h'(a) \leq m_w) \\
(\forall c \in h[a], \forall c' \in h'[a])(\text{val}(c) = \text{val}(c'))
\]

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

\[
h \models \text{iff} \Pi \quad \text{oracle}(\Pi) \\
h \models \text{iff} \quad \text{emp} \quad h = \emptyset \\
h \models \text{iff} \quad a \mapsto v \quad (\exists h')(h = (a, m, v) @ h') \\
h \models \text{iff} \quad a \mapsto v \quad h = (a, m, v) \\
h \models \text{iff} \quad A \ast B \quad (\exists h_1, h_2)(h_1 \# h_2, h = h_1 \# h_2, \\
\ h_1 \models \text{iff} A, \ h_2 \models B) \\
h \models \text{iff} \quad A \land B \quad h = A \land h = B \\
h \models \text{iff} \quad A \lor B \quad h = A \lor h = B
\]

We write \( A \ast B \) 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 \( h \nvdash B \), then \( A \nvdash B \).

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:

\[
\text{size}_{\ast}([]) = m_0 \\
\text{size}_{\ast}((a, m, v) : h) = m + \h[A] \text{size}_{\ast}(h)
\]

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

\[
\text{pt}_{\ast} := (a, m) \quad \text{permission table cells} \\
\text{pt} := \emptyset \quad \text{pt}_{\ast} : \text{pt} \quad \text{permission tables}
\]

\[
\text{size}_{\ast}([]) = 0 \\
\text{size}_{\ast}((a, m, v) : h) = (a, m) \ast \text{size}_{\ast}(h)
\]

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 \( \leq_{\ast} \) while for the per cell abstraction, we use an order on permission tables \( \leq_{\ast} \):

\[
pt \leq_{\ast} pt' \iff (\forall a)(pt(a) \leq pt'(a))
\]

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

\[
\text{min}_{\ast}(m, m') = \begin{cases} m & \text{if} \quad m \leq_{\ast} m' \\
\ m' & \text{otherwise}
\end{cases}
\]
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 \( h_s, h_t \) such that \( h = h_s @ h_t \) and \( h \models A \) and \( \min(A) \leq \text{size}(h) \leq \max(A) \).

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 \( h_t \) of \( A \) whose size is in between \( \min(A) \) and \( \max(A) \) (the possibility of a superfluous heap portion \( h_t \) 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 \( \max(B) \leq \min(B) \) does not hold, then \( A \not\models B \).

**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 \( \text{size}(h_s) \leq \max(A) \).

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

From \( h_s \models A \) and \( h \models B \), by Thm. 2, it follows that \( A \not\models B \). ☐

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, \( \neg (pt \leq \rho pt') \) is not equivalent to \( pt' < \rho pt \) (where \( < \rho \) would be defined in a way similar to \( \leq \rho \)). 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

\( \min(A) \) and \( \max(A) \) are defined as follows:

\[
\begin{align*}
\min(A) &= m_0 \\
\max(A) &= m
\end{align*}
\]

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

\[
\begin{align*}
\min_{\rho}(pt, pt') &= \min\{ \rho pt', (\forall a)(\rho pt'(a) = \min_{\rho}(pt(a), pt'(a))) \} \\
\max_{\rho}(pt, pt') &= \max\{ \rho pt', (\forall a)(\rho pt'(a) = \max_{\rho}(pt(a), pt'(a))) \}
\end{align*}
\]

>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.

4.2 Per Cell Abstraction

\( \min \) and \( \max \) are defined as follows:

\[
\begin{align*}
\min(A) &= m_0 \\
\max(A) &= m
\end{align*}
\]

Although \( \max \)'s definitions are very similar for the two abstractions, there is one case where they differ: in case \( A \land \top \). In analogy with the per cell abstraction, one could expect \( \max(A \land B) \) to be \( \max_{\rho}(\max(A), \max(B)) \) 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 \( A \land \top \) represent separate parts of the heap such as \( 42_{\top} \land \top \land 47_{\top} \land \top \) (where \( \top \) 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 modeling them, we cannot always find an upper-bound on the size of heaps. Hence we add a distinguished "maximal" size that we write \( \infty \) (both for the whole heap and for the per cell abstraction). The meaning of \( \infty \) is axiomatized as follows:

\[
(\forall m)(m \leq m) \quad (\forall pt)(pt \leq m)
\]

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

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

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

**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 \( \text{emp} \); 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:

\[
\begin{align*}
\min(\text{emp}) &= m_0 \\
\max(\text{emp}) &= m_0 \\
\max(A) &= m_0 \\
\max(A \land B) &= \max(A) \land \max(B)
\end{align*}
\]
As an example, consider the following computations of the maximum and minimum of two formulas (for the fractional permissions model):

\[
\max((42 \frac{1}{2} \times 47 \frac{1}{4}) \lor (42 \frac{1}{2} \times 47 \frac{1}{4} \lor 42 \frac{1}{2} \times 47 \frac{1}{4})) = 0 + \max(42 \frac{1}{2} \times 47 \frac{1}{4}, \\
\max(42 \frac{1}{2} \times 47 \frac{1}{4} \lor 42 \frac{1}{2} \times 47 \frac{1}{4})) = \max(42 \frac{1}{2} \times 47 \frac{1}{4}, \min(42 \frac{1}{2} \times 47 \frac{1}{4}, 42 \frac{1}{2} \times 47 \frac{1}{4})) = 42 \frac{1}{2} \times 47 \frac{1}{4}
\]

Now, because \(2 \leq \frac{7}{4}\), our algorithm deduces:

\[
(42 \frac{1}{2} \times 47 \frac{1}{4} \lor (42 \frac{1}{2} \times 47 \frac{1}{4} \lor 42 \frac{1}{2} \times 47 \frac{1}{4})
\]

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

- In the classical semantics, we have \(\max(A \star B) = \max(A) \lor \max(B)\), while we have \(\max(A \land B) = \min(A, B)\). This reflects the intuition that, (1) in \(A \star B\) and \(B\) represents separate heaps while (2) in \(A \land B\), and \(B\) represents different views of the same heap.

- In the intuitionistic semantics, however, we have \(\max(A \star B) = \max(A \land B) = \max(A) \lor \max(B)\). This reflects that, both in \(A \star B\) and \(A \land B\), and \(B\) must (for \(\star\)) or may (for \(\land\)) represent separate parts of the heap.

5.2 Per Cell Abstraction

\textbf{min} and \textbf{max} are defined as follows:

\[
\min(\text{emp}) = \emptyset,
\max(\text{emp}) = \emptyset,
\max(A \lor \neg m \lor \neg \neg m) = \{ (a, m) : \emptyset \},
\max(A \star B) = \begin{cases} 
\max(A) \lor \max(B) & \text{if } \max(A) \neq \infty \text{ and } \max(B) \neq \infty \\
\infty & \text{otherwise}
\end{cases}
\max(A \land B) = \min(A, B),
\max(A \lor B) = \max(A, B)
\]

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

5.3 On \(\infty\)

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

- The \(\star\) and \(\lor\) connectives have a different behavior w.r.t. \(\infty\) than \(\land\). To see why, consider the following properties of \(+, \lor, \neg\), and \(\max\), w.r.t. \(\infty\): for \(\forall m \in \{*, \lor\}, \max(A \circ B) = \infty\) if \(\max(A) = \infty\) or \(\max(B) = \infty\). The \(\land\) operator, however, stops \(\infty\) from spreading: \(\max(A \land B) = \max(A) \land \max(B)\).

Because of the properties above, both the \(\star\) and \(\lor\) operators “spread” \(\infty\): for \(op \in \{*, \lor\}, \max(op A) = \infty\) if \(\max(A) = \infty\) or \(\max(B) = \infty\). The \(\land\) operator, however, stops \(\infty\) from spreading: \(\max(A \land B) = \max(A) \land \max(B)\).

Because of the above, our algorithm can be extended to deal with the magic wand. 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 \(\infty\). 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):

\[
\min(A \rightarrow B) = \begin{cases} 
\min(B) - \neg \max(A) & \text{if } \min(B) \neq \infty \text{ and } \max(A) \neq \infty \\
m_0 & \text{otherwise}
\end{cases}
\max(A \rightarrow B) = \begin{cases} 
\max(B) - \neg \min(A) & \text{if } \max(B) \neq \infty \text{ and } \min(A) \neq \infty \\
\infty & \text{otherwise}
\end{cases}
\]
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:

\[ A, B ::= \ldots | \exists v. A | \exists m. A | \forall v. A | \forall m. A \]

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 \( \varepsilon \) is axiomatized by:

\[(\forall m)(m <_\varepsilon m +_\varepsilon \varepsilon)\]

For both abstractions min and max are defined as follows:

\[
\begin{align*}
\min(\exists v. A) &= \min(A) & \min(\exists m. A) &= \min(A|\varepsilon/m) \\
\max(\exists v. A) &= \max(A) & \max(\exists m. A) &= \max(A|mN/m) \\
\min(\forall v. A) &= \min(A) & \min(\forall m. A) &= \min(A|nW/m) \\
\max(\forall v. A) &= \max(A) & \max(\forall m. A) &= \max(A|mW/m)
\end{align*}
\]

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 \( \rightsquigarrow \) 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 tableau 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


Figure 1: Proof System

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.

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:

\[
\begin{align*}
\text{split1}(\Pi) &= \Pi \\
\text{split1}(\text{emp}) &= \text{emp} \\
\text{split1}(a \mapsto v) &= a \mapsto v \\
\text{split1}(m \mapsto v) &= m \mapsto v \\
\text{split1}(a \cdot B) &= \text{split1}(A) \cdot \text{split1}(B) \\
\text{split1}(A \land B) &= \text{split1}(A) \land \text{split1}(B) \\
\text{split2}(\Pi) &= \Pi \\
\text{split2}(\text{emp}) &= \text{emp} \\
\text{split2}(a \mapsto v) &= a \mapsto v \\
\text{split2}(m \mapsto v) &= m \mapsto v \\
\text{split2}(a \cdot B) &= \text{split2}(A) \cdot \text{split2}(B) \\
\text{split2}(A \land B) &= \text{split2}(A) \land \text{split2}(B)
\end{align*}
\]

B. OMITTED CASES

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

\[
\begin{align*}
\text{min}(A \rightarrow B) &= \begin{cases} 
\text{min}(B @ -\varphi (\text{max}(A))) & \text{if min}(B) \neq \infty \text{ and } \max(A) \neq \infty \\
\mathbb{1} & \text{otherwise} \\
\end{cases} \\
\text{max}(A \rightarrow B) &= \begin{cases} 
\text{max}(B @ -\varphi (\text{min}(A))) & \text{if max}(B) \neq \infty \text{ and } \min(A) \neq \infty \\
\infty & \text{otherwise} \\
\end{cases}
\end{align*}
\]