Disproving Inductive Entailments in Separation Logic via Base Pair Approximation

James Brotherston[1] and Nikos Gorogiannis[2]

[1] Dept. of Computer Science, University College London
[2] Dept. of Computer Science, Middlesex University London

Abstract. We give a procedure for establishing the *invalidity* of logical entailments in the symbolic heap fragment of separation logic with user-defined inductive predicates, as used in program verification. This disproof procedure attempts to infer the existence of a countermodel to an entailment by comparing computable model summaries, a.k.a. *bases* (modified from earlier work), of its antecedent and consequent. Our method is sound and terminating, but necessarily incomplete. Experiments with the implementation of our disproof procedure indicate that it can correctly identify a substantial proportion of the invalid entailments that arise in practice, at reasonably low time cost. Accordingly, it can be used, e.g., to improve the output of theorem provers by returning "no" answers in addition to "yes" and "unknown" answers to entailment questions, and to speed up proof search or automated theory exploration by filtering out invalid entailments.

1 Introduction

Separation logic [23] is a well known and relatively popular formalism for Hoare-style verification of heap-manipulating programs. There are now a number of analyses and tools based on separation logic that are capable of running on industrial-scale code (see e.g. [7, 14, 20]). These tools typically limit the separation logic assertion language to the so-called *symbolic heap* fragment [6] in which only a single fixed restricted inductive predicate, defining linked list segments, is permitted. This fragment is tractable — for example, logical *entailment* becomes polynomial [17] — but the restrictions come at the cost of expressivity: analyses based on this fragment cannot effectively reason about non-list data structures.

Recently, however, there has been significant research interest in developing analyses for the fragment of separation logic in which *arbitrary* user-defined inductive predicates over symbolic heaps are permitted (see e.g. [11, 15, 21, 22]). This fragment is much more expressive than the simple linked-list fragment, but is also computationally much harder. In particular, entailment in this fragment is undecidable [3], although satisfiability is decidable [10] and entailment is decidable when predicates are restricted to have bounded treewidth [19].

In this paper, we focus on the little-considered problem of *disproving* logical entailments in the aforementioned fragment. Any proof procedure for entailment is necessarily incomplete, so the failure of proof search does not tell us whether or not an entailment is valid. A sound disproof procedure would enable us to receive "no" answers to entailment questions as well as "yes" or "don't know" answers. In particular, this has the potential to speed up proof search: we need not try to prove an entailment that is known to be invalid.

Our approach builds on the decision procedure for satisfiability in [10], which builds a summary of the models of a symbolic heap called its *base*. The base of a symbolic heap A is a finite set of *base pairs* recording, for each way of building a model of A, the variables in A that must be allocated on the heap (plus, in this paper, the "types" of the records they point to), and the equalities and disequalities that must hold. In [10] it is shown that satisfiability of a symbolic heap is exactly nonemptiness of its base. Here we go further: we attempt to disprove an entailment $A \vdash B$ by using the bases of A and B to infer the existence of a countermodel *without computing it.* This approach yields an algorithm for disproof that is both sound and terminating, but therefore necessarily incomplete.

Our method is partly reminiscent of the disproof method for separation logic (with *fractional permissions* [8] but without inductive predicates) in [18], which attempts to show that the maximum size of any model of A is strictly less than the minimum size of any model of B. However, this approach does not work well for our fragment since, if A contains an inductive predicate, its models are generally of unbounded size.

We have implemented our disproof algorithm in the CYCLIST theorem proving framework [1, 13]. Our experimental evaluation indicates that our disproof method can identify a significant proportion of invalid entailments arising in three different benchmark suites, and that it is inexpensive on average. Our algorithm might therefore be used to improve both the quality and performance of automatic theorem provers (and the program analyses relying on them).

The remainder of this paper is structured as follows. Section 2 gives an overview of our separation logic fragment, and Section 3 briefly reprises the key concept of base pairs from [10]. In Section 4 we then develop our entailment disproof method in detail. Section 5 describes the implementation of the disproof algorithm and our experimental evaluation, and Section 6 concludes.

2 Separation logic with inductive predicates

In this section we present our fragment of separation logic, which restricts the syntax of formulas to *symbolic heaps* as introduced in [5, 6], but allows arbitrary user-defined inductive predicates over these, as considered e.g. in [9–11].

We often write vector notation to abbreviate tuples, e.g. $\mathbf{x}$ for $(x_1, \ldots, x_m)$, and we write $X \# Y$, where X and Y are sets, as a shorthand for $X \cap Y = \emptyset$.

Syntax. A *term* is either a *variable* in the infinite set Var, or the constant nil. We assume a finite set $P_1, \ldots, P_n$ of *predicate symbols*, each with associated arity.

Definition 2.1. *Spatial formulas* F and *pure formulas* π are given by:

$$F ::= \mathsf{emp} \mid x \mapsto \mathbf{t} \mid P_i\mathbf{t} \mid F * F \qquad \pi ::= t = t \mid t \neq t$$

where x ranges over variables, t over terms, P_i over predicate symbols and $\mathbf{t}$ over tuples of terms (matching the arity of P_i in $P_i\mathbf{t}$). A *symbolic heap* is given by $\exists\mathbf{z}.\ \Pi : F$, where $\mathbf{z}$ is a tuple of variables, F is a spatial formula and Π is a finite set of pure formulas. Whenever one of Π, F is empty, we omit the colon. We write $FV(A)$ for the set of free variables occurring in a symbolic heap A.

Definition 2.2. An *inductive rule set* is a finite set of *inductive rules*, each of the form $A \Rightarrow P_i\mathbf{t}$, where A is a symbolic heap (called the *body* of the rule), $P_i\mathbf{t}$ is a formula (called its *head*), and all variables in $FV(A)$ appear in $\mathbf{t}$.

As usual, the inductive rules with P_i in their head should be read as exhaustive, disjunctive clauses of an inductive definition of P_i. To avoid ambiguity, we write existential quantifiers in the bodies of inductive rules explicitly, rather than leaving them implicit as is done e.g. in [10].

Semantics. We use a RAM model employing heaps of records. We assume an infinite set Val of *values* of which an infinite subset $\mathsf{Loc} \subset \mathsf{Val}$ are addressable *locations*; we insist on at least one non-addressable value $nil \in \mathsf{Val} \setminus \mathsf{Loc}$.

A *stack* is a function $s\colon \mathsf{Var} \to \mathsf{Val}$; we extend stacks to terms by setting $s(\mathsf{nil}) =_{\mathrm{def}} nil$, and write $s[z \mapsto v]$ for the stack defined as s except that $s[z \mapsto v](z) = v$. We extend stacks pointwise to act on tuples of terms.

A *heap* is a partial function $h\colon \mathsf{Loc} \rightharpoonup_{\mathrm{fin}} (\mathsf{Val\ List})$ mapping finitely many locations to *records*, i.e. arbitrary-length tuples of values; we write $\mathrm{dom}(h)$ for the set of locations on which h is defined, and e for the empty heap that is undefined everywhere. We write $\circ$ for *composition* of domain-disjoint heaps: if h_1 and h_2 are heaps, then $h_1 \circ h_2$ is the union of h_1 and h_2 when $\mathrm{dom}(h_1) \# \mathrm{dom}(h_2)$, and undefined otherwise. If $\ell \in \mathrm{dom}(h)$ then we call $|h(\ell)|$ (i.e. the length of the record $h(\ell)$) the *type* of ℓ in h, and we define the *footprint* $\mathrm{fp}(h)$ of a heap h by $\{(\ell, |h(\ell)|) \mid \ell \in \mathrm{dom}(h)\}$, i.e. by pairing each location in $\mathrm{dom}(h)$ with its type.

Definition 2.3. Given an inductive rule set Φ, the relation $s, h \models_\Phi A$ for satisfaction of a symbolic heap A by stack s and heap h is defined by:

$$
\begin{array}{lcl}
s,h \models_\Phi t_1 = t_2 & \Leftrightarrow & s(t_1) = s(t_2)\\
s,h \models_\Phi t_1 \neq t_2 & \Leftrightarrow & s(t_1) \neq s(t_2)\\
s,h \models_\Phi \mathsf{emp} & \Leftrightarrow & h = e\\
s,h \models_\Phi x \mapsto \mathbf{t} & \Leftrightarrow & \mathrm{dom}(h) = \{s(x)\} \text{ and } h(s(x)) = s(\mathbf{t})\\
s,h \models_\Phi P_i\mathbf{t} & \Leftrightarrow & (s(\mathbf{t}), h) \in [\![P_i]\!]^\Phi\\
s,h \models_\Phi F_1 * F_2 & \Leftrightarrow & \exists h_1, h_2.\ h = h_1 \circ h_2 \text{ and } s, h_1 \models_\Phi F_1 \text{ and } s, h_2 \models_\Phi F_2\\
s,h \models_\Phi \exists\mathbf{z}.\ \Pi : F & \Leftrightarrow & \begin{array}{l}\exists \mathbf{v} \in \mathsf{Val}^{|\mathbf{z}|}.\ s[\mathbf{z} \mapsto \mathbf{v}], h \models_\Phi \pi \text{ for all } \pi \in \Pi\\ \text{and } s[\mathbf{z} \mapsto \mathbf{v}], h \models_\Phi F\end{array}
\end{array}
$$

where the semantics $[\![P_i]\!]^\Phi$ of the inductive predicate P_i under Φ is defined below. We say that (s, h) is a *model* of a symbolic heap A (under Φ) if $s, h \models_\Phi A$.

The following definition gives the standard semantics of the inductive predicate symbols $\mathbf{P} = (P_1, \dots, P_n)$ as the least fixed point of an n-ary monotone operator constructed from Φ. We write π_i for the ith projection on tuples.

Definition 2.4. For each predicate $P_i \in \mathbf{P}$ with arity α_i say, we define $\tau_i = \mathrm{Pow}(\mathsf{Val}^{\alpha_i} \times \mathsf{Heap})$ (where $\mathrm{Pow}(-)$ is powerset). Next, let Φ be an inductive rule set, and partition Φ into $\Phi_1, \ldots, \Phi_n$, where Φ_i is the set of all inductive rules in Φ of the form $A \Rightarrow P_i\mathbf{x}$. Letting each Φ_i be indexed by j, for each inductive rule $\Phi_{i,j}$ of the form $\exists \mathbf{z}.\ \Pi : F \Rightarrow P_i\mathbf{x}$, we define an operator $\varphi_{i,j} : \tau_1 \times \ldots \times \tau_n \to \tau_i$:

$$\varphi_{i,j}(\mathbf{Y}) =_{\text{def}} \{(s(\mathbf{x}), h) \mid s, h \models_{\mathbf{Y}} \Pi : F\}$$

where $\mathbf{Y} \in \tau_1 \times \ldots \tau_n$ and $\models_{\mathbf{Y}}$ is the satisfaction relation given in Defn. 2.3, except that $[\![P_i]\!]^{\mathbf{Y}} =_{\text{def}} \pi_i(\mathbf{Y})$. We then define the n-tuple $[\![\mathbf{P}]\!]^{\Phi}$ by:

$$[\![\mathbf{P}]\!]^{\Phi} =_{\text{def}} \mu\mathbf{Y}.\ (\textstyle\bigcup_j \varphi_{1,j}(\mathbf{Y}), \ldots, \bigcup_j \varphi_{n,j}(\mathbf{Y}))$$

We write $[\![P_i]\!]^{\Phi}$ as an abbreviation for $\pi_i([\![\mathbf{P}]\!]^{\Phi})$.

Note that satisfaction of pure formulas depends neither on the heap nor on the inductive rules; we write $s \models \Pi$, where Π is a set of pure formulas, to mean that $s, h \models_{\Phi} \Pi$ for any heap h and inductive rule set Φ. Indeed, whether $s \models \Pi$ depends only on the values s assigns to the variables in $FV(\Pi)$, which is finite; when considering such satisfaction questions, we typically consider "partial stacks", defined in the obvious way, with finite domain denoted by $\mathrm{dom}(s)$.

3 Base pairs of symbolic heaps

In [10] it is shown how to construct a computable "summary" of the models of a symbolic heap A under any rule set Φ, called its *base* and written as $\mathit{base}^{\Phi}(A)$. Each such summary is a set of so-called *base pairs*, each of which essentially records a way of constructing models (s, h) of A under Φ, as projected onto the free variables in A. Each base pair in $\mathit{base}^{\Phi}(A)$ comprises

1. a set X of "typed" variable expressions $x : n$, where $x \in FV(A)$ and $n \in \mathbb{N}$, whose intuitive meaning is that the address $s(x)$ must be allocated with type (record length) n in h; and
2. a set Π of pure formulas (i.e. (dis)equalities) over $FV(A) \cup \{\mathsf{nil}\}$ that must be satisfied by s.

The following example is intended to illustrate the intuition behind our "base pair" summaries of symbolic heaps.

Example 3.1. Let Φ be the inductive rule set defining the standard predicates `ls` and `bt`, for linked list segments and nil-terminated binary trees respectively:

$$\begin{aligned} \mathsf{emp} &\Rightarrow \mathsf{ls}\,x\,x \\ \exists z.\ x \neq \mathsf{nil} : x \mapsto z * \mathsf{ls}\,z\,y &\Rightarrow \mathsf{ls}\,x\,y \\ x = \mathsf{nil} : \mathsf{emp} &\Rightarrow \mathsf{bt}\,x \\ \exists y, z.\ x \neq \mathsf{nil} : x \mapsto (y, z) * \mathsf{bt}\,y * \mathsf{bt}\,z &\Rightarrow \mathsf{bt}\,x \end{aligned}$$

We obtain the following bases for $\mathsf{ls}\,x\,y$ and $\mathsf{bt}\,x$:

$$\begin{array}{rcl} \mathit{base}^{\Phi}(\mathsf{ls}\,x\,y) & = & \{(\emptyset, \{x = y\}), (\{x : 1\}, \{x \neq \mathsf{nil}\})\} \\ \mathit{base}^{\Phi}(\mathsf{bt}\,x) & = & \{(\emptyset, \{x = \mathsf{nil}\}), (\{x : 2\}, \{x \neq \mathsf{nil}\})\} \end{array}$$

The intuitive reading of $\mathit{base}^{\Phi}(\mathsf{ls}\,x\,y)$ is that $s, h \models_{\Phi} \mathsf{ls}\,x\,y$ if and only if either: (a) $s \models x = y$ and neither $s(x)$ nor $s(y)$ is allocated by h; or (b) $s \models x \neq \mathsf{nil}$ and $s(x)$ is allocated with record type 1 in h.

Similarly, the intuitive reading of $\mathit{base}^{\Phi}(\mathsf{bt}\,x)$ is that $s, h \models_{\Phi} \mathsf{bt}\,x$ if and only if either: (a) $s \models x = \mathsf{nil}$ (and therefore $s(x)$ cannot be allocated in h); or (b) $s \models x \neq \mathsf{nil}$ and $s(x)$ is allocated with record type 2 in h.

The set $\mathit{base}^{\Phi}(A)$ is always finite, since $FV(A)$ is finite and the maximum type numeral of any allocated location in a model of A can be shown to be finite as well. The full details[3] of the construction of $\mathit{base}^{\Phi}(A)$ can be found in [10]. However, for the purposes of the present paper, these details are in fact not especially relevant. The information from [10] that we *do* however rely on is (a) the fact that $\mathit{base}^{\Phi}(A)$ is computable, and (b) the precise relationship between $\mathit{base}^{\Phi}(A)$ and the models of A under Φ. The latter is captured formally by the following pair of technical results, where we define $s(x : n) = (s(x), n)$, and extend by pointwise union to sets.

Lemma 3.2 (Soundness [10]). *Given a base pair $(X, \Pi) \in \mathit{base}^{\Phi}(A)$, a stack s such that $s \models \Pi$, and a finite "footprint" $W \subset \mathsf{Loc} \times \mathbb{N}$ such that $W \# s(X)$, one can construct a heap h such that $s, h \models_{\Phi} A$ and $W \# \mathrm{fp}(h)$.*

Lemma 3.3 (Completeness [10]). *If $s, h \models_{\Phi} A$, there is a base pair $(X, \Pi) \in \mathit{base}^{\Phi}(A)$ such that $s(X) \subseteq \mathrm{fp}(h)$ and $s \models \Pi$.*

An immediate consequence of Lemmas 3.2 and 3.3, used in [10], is that *satisfiability* of a symbolic heap A, i.e. the existence of at least one model of A, exactly corresponds to nonemptiness of $\mathit{base}^{\Phi}(A)$, and is therefore decidable.

4 An algorithm for entailment disproof

In this section, we develop the main contribution of our paper: an algorithm for disproving *entailments* in our separation logic fragment.

Definition 4.1. An *entailment* is given by $A \vdash B$, where A and B are symbolic heaps. The entailment $A \vdash B$ is said to be *valid* if for all stacks s and heaps h it holds that $s, h \models_{\Phi} A$ implies $s, h \models_{\Phi} B$, and *invalid* otherwise.

Thus, as usual, to *disprove* (i.e. show invalid) an entailment $A \vdash B$, we need to exhibit a *countermodel* (s, h) such that $s, h \models_{\Phi} A$ but $s, h \not\models_{\Phi} B$. Unfortunately,

[3] In fact, the original construction does not include the record types of allocated variables in its base pairs, but the required adaptations are quite straightforward.

this is not straightforward, since the entailment problem for our fragment of separation logic is undecidable [3].

One naive approach would simply be to generate and test possible counter-models (s, h) of increasing heap "sizes" (defined in some reasonable way). This approach has only just become potentially viable at the time of going to press, following the very recent development of a model checking procedure for our logic [12]. However, this approach still presents some fairly significant obstacles. Firstly, the generation of possible counter-models is not simply a matter of blind enumeration, since the values of stack variables, the addresses of allocated heap cells and the contents of those cells all range over *infinite* sets (Val and Loc). That is to say, there are infinitely many distinct models of a given size, and so some quotienting over these values is required so as to restrict these models to finitely many "representative cases". Secondly, this approach also seems likely to be quite expensive even in average cases, since the model checking problem itself, according to [12], is $\mathsf{EXPTIME}$-complete: Many models would be generated and most of them would inevitably fail to be countermodels (e.g., for the trivial reason that they do not satisfy A). Finally, any complete enumeration-based approach will, in general, fail to terminate.

However, the technical lemmas in Section 3 relating the base of a symbolic heap to its models suggest an alternative way in which we can nevertheless proceed. Lemma 3.2 tells us that we can construct a model (s, h) of A by choosing a base pair (X, Π) of A, an s that satisfies Π and a footprint W, disjoint from $s(X)$, to be "avoided" by the footprint of h. Lemma 3.3 then tells us that if $s, h \models_{\Phi} B$ then we can find a base pair (Y, Θ) of B with which (s, h) is "consistent", in that s satisfies Θ and the footprint of h covers $s(Y)$. Thus if we can construct a model of A with which *no* base pair of B is consistent, then this model is a counter-model. We first formulate this idea directly as a simple two-player game, and then refine this game into an implementable form.

4.1 Disproof via base pair games

In the following, we assume a fixed inductive rule set Φ. We extend the function $FV(-)$ to base pairs by $FV((X, \Pi)) =_{\text{def}} \bigcup_{x:n \in X} \{x\} \cup FV(\Pi)$, and then by pointwise union to sets of base pairs.

Game 1. Given an entailment $A \vdash B$, we define a simple two-player game as follows. A *move* by Player 1 is a tuple $((X, \Pi), s, W)$ obtained by choosing:

- a base pair $(X, \Pi) \in \mathit{base}^{\Phi}(A)$;
- a partial stack $s : (FV((X, \Pi)) \cup FV(\mathit{base}^{\Phi}(B))) \to \mathsf{Val}$ such that $s \models \Pi$;
- and a finite footprint $W \subset \mathsf{Loc} \times \mathbb{N}$ such that $W \,\#\, s(X)$.

A *response* by Player 2 to such a move is a base pair $(Y, \Theta) \in \mathit{base}^{\Phi}(B)$ such that $s \models \Theta$ and $W \,\#\, s(Y)$.

A move is said to be a *winning move* if there is no possible response to it.

As a game, Game 1 is not especially interesting, as any game can be won by Player 1 either in one move or not at all. Our formulation is for convenience.

Proposition 4.2. *If Player 1 has a winning move for $A \vdash B$ in Game 1 then $A \vdash B$ is invalid.*

Proof. Let $((X, \Pi), s, W)$ be a winning move for $A \vdash B$. That is, for some base pair (X, Π) of A we have a partial stack s such that $s \models \Pi$ and a finite footprint W with $W \# s(X)$. By Lemma 3.2, there exists a heap h such that $s, h \models_{\Phi} A$ and $W \# \mathrm{fp}(h)$.

Now suppose for contradiction that $A \vdash B$ is valid. Thus, as $s, h \models_{\Phi} A$, we have $s, h \models_{\Phi} B$. By Lemma 3.3, there exists a base pair (Y, Θ) of B such that $s(Y) \subseteq \mathrm{fp}(h)$ and $s \models \Theta$. As $W \# \mathrm{fp}(h)$ and $s(Y) \subseteq \mathrm{fp}(h)$, we have $W \# s(Y)$. Thus (Y, Θ) is a response to a winning move, contradiction. $\square$

Our formulation of Game 1 exploits Lemmas 3.2 and 3.3 in a way that is intended to be maximally general, but it cannot be directly implemented as a terminating algorithm: Player 1 has to choose a partial stack with finite domain but infinite codomain, and an arbitrary finite footprint $W \subset \mathsf{Loc} \times \mathbb{N}$. However, we can reformulate Game 1 so as to entirely obviate the latter difficulty.

Game 2. Given an entailment $A \vdash B$, a *move* by Player 1 is a tuple $((X, \Pi), s)$ obtained by choosing:

- a base pair $(X, \Pi) \in \mathit{base}^{\Phi}(A)$, and
- a partial stack $s : (FV((X, \Pi)) \cup FV(\mathit{base}^{\Phi}(B))) \to \mathsf{Val}$ such that $s \models \Pi$.

Given such a move, a *response* by Player 2 is a base pair $(Y, \Theta) \in \mathit{base}^{\Phi}(B)$ such that $s \models \Theta$ and $s(Y) \subseteq s(X)$. A *winning move* is defined as for Game 1.

Lemma 4.3. *Player 1 has a winning move for $A \vdash B$ in Game 2 if and only if she has a winning move for $A \vdash B$ in Game 1.*

Proof. ($\Leftarrow$) Let $((X, \Pi), s, W)$ be a winning move for $A \vdash B$ in Game 1. That is, we have a base pair (X, Π) of A, a partial stack s and a finite footprint W such that $s \models \Pi$ and $W \# s(X)$; moreover, there is no response to this move.

The required winning move for $A \vdash B$ in Game 2 is then given by $((X, \Pi), s)$. Suppose for contradiction that $(Y, \Theta) \in \mathit{base}^{\Phi}(B)$ is a response to this move, i.e., $s \models \Theta$ and $s(Y) \subseteq s(X)$. As $W \# s(X)$ and $s(Y) \subseteq s(X)$, we have $W \# s(Y)$. As $s \models \Theta$ and $W \# s(Y)$, the base pair (Y, Θ) is a response to the winning move $((X, \Pi), s, W)$ for $A \vdash B$ in Game 1, contradiction.

($\Rightarrow$) Let $((X, \Pi), s)$ be a winning move for $A \vdash B$ in Game 2. That is, we have a base pair (X, Π) of A and a partial stack s such that $s \models \Pi$; moreover, there is no response to this move. We claim that $((X, \Pi), s, W)$ is a winning move for $A \vdash B$ in Game 1, where we choose the finite footprint $W \subset \mathsf{Loc} \times \mathbb{N}$ as follows:

$$W =_{\mathrm{def}} \left(\bigcup_{(Y,\Theta) \in \mathit{base}^{\Phi}(B)} s(Y) \right) \setminus s(X)$$

Now $W \# s(X)$ by construction, so $((X, \Pi), s, W)$ is certainly *a* valid move in Game 1. To see that it is a *winning* move, suppose for contradiction that Player

2 has a response to this move, that is, a base pair (Y, Θ) of B with $s \models \Theta$ and $W \# s(Y)$. By construction, $s(Y) \setminus s(X) \subseteq W$, so $s(Y) \setminus s(X) \# s(Y)$. This implies $s(Y) \setminus s(X) = \emptyset$ and thus $s(Y) \subseteq s(X)$. Thus (Y, Θ) is a response to the winning move $((X, \Pi), s)$ for $A \vdash B$ in Game 2, contradiction. $\square$

We now give an example of how Game 2 works in practice.

Example 4.4. Let Φ define the linked list predicate ls given in Example 3.1, and consider the invalid entailment $\mathsf{ls}\, x\, y \vdash \mathsf{ls}\, y\, x$. We have the following bases:

$$\begin{array}{rcl} \mathit{base}^{\Phi}(\mathsf{ls}\, x\, y) & = & \{(\emptyset, \{x = y\}), (\{x : 1\}, \{x \neq \mathsf{nil}\})\} \\ \mathit{base}^{\Phi}(\mathsf{ls}\, y\, x) & = & \{(\emptyset, \{y = x\}), (\{y : 1\}, \{y \neq \mathsf{nil}\})\} \end{array}$$

Then Player 1 has a winning move in Game 2 by choosing her second base pair $(\{x : 1\}, \{x \neq \mathsf{nil}\})$ together with any stack s in which $s(x) \neq \mathit{nil}$ and $s(x) \neq s(y)$. The first constraint is required to validate Player 1's move, and the second rules out both of Player 2's base pairs as responses: for the first pair $s \not\models y = x$, and for the second we have $s(\{y : 1\}) \not\subseteq s(\{x : 1\})$.

As Example 3.1 suggests, we can refine Game 2 further: instead of a (partial) stack, Player 1 can simply choose a *partition* of the stack domain.

Definition 4.5. Let σ be a partition of a set of terms T. Then, for $t, t' \in T$, we write $\sigma \models t = t'$ to mean that t and t' are in the same σ-equivalence class, and $\sigma \models t \neq t'$ otherwise. This relation extends conjunctively to sets of pure formulas over T.

Lemma 4.6. *For any partial stack s, we can construct a partition σ_s of* $\mathrm{dom}(s) \cup \{\mathsf{nil}\}$ *such that, for any set Π of pure formulas with $FV(\Pi) \subseteq \mathrm{dom}(s)$,*

$$s \models \Pi \;\Leftrightarrow\; \sigma_s \models \Pi \;.$$

Conversely, for any partition σ of a finite set T of terms we can construct a partial stack s_σ such that, for any set Π of pure formulas with $FV(\Pi) \subseteq T$,

$$s_\sigma \models \Pi \;\Leftrightarrow\; \sigma \models \Pi \;.$$

Proof. For the first part of the lemma, we simply put t and t' in the same σ-equivalence class if $s(t) = s(t')$ and in different classes otherwise. By construction, for a pure formula of the form $t = t'$,

$$s \models t = t' \;\Leftrightarrow\; s(t) = s(t') \;\Leftrightarrow\; \sigma_s \models t = t' \;,$$

and similarly for formulas of the form $t \neq t'$.

For the second part, we construct s_σ simply by mapping terms in the same σ-equivalence class to the same value in Val, and terms in different classes to distinct values. This is always possible since the range Val of our stacks is infinite. Then we just observe that for a pure formula of the form $t = t'$, we have,

$$s_\sigma \models t = t' \;\Leftrightarrow\; s_\sigma(t) = s_\sigma(t') \;\Leftrightarrow\; \sigma \models t = t' \;,$$

and similarly for formulas of the form $t \neq t'$. $\square$

Game 3. Given an entailment $A \vdash B$, a *move* by Player 1 is a choice of:

- a base pair $(X, \Pi) \in \mathit{base}^{\Phi}(A)$, and
- a partition σ of $FV((X, \Pi)) \cup FV(\mathit{base}^{\Phi}(B)) \cup \{\mathsf{nil}\}$ such that $\sigma \models \Pi$.

Given such a move, a *response* by Player 2 is a base pair $(Y, \Theta) \in \mathit{base}^{\Phi}(B)$ such that $\sigma \models \Theta$ and for any $y : n \in Y$ there is $x : n \in X$ such that $\sigma \models x = y$.

A *winning move* is defined as for the previous games.

Lemma 4.7. *Player 1 has a winning move for $A \vdash B$ in Game 3 if and only if she has a winning move for $A \vdash B$ in Game 2.*

Proof. ($\Leftarrow$) Let $((X, \Pi), s)$ be a winning move for $A \vdash B$ in Game 2. That is, for some base pair (X, Π) of A we have a partial stack s such that $s \models \Pi$, and moreover there is no response to this move.

We claim that $((X, \Pi), \sigma_s)$ is then a winning move in Game 3, where σ_s is the the partition σ_s of $\mathrm{dom}(s) \cup \{\mathsf{nil}\}$ given by the first part of Lemma 4.6. Since $s \models \Pi$, the lemma guarantees that $\sigma_s \models \Pi$, so $((X, \Pi), \sigma_s)$ is indeed a move. To see that it is a winning move, suppose for contradiction that $(Y, \Theta) \in \mathit{base}^{\Phi}(B)$ is a response to this move, i.e., $\sigma_s \models \Theta$ and $\exists x : n \in X.\ \sigma_s \models x = y$ whenever $y : n \in Y$. We claim that (Y, Θ) is then a response to the winning move $((X, \Pi), s)$ in Game 2. Since $\sigma_s \models \Theta$, we have $s \models \Theta$ by the first part of Lemma 4.6. It just remains to show that $s(Y) \subseteq s(X)$. Let $y : n \in Y$. By assumption, there exists $x : n \in X$ such that $\sigma_s \models x = y$. By Lemma 4.6, we have $s \models x = y$, i.e. $s(x : n) = s(y : n)$, and so $s(y : n) \in s(X)$. Thus $s(Y) \subseteq s(X)$, which completes the case.

($\Rightarrow$) Let $((X, \Pi), \sigma)$ be a winning move for $A \vdash B$ in Game 3. That is, for some base pair (X, Π) of A we have a partition σ such that $\sigma \models \Pi$, and moreover there is no response to this move.

We define a winning move in Game 2 by $((X, \Pi), s_\sigma)$, where s_σ is the partial stack constructed from σ by the second part of Lemma 4.6. Since $\sigma \models \Pi$, the lemma guarantees that $s_\sigma \models \Pi$ as required. Suppose for contradiction $(Y, \Theta) \in \mathit{base}^{\Phi}(B)$ is a response to this move, i.e., $s_\sigma \models \Theta$ and $s_\sigma(Y) \subseteq s_\sigma(X)$. We claim that (Y, Θ) is then a response to the winning move $((X, \Pi), \sigma)$ in Game 3. First, since $s_\sigma \models \Theta$, we have $\sigma \models \Theta$ by the second part of Lemma 4.6. Now, letting $y : n \in Y$, we have to show there exists an $x : n \in X$ such that $\sigma \models x = y$. Since $s_\sigma(Y) \subseteq s_\sigma(X)$ and $y : n \in Y$, there exists $x : n \in X$ such that $s_\sigma(y) = s_\sigma(x)$, i.e. $s_\sigma \models x = y$. By Lemma 4.6, we then have $\sigma \models x = y$, as required. $\square$

Example 4.8. Let Φ define ls and bt from Example 3.1. We have:

$$\begin{array}{rcl} \mathit{base}^{\Phi}(\mathsf{ls}\, x\, \mathsf{nil}) & = & \{(\emptyset, \{x = \mathsf{nil}\}), (\{x : 1\}, \{x \neq \mathsf{nil}\})\} \\ \mathit{base}^{\Phi}(\mathsf{bt}\, x) & = & \{(\emptyset, \{x = \mathsf{nil}\}), (\{x : 2\}, \{x \neq \mathsf{nil}\})\} \end{array}$$

Now, both $\mathsf{bt}\, x \vdash \mathsf{ls}\, x\, \mathsf{nil}$ and $\mathsf{ls}\, x\, \mathsf{nil} \vdash \mathsf{bt}\, x$ are invalid, and Player 1 has a winning move for both entailments in Game 3 by choosing her second base pair $(\{x : i\}, \{x \neq \mathsf{nil}\})$, where $i \in \{1, 2\}$, together with a partition σ such that $\sigma \models x \neq \mathsf{nil}$.

Player 2 cannot respond with his first base pair because $\sigma \not\models x = \mathsf{nil}$, nor with his second because the type of x does not match that in Player 1's pair.

Theorem 4.9. *Games 1, 2 and 3 are all equivalent to each other, and decidable. That is, for any entailment $A \vdash B$ we can decide which player wins, and this answer is consistent across all three games.*

Proof. Equivalence is an immediate consequence of Lemmas 4.3 and 4.7. For decidability, it suffices to observe just that Game 3 is decidable for any $A \vdash B$. As there are only finitely many base pairs (X, Π) of A and for each of these only finitely many partitions of the finite set $FV((X, \Pi)) \cup FV(\mathit{base}^{\Phi}(B)) \cup \{\mathsf{nil}\}$, there are only finitely many possible moves for Player 1. Moreover, for each such move there are only finitely many possible responses by Player 2, since $\mathit{base}^{\Phi}(B)$ is finite. Hence checking whether or not Player 1 has a winning move is simply a case of checking the finitely many possibilities. $\square$

It is informative to examine the kinds of entailments our method cannot, in principle, recognise as invalid. We can only disprove entailments $A \vdash B$ in which B imposes allocation or (dis)equality requirements on its free variables which can be violated by models of A. For example, the entailment $x \mapsto \mathsf{nil} \vdash \mathsf{emp}$ is invalid, while $x \mapsto \mathsf{nil} \vdash \exists y.\ y \mapsto \mathsf{nil}$ is valid, but our base pair approximation cannot distinguish between the two because neither RHS has any free variables: we have $\mathit{base}^{\Phi}(\mathsf{emp}) = \mathit{base}^{\Phi}(\exists y.\ y \mapsto \mathsf{nil}) = \{(\emptyset, \emptyset)\}$. The base pair construction also discards information on bounds, such as the number of allocated cells in a heap; therefore, for example, we cannot distinguish between an even-length list and an odd-length one.

4.2 Efficiency considerations

Having established that Game 3 is a sound and terminating algorithm for disproving entailments (Theorem 4.9), we now consider possible ways of improving its efficiency. First, we give an upper bound for the worst-case runtime.

Proposition 4.10. *Checking whether Player 1 has a winning strategy for $A \vdash B$ in Game 3 can be done in time exponential in the size of A, B and the definitions of the predicates in the underlying inductive rule set Φ.*

Proof. First, the number of base pairs for any symbolic heap is, in the worst case, exponential in the size of the symbolic heap and its predicate definitions [10]. Second, the number of partitions σ over $FV((X, \Pi)) \cup FV(\mathit{base}^{\Phi}(B)) \cup \{\mathsf{nil}\}$ where $(X, \Pi) \in \mathit{base}^{\Phi}(A)$, is bounded by an exponential in the size of A and B. Finally, checking whether a base pair $(Y, \Theta) \in \mathit{base}^{\Phi}(B)$ is a response to a move $((X, \Pi), \sigma)$ can be performed in polynomial time. Thus, searching for a winning move for Player 1 can take up to exponential time in the size of A, B and the predicate definitions in Φ. $\square$

Next, we give some simple results identifying redundant base pairs in our game instances. If Π and Π' are sets of pure formulas we write $\Pi \models \Pi'$ to mean that $\Pi \vdash \Pi'$ is valid, i.e. $\sigma \models \Pi$ implies $\sigma \models \Pi'$ for all partitions σ.

Definition 4.11. *If (X,Π) and (X',Π') are both base pairs (of some symbolic heap) then we write $(X,\Pi) \sqsubseteq (X',\Pi')$ to mean that $\Pi' \models \Pi$ and for any $x : n \in X$ there is an $x' : n \in X'$ such that $\Pi' \models x = x'$. We write $(X,\Pi) \sim (X',\Pi')$ to mean that $(X,\Pi) \sqsubseteq (X',\Pi')$ and $(X',\Pi') \sqsubseteq (X,\Pi)$.*

Clearly $\sim$ is an equivalence on base pairs, and $\sqsubseteq$ is a partial order up to $\sim$.

Proposition 4.12. *The following hold for any entailment $A \vdash B$ in Game 3:*

1. *Let $(X,\Pi), (X',\Pi') \in base^{\Phi}(A)$ with $(X,\Pi) \sqsubseteq (X',\Pi')$. If $((X',\Pi'),\sigma)$ is a winning move then so is $((X,\Pi),\sigma)$.*
2. *Let $(Y,\Theta), (Y',\Theta') \in base^{\Phi}(B)$ with $(Y,\Theta) \sqsubseteq (Y',\Theta')$. If (Y',Θ') is a response to the move $((X,\Pi),\sigma)$ then so is (Y,Θ).*
3. *Let $(X,\Pi) \in base^{\Phi}(A)$, $(Y,\Theta) \in base^{\Phi}(B)$ with $(Y,\Theta) \sqsubseteq (X,\Pi)$. Then (Y,Θ) is a response to* any *move of the form $((X,\Pi),\sigma)$.*

Therefore, without loss of generality, we may remove all base pairs from $base^{\Phi}(A)$ and $base^{\Phi}(B)$ that are not $\sqsubseteq$-minimal, and any $\sim$-duplicates; and we may also remove all $(X,\Pi) \in base^{\Phi}(A)$ that are not $\sqsubseteq$-minimal with respect to $base^{\Phi}(B)$.

Proof. 1. First note that $\sigma \models \Pi'$ and $\Pi' \models \Pi$ by assumption, so $\sigma \models \Pi$, and thus $((X,\Pi),\sigma)$ is a valid move. To see that it is a winning move, suppose for contradiction that (Y,Θ) is a response to it. We show for contradiction that (Y,Θ) is also a response to $((X',\Pi'),\sigma)$. First, $\sigma \models \Theta$ by assumption. Now let $y : n \in Y$. By assumption, there is an $x : n \in X$ such that $\sigma \models x = y$. As $(X,\Pi) \sqsubseteq (X',\Pi')$, there is an $x' : n \in X'$ such that $\Pi' \models x = x'$. As $\sigma \models \Pi'$ it follows that $\sigma \models x' = y$, as required.
2. We show that (Y,Θ) is a response to $((X,\Pi),\sigma)$. First, by assumption we have $\sigma \models \Theta'$ and $\Theta' \models \Theta$, so $\sigma \models \Theta$ as required. Now let $y : n \in Y$. As $(Y,\Theta) \sqsubseteq (Y',\Theta')$, there is $y' : n \in Y'$ such that $\Theta' \models y' = y$, and thus $\sigma \models y' = y$. By assumption, for any $y' : n \in Y'$ there is $x : n \in X$ such that $\sigma \models y' = x$. Thus we have $x : n \in X$ such that $\sigma \models x = y$, as required.
3. First we have to check that $\sigma \models \Theta$, which follows from $\sigma \models \Pi$ and $\Pi \models \Theta$. Now let $y : n \in Y$. Since $(Y,\Theta) \sqsubseteq (X,\Pi)$, there is $x : n \in X$ such that $\Pi \models x = y$. As $\sigma \models \Pi$ by assumption, $\sigma \models x = y$, as required. □

A major source of complexity in Game 3 is the need to consider all possible partitions of a set of variables (plus nil) for any given base pair of A in order to obtain all possible moves for Player 1. The number of partitions of a set of size n is given by the nth Bell number [4], which grows extremely quickly in n. Fortunately, as our final theorem shows, we may regard certain pairs of terms as nonequal by default, which can potentially reduce the search space.

Theorem 4.13. *Suppose Player 1 has a winning move $((X,\Pi),\sigma)$ for $A \vdash B$ (in Game 3). Then there is also a winning move of the form $((X,\Pi),\sigma')$ where the partition σ' satisfies the following constraint:*

If t,u are distinct terms in $FV((X,\Pi)) \cup FV(base^{\Phi}(B)) \cup \{\mathsf{nil}\}$, then $\sigma' \models t \neq u$ whenever both of the following hold:

1. $\Pi \not\models t = u$; and
2. for all base pairs $(Y, \Theta) \in base^{\Phi}(B)$ and disequalities $v \neq w \in \Theta$, we have $\Pi \models t = v$ if and only if $\Pi \models t = w$.

Proof. First, for any partition σ over $FV((X, \Pi)) \cup FV(base^{\Phi}(B)) \cup \{\mathsf{nil}\}$ we define the set $BadEqs(\sigma)$ to be the set of all pairs of terms (t, u) such that $\sigma \models t = u$ and t, u satisfy the constraints 1 and 2 above. By induction, it then suffices to show that we can construct a partition σ' such that $((X, \Pi), \sigma')$ is a winning move for Player 1 and $BadEqs(\sigma') \subset BadEqs(\sigma)$, provided $BadEqs(\sigma) \neq \emptyset$.

Now, letting $(t, u) \in BadEqs(\sigma)$, we write $[t]_\sigma$ for the σ-equivalence class of t, i.e., $\{t' \mid \sigma \models t' = t\}$. We then define a new partition σ' obtained from σ by further dividing $[t]_\sigma$ into the following two subpartitions:

$$P_1 =_{\text{def}} \{t' \mid \Pi \models t' = t\} \quad \text{and} \quad P_2 =_{\text{def}} [t]_\sigma \setminus P_1$$

We observe that this is indeed a non-trivial partitioning of $[t]_\sigma$. On the one hand, we trivially have $t \in P_1$ and, since $\sigma \models \Pi$ by assumption, we have $t' \in [t]_\sigma$ whenever $\Pi \models t' = t$. On the other hand, we have $u \in P_2$ because $\Pi \not\models u = t$ according to constraint 1. Furthermore, we have $BadEqs(\sigma') \subset BadEqs(\sigma)$ because, by construction, $(t, u) \notin BadEqs(\sigma')$ and σ' differs from σ only in the subdivision of the equivalence class $[t]_\sigma$.

Now we require to show that $((X, \Pi), \sigma')$ is a winning move for Player 1. First, we have to check that it is a valid move at all, i.e., that $\sigma' \models \Pi$. We check that σ' satisfies each equality and disequality in Π. If $v \neq w \in \Pi$ then, since $\sigma \models \Pi$, we have $\sigma \models v \neq w$. By construction of σ', we clearly then also have $\sigma' \models v \neq w$ as required. For $v = w \in \Pi$, then we have $\sigma \models v = w$ by assumption and, by construction of σ', we also have $\sigma' \models v = w$ unless it happens that $v \in P_1$ while $w \in P_2$ (or $w \in P_1$ and $v \in P_2$, which is symmetric). In that case, since $v = w \in \Pi$ we trivially have $\Pi \models v = w$, and since $v \in P_1$ we have $\Pi \models v = t$, and so $\Pi \models w = t$. This means that $w \in P_1$, which contradicts $w \in P_2$. Thus indeed we have $\sigma' \models v = w$ as required.

It remains to show that $((X, \Pi), \sigma')$ is indeed a *winning* move. Suppose for contradiction that (Y, Θ) is a response to this move. It suffices to show that (Y, Θ) is then also a response to the original $((X, \Pi), \sigma)$. First we have to show that $\sigma \models \Theta$. We check that σ satisfies each equality and disequality in Θ. For $v = w \in \Theta$ we have $\sigma' \models v = w$ since $\sigma' \models \Theta$ by assumption. By construction of σ', we then clearly have $\sigma \models v = w$ as required. For $v \neq w \in \Theta$, we have $\sigma' \models v \neq w$ by assumption and, again by construction, we have $\sigma \models v \neq w$ unless it happens that $v \in P_1$ while $w \in P_2$ (or vice versa, which is symmetric). In that case, we have $\Pi \models t = v$ while $\Pi \not\models t = w$. This situation is precisely excluded by constraint 2. Finally, we have to check that for all $y : n \in Y$, there is an $x : n \in X$ with $\sigma \models x = y$. Let $y : n \in Y$. By assumption, there is $x : n \in X$ such that $\sigma' \models x = y$. Hence, by construction of σ', we immediately have $\sigma \models x = y$ too. This completes the proof. □

5 Experimental evaluation

Implementation and experimental framework. Our method for checking invalidity, using Game 3 and the optimisations given by Proposition 4.12 and Theorem 4.13, has been implemented in OCaml (openly available at [1]). We used the theorem prover CYCLIST as the basis for our implementation, as it provides facilities for separation logic entailments with inductive predicates [13], including a procedure for computing the base pairs of formulas [10].

Finding benchmark entailments that have known validity status (so as to assess precision), and which are ostensibly relevant to the needs of program analysis frontends, is challenging. Currently, the main such source of test cases is the Separation Logic Competition (SL-COMP)[2]. In addition to these benchmarks, we provide a large new synthetic test suite (LEM) designed to exercise our disprover over cases that are in some sense “typical”. The three classes of test cases we consider are as follows:

UDP: This is the class of entailments from SL-COMP that is most relevant to our logical fragment. It comprises 172 mostly hand-crafted sequents employing various user-defined inductive predicates representing singly- and doubly-linked lists, skip lists, trees and other structures. Unfortunately, however, only 20 sequents in the UDP set are invalid.

LEM: As invalid sequents are badly under-represented in the UDP benchmarks, we generated a large synthetic test suite in the following way. First, we took the inductive predicate definitions from the UDP suite, amounting to 63 distinct predicates. Then, for every pair of distinct predicates P, Q in this set we form the sequent $P\mathbf{x} \vdash Q\mathbf{y}$ where $\mathbf{x}$ is a tuple of distinct variables and $\mathbf{y}$ is any possible tuple of variables from $\mathbf{x}$ matching the arity of Q. This yields 818988 entailments, of which we would expect most to be invalid. Entailments of this kind are typical of *automated theory exploration* (see e.g. [16]), where potential lemmas are generated bottom-up from the definitions of the theory and, if proven valid, added to a lemma library. Such approaches rely heavily on relatively cheap methods of filtering out the many invalid “lemmas”.

SLL: Finally, this class, also from SL-COMP, consists of 292 entailments (produced by program analysis tools, by hand and by random generation) involving only a single inductive predicate denoting possibly empty, acyclic, singly-linked list segments. Validity for entailments in this fragment are already known to be polynomially decidable [17], whereas our procedure is much more general but incomplete, so we included these benchmarks mainly as a way of checking the soundness of our procedure.

All tests were performed on an Intel i5-3570 CPU running at 3.4GHz with 8Gb of RAM running Linux and a 60-second time-out.

Soundness. Of all test cases in the UDP and SLL test suites, where the validity status of test cases has been independently checked, we encountered one apparent false positive, where an entailment in UDP was disproved by our implementation but marked as valid. This entailment is over possibly-empty, acyclic, doubly-linked list segments, given by the predicate dll defined as follows:

$$x = z, y = w : \mathsf{emp} \Rightarrow \mathsf{dll}(x, y, w, z)$$
$$\exists u'.\ x \neq z, y \neq w : x \mapsto (u', w) * \mathsf{dll}(u', y, x, z) \Rightarrow \mathsf{dll}(x, y, w, z)$$

The entailment which was disproved but marked in UDP as valid is:

$$x \neq w, w \neq t, w \neq z : w \mapsto (t, u) * \mathsf{dll}(x, u, \mathsf{nil}, w) * \mathsf{dll}(t, y, w, z) \vdash \mathsf{dll}(x, y, \mathsf{nil}, z)$$

In fact, the above entailment is *not* valid. There is a model of the LHS where the subformula $\mathsf{dll}(t, y, w, z)$ represents a segment of length two (or more), thus setting $y \neq \mathsf{nil}$. At the same time, x can alias z; thus there is a Player 1 move where $y \neq \mathsf{nil}$ and $x = z$. Player 2 cannot respond to this move because the RHS allows either $x = z, y = \mathsf{nil}$ *or* $x \neq z, y \neq \mathsf{nil}$. Concrete countermodels are those that satisfy the following formula.

$$x = z : x \mapsto (u, \mathsf{nil}) * u \mapsto (w, x) * w \mapsto (t, u) * t \mapsto (y, w) * y \mapsto (z, t)$$

This benchmark bug was confirmed and fixed by the SL-COMP maintainers.

Benchmark	Count	# Invalid	Precision	Timeouts
UDP	172	20	50%	3%
LEM	818988	?	>97%	0%
SLL	292	120	24%	7%

Fig. 1. Precision and timeouts (>60s) for the UDP, SLL and LEM benchmark classes.

Precision and performance. Figure 1 summarises the experimental results on the precision and efficiency of our method.

In the UDP suite our method disproves 10 of 20 invalid sequents. The heuristic timed-out on only 3% of all sequents, analysed nearly 80% of sequents in time less than 1 millisecond and nearly 95% in fewer than 100 milliseconds.

For the LEM test suite, our method disproved 800667 of 818988 test entailments, or 97.7%. Strictly speaking, this is only a measure of precision if one assumes our implementation is correct, as these entailments have not been manually checked. However, under such an assumption, the above figure can be taken as a lower bound to precision on LEM. Indeed, since we expect most entailments in LEM to be invalid, this figure is likely near the actual precision. No test case in the LEM suite required more than 30 milliseconds for analysis.

Only 24% of invalid sequents in the SLL set were disproved. A manual inspection of the invalid entailments in both SLL and UDP not disproved by our implementation revealed that, as expected, they fall into the category described at the end of section 4.1, where the RHS imposes very weak constraints on its free variables. Time-outs were observed only on large invalid sequents, comprising 7% of all test cases having 33–109 atomic formulas and 12–20 list predicate occurrences each. More than 50% of test cases require time less than 1 ms.

Overall, given the very low cost overhead of our disprover, we believe its precision represents a good value proposition; the LEM performance shows that this should especially be the case when exploring large spaces of entailments, e.g. in automated proof search or automated theory exploration. We note that one would never run a general prover or disprover on SLL entailments in practice, since the PTIME decision procedure for this fragment is, essentially, optimal.

6 Conclusion and future work

Our main contribution in this paper is an algorithm for detecting invalid entailments in the symbolic heap fragment of separation logic with user-defined inductive predicates. Our method is sound and terminating, but necessarily incomplete. However, our experiments show that we can identify a non-trivial proportion of invalid entailments that typically occur in practice. Moreover, our method is very inexpensive compared to the typically high cost of proof search; therefore, we believe there is very little reason not to use it.

Our analysis essentially works by comparing the *bases* of symbolic heaps, as introduced to check satisfiability in [10]. These bases abstract away a great deal of information about the precise shape of models, and so there is a fundamental limitation on the amount of information that can be obtained by comparing them; unavoidably, there are many invalid entailments that our method fails to recognise. To improve the precision of our analysis, one might refine the base pair construction further to retain more information about the shape of underlying models (while remaining within the bounds of computability), or seek to develop entirely separate heuristics designed to complement our method.

Another possible line of future work, building on the very recent development of a *model checking* procedure for our logic [12], is to explore the possibility of disproving entailments by directly generating and checking potential countermodels. However, such an analysis might be significantly more expensive than the one we present here.

To the best of our knowledge, invalidity questions have been rather less well studied than validity questions in the separation logic literature to date. We hope that the present paper will serve to stimulate wider interest in such questions, and techniques for addressing them.

References

1. Cyclist: software distribution for this paper.
`https://github.com/ngorogiannis/cyclist/releases/tag/TABLEAUX15`.
2. The first Separation Logic Competition (SL-COMP14).
`http://www.liafa.univ-paris-diderot.fr/~sighirea/slcomp14/`.

3. T. Antonopoulos, N. Gorogiannis, C. Haase, M. Kanovich, and J. Ouaknine. Foundations for decision problems in separation logic with general inductive predicates. In *Proc. FoSSaCS-17*, pages 411–425. Springer, 2014.
4. E. Bell. Exponential numbers. *The American Mathematical Monthly*, 41(7):411–419, 1934.
5. J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *Proc. FSTTCS-24*, pages 97–109. Springer, 2004.
6. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Proc. APLAS-3*, pages 52–68. Springer, 2005.
7. J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: memory safety for systems-level code. In *Proc. CAV-23*, pages 178–183. Springer, 2011.
8. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Proc. POPL-32*, pages 59–70. ACM, 2005.
9. J. Brotherston. Formalised inductive reasoning in the logic of bunched implications. In *Proc. SAS-14*, pages 87–103. Springer, 2007.
10. J. Brotherston, C. Fuhs, N. Gorogiannis, and J. Navarro Pérez. A decision procedure for satisfiability in separation logic with inductive predicates. In *Proc. CSL-LICS*, pages 25:1–25:10. ACM, 2014.
11. J. Brotherston and N. Gorogiannis. Cyclic abduction of inductively defined safety and termination preconditions. In *Proc. SAS-21*, pages 68–84. Springer, 2014.
12. J. Brotherston, N. Gorogiannis, M. Kanovich, and R. Rowe. Model checking for symbolic-heap separation logic with inductive predicates. Submitted, 2015.
13. J. Brotherston, N. Gorogiannis, and R. L. Petersen. A generic cyclic theorem prover. In *Proc. APLAS-10*, pages 350–367. Springer, 2012.
14. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6), 2011.
15. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006–1036, 2012.
16. K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In *Proc. CADE-24*, pages 392–406. Springer, 2013.
17. B. Cook, C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR'11*, pages 235–249. Spinger, 2011.
18. C. Hurlin, F. Bobot, and A. J. Summers. Size does matter: Two certified abstractions to disprove entailment in intuitionistic and classical separation logic. In *Proc. IWACO*, pages 5:1–5:6. ACM, 2009.
19. R. Iosif, A. Rogalewicz, and J. Simacek. The tree width of separation logic with recursive definitions. In *Proc. CADE-24*, pages 21–38. Springer, 2013.
20. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *Proceedings of NFM-3*, pages 41–55. Springer, 2011.
21. S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proc. POPL-37*, pages 211–222. ACM, 2010.
22. E. Pek, X. Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in C using separation logic. In *Proc. PLDI-35*, pages 440–451. ACM, 2014.
23. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS-17*, pages 55–74. IEEE Computer Society, 2002.