A Decision Procedure for Satisfiability in Separation Logic with Inductive Predicates

James Brotherston * Carsten Fuhs †
Juan A. Navarro Pérez ‡
University College London
{j.brotherston,c.fuhs,juan.navarro}@ucl.ac.uk

Nikos Gorogiannis
Middlesex University London
nikos.gorogiannis@gmail.com

Abstract

We show that the satisfiability problem for the "symbolic heap" fragment of separation logic with general inductively defined predicates — which includes most fragments employed in program verification — is *decidable*. Our decision procedure is based on the computation of a certain fixed point from the definition of an inductive predicate, called its "base", that exactly characterises its satisfiability.

A complexity analysis of our decision procedure shows that it runs, in the worst case, in exponential time. In fact, we show that the satisfiability problem for our inductive predicates is EXPTIME-complete, and becomes NP-complete when the maximum arity over all predicates is bounded by a constant.

Finally, we provide an implementation of our decision procedure, and analyse its performance both on a synthetically generated set of test formulas, and on a second test set harvested from the separation logic literature. For the large majority of these test cases, our tool reports times in the low milliseconds.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification, assertions; F.2.2 [*Analysis of Algorithms and Problem Complexity*]: Nonnumerical Algorithms and Problems—Complexity of proof procedures

General Terms Algorithms, Theory, Verification

Keywords separation logic, inductive predicates, satisfiability, decision procedure

1. Introduction

Separation logic [26] is an established and fairly popular formalism for verifying imperative, heap-manipulating programs. At the time of writing, there are a number of automatic program verification tools based on separation logic, such as SLAYER [5] and ABDUCTOR [12], capable of establishing memory safety properties of code bases extending into millions of lines [28]. These verification tools are highly dependent on the use of *inductively defined predicates* to describe the shape of data structures held in memory, such as linked lists or trees. Currently, such predicates must be hard-coded into these verification tools, which limits the range of data structures that they can handle automatically. Thus, the next step in automation is to handle *general* inductive predicates, which might be provided to the analysis by the user, or even inferred automatically [10]. When one considers arbitrary inductive predicates, however, it becomes much more difficult to determine whether a given formula is *satisfiable* or not. This may lead to performance degradation when time is spent on the analysis of inconsistent scenarios.

In this paper we address the latter problem by showing that the satisfiability problem for the most commonly considered *symbolic heap* fragment of separation logic, extended with general inductive predicates, is in fact *decidable*. Our decision procedure rests upon the observation that the satisfiability of each inductive predicate can be precisely characterised by an approximation of its set of models, an object which we refer to as the *base* of the predicate. Roughly speaking, the base of a predicate records, for each distinct way of constructing a satisfying model, the subset of arguments of the predicate that are required to be allocated and distinct in memory, as well as the equalities and disequalities that must hold between these arguments. Since there are clearly only finitely many possible such subsets and equality/disequality relations between predicate arguments, the base can be straightforwardly computed in finite time. Having computed the base of all required predicates, deciding the satisfiability of arbitrary formulas in the fragment then becomes a straightforward matter.

A complexity analysis of our decision procedure shows that, in the worst case, it runs in exponential time (in the size of the underlying set of inductive definitions). Essentially, this is because our inductive definition schema allows us to construct predicates that admit an exponential number of elements in their base. Indeed, we show that the satisfiability problem for our inductive predicates is EXPTIME-complete. Additionally, if the maximum arity of all inductive predicates is bounded, then the satisfiability problem becomes NP-complete.

We also provide an implementation of our decision procedure, which is available online [1], and evaluate its performance over three test sets: (a) a collection of example formulas drawn from the literature on separation logic verification; (b) a large set of inductive predicates automatically generated by the CABER predicate inference tool [10]; and (c) a set of synthetically generated examples, varying in parameters such as the number of arguments

* Research supported by an EPSRC Career Acceleration Fellowship.

† Research supported by EPSRC grants EP/K040863/1 and EP/H008373/2.

‡ Research supported by EPSRC grants EP/K040863/1 and EP/K032542/1.

CSL-LICS 2014, July 14–18, 2014, Vienna, Austria.

ACM 978-1-4503-2886-9/14/07.
http://dx.doi.org/10.1145/2603088.2603091

or recursive calls. Although exponential time performance can be produced by a suitable choice of parameters, we find that for the vast majority of examples, the algorithm terminates in a matter of milliseconds. Consequently, we believe our decision procedure is highly suitable for use as a black-box satisfiability checker in automated separation logic verification. We propose a number of direct applications of such a satisfiability checker at the end of this paper (in Section 7).

The problem of satisfiability for separation logic with inductive definitions was also recently considered by Iosif et al. [20]. Their paper establishes decidability results for both satisfiability *and* entailment problems via an embedding into monadic second order logic, but for technical reasons their results are restricted to a fragment of the logic from which many natural data structure definitions are excluded; for example, inductive predicates may not define structures with dangling data pointers (cf. Section 6.1). In contrast, the fragment of separation logic for which we decide satisfiability includes a much larger class of inductive predicates, but we do not consider the entailment problem for this fragment, which is *undecidable* [2]. We also provide complexity results on the satisfiability problem for our fragment, and an implementation of our decision procedure.

The remainder of this paper is structured as follows. We introduce our fragment of separation logic with inductive definitions in Section 2, and then present our decision procedure and the proof of its correctness in Section 3. Section 4 contains our complexity analysis of our algorithm and of the satisfiability problem itself. Section 5 describes the implementation of our decision procedure and its evaluation. Section 6 surveys related work in the area, and Section 7 concludes.

2. Inductive definitions in separation logic

Here we present our fragment of inductive definitions in separation logic, following the approach in [9].

2.1 Syntax

First, some preliminaries. A *term* is either a *variable* drawn from the infinite set Var, or the constant symbol nil. We write Term for the set of all terms. We also assume a fixed finite set $P_1, \ldots, P_n$ of *predicate symbols*, each with an associated arity. We often write vector notation to abbreviate tuples; e.g. we typically write $\mathbf{x}$ rather than $(x_1, \ldots, x_m)$. We write $\pi_i(-)$ for the i-th projection function on tuples, and sometimes abuse notation slightly by writing $x \in \mathbf{x}$ to mean that x occurs in the tuple $\mathbf{x}$. Finally, we write $\mathrm{Pow}(X)$ for the powerset of a set X.

Definition 2.1. *Spatial formulas* F and *pure formulas* G are given by the following grammar:

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

where t ranges over terms, P_i over the predicate symbols and $\mathbf{t}$ over tuples of terms (matching the arity of P_i in $P_i\mathbf{t}$). We write $\mathsf{PureSet}$ for the set of all *finite sets* of pure formulas.

A *symbolic heap* is given by $\Pi : F$, where F is a spatial formula and $\Pi \in \mathsf{PureSet}$ (note that Π should be read intuitively as the conjunction of its elements). Whenever one of Π, F is empty, we will omit the colon.

We write the substitution notation $F[t/x]$ for the result of simultaneously replacing all occurrences of the variable x by the term t in the formula F. Substitution extends to sets of formulas in the obvious way.

Definition 2.2. An *inductive rule set* is a finite set of *inductive rules*, each of the form $\Pi : F \Rightarrow P_i\mathbf{x}$, where $\Pi : F$ is a symbolic heap, P_i is a predicate symbol of arity a_i, and $\mathbf{x}$ is a tuple of a_i distinct variables.

Variables in the body, $\Pi\colon F$, but not the head, $P_i(\mathbf{x})$, of a rule are implicitly existentially quantified, while multiple bodies of the same head are interpreted as a disjunction. The formal semantics is given in the next section.

Our strict formatting of the heads of inductive rules, with variable repetitions and occurrences of nil disallowed, is for technical convenience. It does not restrict expressivity since we can achieve the same effect by placing equalities in the bodies of inductive rules. For example, a rule of the form $\Rightarrow P(x, x, \mathsf{nil})$ is not allowed by our schema, but the equivalent inductive rule $x = y, z = \mathsf{nil} \Rightarrow P(x, y, z)$ is allowed.

2.2 Semantics

We use a typical RAM model employing heaps of records. An infinite set Val of *values* is assumed, of which an infinite subset $\mathsf{Loc} \subset \mathsf{Val}$ are *locations*, i.e., the addresses of heap cells. We also assume a "nullary" value $nil \in \mathsf{Val}\backslash\mathsf{Loc}$ which is not the address of any heap cell. A *stack* is a function $s\colon \mathsf{Var} \to \mathsf{Val}$; we extend stacks to terms by setting $s(\mathsf{nil}) =_{\text{def}} nil$, and extend stacks pointwise to act on tuples of terms. We write $s[x \mapsto v]$ for the stack defined as s except that $(s[x \mapsto v])(x) = v$.

A *heap* is a partial function $h\colon \mathsf{Loc} \rightharpoonup_{\text{fin}} (\mathsf{Val}\ \mathsf{List})$ mapping finitely many locations to (arbitrary-length) tuples of values; we define

$$\mathrm{dom}(h) =_{\text{def}} \{\ell \in \mathsf{Loc} \mid h(\ell) \text{ is defined}\}$$

and e to be the empty heap that is undefined everywhere. We write $\circ$ to denote *composition* of heaps: if h_1 and h_2 are heaps, then $h_1 \circ h_2$ is the union of (partial functions) h_1 and h_2 when $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$, and undefined otherwise. We write $h[\ell \mapsto v]$ for the heap defined as h except that $(h[\ell \mapsto v])(\ell) = v$, and just $[\ell \mapsto v]$ as a shorthand for $e[\ell \mapsto v]$. We write Heap for the set of all heaps.

Given an inductive rule set Φ, the relation $s, h \models_\Phi F$ for satisfaction of a pure or spatial formula F by the stack s and heap h is defined as follows:

$$\begin{array}{lcl}
s,h \models_\Phi t_1 = t_2 & \Leftrightarrow & s(t_1) = s(t_2) \\
s,h \models_\Phi t_1 \neq t_2 & \Leftrightarrow & s(t_1) \neq s(t_2) \\
s,h \models_\Phi \mathsf{emp} & \Leftrightarrow & h = e \\
s,h \models_\Phi t \mapsto \mathbf{t} & \Leftrightarrow & \mathrm{dom}(h) = \{s(t)\} \text{ and } h(s(t)) = s(\mathbf{t}) \\
s,h \models_\Phi P_i\mathbf{t} & \Leftrightarrow & (s(\mathbf{t}), h) \in [\![P_i]\!]^\Phi \\
s,h \models_\Phi F_1 * F_2 & \Leftrightarrow & h = h_1 \circ h_2 \text{ and } s, h_1 \models_\Phi F_1 \\
 & & \quad \text{and } s, h_2 \models_\Phi F_2
\end{array}$$

where the semantics $[\![P_i]\!]^\Phi$ of the inductive predicate P_i under Φ is defined below. If $\Pi : F$ is a symbolic heap then we write $s, h \models_\Phi \Pi : F$ to mean that $s, h \models_\Phi F$ and $s, h \models_\Phi G$ for all $G \in \Pi$; equivalently, we say that (s, h) is a *model* of $\Pi : F$ (w.r.t. Φ). A symbolic heap $\Pi : F$ is *satisfiable* (w.r.t. Φ) if it has at least one model.

We remark that satisfaction of pure formulas does not depend on the heap nor on the inductive rules: we write $s \models \Pi$, where $\Pi \in \mathsf{PureSet}$, to mean that $s, h \models_\Phi \Pi$ for any heap h and inductive definition set Φ.

The following definition gives the standard semantics of the inductive predicate symbols $\mathbf{P}$ according to a fixed inductive rule set Φ, i.e., as the least fixed point of an n-ary monotone operator constructed from Φ:

Definition 2.3. First, for each predicate $P_i \in \mathbf{P}$ with arity a_i say, we define $\tau_i = \mathrm{Pow}(\mathsf{Val}^{a_i} \times \mathsf{Heap})$. Furthermore, we partition

the rule set Φ into $\Phi_1, \ldots, \Phi_n$, where Φ_i is the set of all inductive rules in Φ of the form $\Pi : F \Rightarrow P_i\mathbf{x}$.

We let each Φ_i be indexed by j (i.e., $\Phi_{i,j}$ is the j-th rule defining P_i), and for each inductive rule $\Phi_{i,j}$ of the form $\Pi : F \Rightarrow P_i\mathbf{x}$, we define the operator $\varphi_{i,j} : \tau_1 \times \ldots \times \tau_n \to \tau_i$ by:

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

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

As in [9], Definition 2.3 interprets the inductive rules Φ_i as *exhaustive, disjunctive clauses* of the definition of the predicate P_i.

3. A decision procedure for satisfiability of inductive predicates

In this section, we present our decision procedure for satisfiability in the symbolic heap fragment of separation logic with inductive predicates, given in Section 2.

Our first definition is used to normalise terms in pure formulas with respect to a given set of variables. This is then used in our second definition, which provides a satisfiability-preserving means of restricting the variables which occur in a set of pure formulas.

Definition 3.1. Let $\Pi \in \mathsf{PureSet}$ and let $\mathbf{x}$ be a tuple of distinct variables. The equalities in Π determine an equivalence relation among terms. We say that an equivalence class is *observable (from $\mathbf{x}$ with respect to Π)*, if the class contains some term in $\mathbf{x} \cup \{\mathsf{nil}\}$. A term is observable just in case it belongs to an observable equivalence class.

For each equivalence class, we choose a term from the class, called its *canonical representative*. For observable classes, we choose nil to be the representative if nil is in the class, otherwise we choose a variable from $\mathbf{x}$. For non-observable classes, the choice is arbitary. The *normal form* of a term t, denoted $\langle t \rangle_{\Pi,\mathbf{x}}$, is defined to be t itself when $t \in \mathbf{x} \cup \{\mathsf{nil}\}$, or the canonical representative of its equivalence class otherwise.

Definition 3.2. Let $\Pi \in \mathsf{PureSet}$ and let $\mathbf{x}$ be a tuple of distinct variables. We denote by $\Pi \restriction \mathbf{x}$ the formula obtained after: (i) removing all pure formulas, $t_1 = t_2$ or $t_1 \neq t_2$, where at least one of t_1 or t_2 is not observable; and (ii) replacing all remaining terms t by their normal forms $\langle t \rangle_{\Pi,\mathbf{x}}$.

Note that Definition 3.2 ensures every term occurring in $\Pi \restriction \mathbf{x}$ is either nil, or a variable from $\mathbf{x}$. The satisfiability-preserving nature of this variable restriction is formalised by the following pair of lemmas.

Lemma 3.3. *Let Π be a finite set of pure formulas. If $s \models \Pi$ and $s(\mathbf{x}) = s'(\mathbf{x})$ then $s' \models \Pi \restriction \mathbf{x}$.*

Proof. Since $s \models \Pi$, we know that s must assign the same value to all variables sharing the same equivalence class. Thus $s \models \Pi \restriction \mathbf{x}$, since both dropping formulas from Π and replacing terms by their normal forms preserve its satisfiability.

Now, $\Pi \restriction \mathbf{x}$ is restricted to contain only variables from $\mathbf{x}$ and, since s and s' agree on those variables, $s' \models \Pi \restriction \mathbf{x}$. □

Lemma 3.4. *Let Π be a finite set of pure formulas. If Π is satisfiable and $s \models \Pi \restriction \mathbf{x}$ then there exists a stack s' with $s'(\mathbf{x}) = s(\mathbf{x})$ such that $s' \models \Pi$.*

Furthermore, given any finite set $W \subseteq \mathsf{Loc}$, we can choose s' such that $s'(y) \notin W$ for all non-observable variables y.

define $base^{\Phi}\mathbf{P}$:
  $\mathbf{Y} \coloneqq (\lambda\mathbf{t}_1.\ \emptyset, \ldots, \lambda\mathbf{t}_n.\ \emptyset)$
  repeat until $\mathbf{Y}$ reaches a fixed point:
    pick a rule $\Phi_{i,j} \in \Phi$ with head $P_i\mathbf{x}$
    for each $P_{j_\ell}(\mathbf{x}_\ell)$ in the body of $\Phi_{i,j}$:
      pick a base pair $(V_\ell, \Pi_\ell) \in Y_{j_\ell}(\mathbf{x}_\ell)$
    take $y_1, \ldots, y_k$ from all $y_\ell \mapsto \mathbf{u}_\ell$ in the body of $\Phi_{i,j}$
    take Π_0 the pure part in the body of $\Phi_{i,j}$
    $V \coloneqq V_1 \cup \cdots \cup V_m \cup \{y_1, \ldots, y_k\}$
    $\Pi \coloneqq \bigotimes V \cup \Pi_0 \cup \Pi_1 \cup \cdots \cup \Pi_m$
    if Π is satisfiable:
      add $\big(alloced(V, \mathbf{x}, \Pi)[\mathbf{t}/\mathbf{x}], (\Pi \restriction \mathbf{x})[\mathbf{t}/\mathbf{x}]\big)$ to $Y_i(\mathbf{t})$
  return $\mathbf{Y}$

Figure 1. Pseudocode for the computation of $base^{\Phi}\mathbf{P}$ in Defn. 3.5.

Proof. Let $Y_1, \ldots, Y_n$ be the equivalence classes of all variables occurring in, and determined by, Π. Choose distinct values $\ell_1, \ldots, \ell_n$ in Loc such that each $\ell_i \notin s(\mathbf{x} \cup \{\mathsf{nil}\}) \cup W$. This is possible because the number of locations in Loc is infinite, while the number of forbidden locations is finite. We then define

$$s'(x) = \begin{cases} s(\langle x \rangle_{\Pi,\mathbf{x}}) & \text{if } x \text{ is observable} \\ \ell_i & \text{otherwise, where } x \in Y_i\ . \end{cases}$$

By construction s and s' agree on variables from $\mathbf{x}$, i.e. directly observable ones, and $s'(y) \notin W$ when y is not observable. It is easy to check now that s' satisfies all pure formulas in Π.

- For every $t_1 = t_2 \in \Pi$: By definition, t_1 and t_2 are in the same equivalence class. Thus if t_1 is observable then so, necessarily, is t_2, and furthermore the formula $\langle t_1 \rangle_{\Pi,\mathbf{x}} = \langle t_2 \rangle_{\Pi,\mathbf{x}}$ is in $\Pi \restriction \mathbf{x}$. Therefore, by assumption, $s \models \langle t_1 \rangle_{\Pi,\mathbf{x}} = \langle t_2 \rangle_{\Pi,\mathbf{x}}$, and since $s'(t_i) = s(\langle t_i \rangle_{\Pi,\mathbf{x}})$, we have $s' \models t_1 = t_2$.
  Otherwise, if t_1 is not observable, then neither is t_2; but, since they share an equivalence class, we have $s'(t_1) = \ell_i = s'(t_2)$ for some i by construction, and so $s' \models t_1 = t_2$ as required.
- For every $t_1 \neq t_2 \in \Pi$: If both t_1 and t_2 are observable, then $\langle t_1 \rangle_{\Pi,\mathbf{x}} \neq \langle t_2 \rangle_{\Pi,\mathbf{x}}$ is in $\Pi \restriction \mathbf{x}$, so $s \models \langle t_1 \rangle_{\Pi,\mathbf{x}} \neq \langle t_2 \rangle_{\Pi,\mathbf{x}}$ by assumption. Since $s'(t_i) = s(\langle t_i \rangle_{\Pi,\mathbf{x}})$, it follows that $s' \models t_1 \neq t_2$.
  If one of the terms, say t_1, is observable but the other, t_2, is not, then $s'(t_2)$ was explicitly chosen to be different to $s(\langle t_1 \rangle_{\Pi,\mathbf{x}}) = s'(t_1)$, and so $s' \models t_1 \neq t_2$.
  If neither t_1 nor t_2 is observable, then they must be in different equivalence classes, otherwise Π would be unsatisfiable, contrary to assumption. Thus, by construction, $s' \models t_1 \neq t_2$. □

We now present our main definition, which explains how to extract from a fixed inductive rule set Φ all the information needed to decide the satisfiability of any symbolic heap over Φ. The pseudocode in Figure 1 provides an informal aid to navigate the steps of the computation.

Definition 3.5 (Base pair computation)**.** First, for each predicate symbol P_i with associated arity a_i, where $1 \leq i \leq n$, we define

$$\sigma_i =_{\text{def}} \mathsf{Term}^{a_i} \to \mathrm{Pow}(\mathrm{MPow}(\mathsf{Var}) \times \mathsf{PureSet})$$

where $\mathrm{MPow}(\mathsf{Var})$ denotes the set of all multisets of variables. For any finite $V \in \mathrm{MPow}(\mathsf{Var})$ we define $\bigotimes V \in \mathsf{PureSet}$ to be the set of pure formulas comprising:

- all formulas of the form $x \neq x'$ such that x and x' are different elements of V (note this means that if V contains duplicates then $\bigotimes V$ is unsatisfiable);
- the formula $x \neq \mathsf{nil}$ for every element x of V.

For any finite multiset V of variables, any tuple of variables $\mathbf{x}$, and any $\Pi \in \mathsf{PureSet}$, let $\mathit{alloced}(V, \mathbf{x}, \Pi)$ denote the multiset containing the term $\langle y \rangle_{\Pi,\mathbf{x}}$ for each $y \in V$ observable from $\mathbf{x}$ with respect to Π. Note that if $\Pi \cup \bigotimes V$ is satisfiable, then $\mathit{alloced}(V, \mathbf{x}, \Pi)$ contains only variables (no nil) and no duplicates. In this way we represent, using only variables from $\mathbf{x}$, the observable locations allocated via V.

The inductive rule set Φ is partitioned into $\Phi_1, \ldots, \Phi_n$ with each Φ_i further indexed by j as in Definition 2.3. Without loss of generality, we consider each $\Phi_{i,j} \in \Phi$ to be written in the form

$$\Pi_0 : y_1 \mapsto \mathbf{u_1} * \ldots * y_k \mapsto \mathbf{u_k} * P_{j_1}(\mathbf{x_1}) * \ldots * P_{j_m}(\mathbf{x_m}) \Rightarrow P_i\mathbf{x}, \quad \text{(IndRule)}$$

where $\Pi_0 \in \mathsf{PureSet}$. We use the inductive rule $\Phi_{i,j}$ to define an operator $\Psi_{i,j} \colon \sigma_1 \times \ldots \times \sigma_n \to \sigma_i$ as follows. If $\mathbf{Y} = (Y_1, \ldots, Y_n)$ where each $Y_i \in \sigma_i$, and $\mathbf{t}$ is a tuple of a_i terms, then $\Psi_{i,j}(\mathbf{Y}) \colon \sigma_i$ sends $\mathbf{t}$ to the set of pairs (we shall call these pairs also *base pairs*)

$$\big(\mathit{alloced}(V, \mathbf{x}, \Pi)[\mathbf{t}/\mathbf{x}], (\Pi \upharpoonright \mathbf{x})[\mathbf{t}/\mathbf{x}]\big)$$

that satisfy the following:

$$\begin{aligned} &V = V_1 \cup \cdots \cup V_m \cup \{y_1, \ldots, y_k\}, \\ &\Pi = \textstyle\bigotimes V \cup \Pi_0 \cup \Pi_1 \cup \cdots \cup \Pi_m \text{ is satisfiable,} \\ &\text{and } \forall 1 \leq \ell \leq m.\ (V_\ell, \Pi_\ell) \in Y_{j_\ell}(\mathbf{x}_\ell). \end{aligned}$$

Note that the substitution $[\mathbf{t}/\mathbf{x}]$ in the above is defined pointwise over tuples; this is well defined since $\mathbf{x}$ is a tuple of *distinct* variables, as per Definition 2.2. Note also that while $\mathit{alloced}(V, \mathbf{x}, \Pi)$ might be viewed as a set (i.e., without repetitions), it is important to understand $\mathit{alloced}(V, \mathbf{x}, \Pi)[\mathbf{t}/\mathbf{x}]$ as a multiset since the substitution may map different variables to the same term.

We then define $\mathit{base}^\Phi\mathbf{P} \in \sigma_1 \times \ldots \times \sigma_n$ as follows:

$$\mathit{base}^\Phi\mathbf{P} =_{\text{def}} \mu\mathbf{Y}.\ (\textstyle\bigcup_j \Psi_{1,j}(\mathbf{Y}), \ldots, \bigcup_j \Psi_{n,j}(\mathbf{Y}))$$

where, by pointwise extension of the union to act on functions, we write $\bigcup_j \Psi_{i,j}(\mathbf{Y})$ to denote the function mapping a tuple of terms $\mathbf{t}$ to the set $\bigcup_j \Psi_{i,j}(\mathbf{Y})(\mathbf{t})$. We also write $\mathit{base}^\Phi P_i$ to abbreviate $\pi_i(\mathit{base}^\Phi\mathbf{P})$.

Example 3.6. Consider the following set Φ of inductive rules:

$$\begin{aligned} &\Phi_{1,1}\colon && x = \mathsf{nil} \Rightarrow P(x) \\ &\Phi_{1,2}\colon && x \neq \mathsf{nil} : Q(x,x) \Rightarrow P(x) \\ &\Phi_{2,1}\colon && y = \mathsf{nil}, x \neq \mathsf{nil} : x \mapsto (d,c) * P(d) \Rightarrow Q(x,y) \\ &\Phi_{2,2}\colon && y \neq \mathsf{nil} : y \mapsto (d,c) * Q(x,c) \Rightarrow Q(x,y). \end{aligned}$$

These rules are an intermediate product of the analysis done by the tool CABER [10] on code that traverses a list of lists, and are found to be ultimately unsatisfiable by our algorithm, triggering backtracking in CABER. In accordance with Definition 3.5, we define:

$$\begin{aligned} \sigma_1 &= \mathsf{Term} \to \mathrm{Pow}(\mathrm{MPow}(\mathsf{Var}) \times \mathsf{PureSet}) \\ \sigma_2 &= \mathsf{Term} \times \mathsf{Term} \to \mathrm{Pow}(\mathrm{MPow}(\mathsf{Var}) \times \mathsf{PureSet})\ . \end{aligned}$$

That is, σ_1 and σ_2 are function spaces mapping suitably many terms to sets of base pairs for P and Q, respectively. We can compute $\mathit{base}^\Phi(P, Q)$ by iteratively computing the sequence of its fixed point approximants $((Y_1^i, Y_2^i) \in \sigma_1 \times \sigma_2)_{i \geq 0}$ in the standard way. That is, we begin with an empty base for both predicates, namely $\mathbf{Y}^0 = (\lambda x.\ \emptyset, \lambda x, y.\ \emptyset)$, and iteratively apply the operators $\Psi_{i,j}$ given by Definition 3.5.

First iteration: The inductive rule $\Phi_{1,1}$ for $P(x)$ can be applied to $\mathbf{Y}^0$, yielding $V = \emptyset$ and $\Pi = \{x = \mathsf{nil}\}$. The set Π is satisfiable and, therefore,

$$\Psi_{1,1}(Y_1^0, Y_2^0) = \lambda x.\ \big\{(\emptyset, \{x = \mathsf{nil}\})\big\}\ .$$

Since $\mathbf{Y}^0$ does not contain any base pairs, the remaining rules cannot be applied to it, and so $\mathbf{Y}^1 = (Y_1^1, Y_2^1)$ where

$$\begin{aligned} Y_1^1 &= \lambda x.\ \big\{(\emptyset, \{x = \mathsf{nil}\})\big\} \\ Y_2^1 &= \lambda x, y.\ \emptyset\ . \end{aligned}$$

Second iteration: The inductive rule $\Phi_{1,1}$ generates the same base pair as before, while $\Phi_{1,2}$ and $\Phi_{2,2}$ are still not applicable to $\mathbf{Y}^1$. However, the rule $\Phi_{2,1}$ defining $Q(x, y)$ can be now applied, taking $(\emptyset, \{x = \mathsf{nil}\})$ from the current base Y_1^1 of $P(x)$. After restricting and normalising to the set of observable variables, this produces the new base pair $(\{x\}, \{y = \mathsf{nil}, x \neq \mathsf{nil}\})$. Thus, at the end of the iteration, $\mathbf{Y}^2 = (Y_1^2, Y_2^2)$ where

$$\begin{aligned} Y_1^2 &= \lambda x.\ \big\{(\emptyset, \{x = \mathsf{nil}\})\big\} \\ Y_2^2 &= \lambda x, y.\ \big\{(\{x\}, \{y = \mathsf{nil}, x \neq \mathsf{nil}\})\big\}\ . \end{aligned}$$

Third iteration: The rule $\Phi_{1,2}$ is now applicable to $\mathbf{Y}^2$, instantiating the current pair for $Q(x, x)$, but yields an unsatisfiable Π. Similarly, the rule $\Phi_{2,2}$ can be applied, taking the base pair $(\{x\}, \{c = \mathsf{nil}, x \neq \mathsf{nil}\})$ from Y_2^2 instantiated to $Q(x, c)$. After a suitable variable restriction and normalisation, this produces the following new base pair for $Q(x, y)$:

$$(\{x, y\}, \{x \neq y, y \neq \mathsf{nil}, x \neq \mathsf{nil}\})$$

Thus $\mathbf{Y}^3 = (Y_1^3, Y_2^3)$ where

$$\begin{aligned} Y_1^3 &= \lambda x.\ \big\{(\emptyset, \{x = \mathsf{nil}\})\big\} \\ Y_2^3 &= \lambda x, y.\ \left\{ \begin{array}{c} (\{x\}, \{y = \mathsf{nil}, x \neq \mathsf{nil}\}), \\ (\{x, y\}, \{x \neq y, y \neq \mathsf{nil}, x \neq \mathsf{nil}\}) \end{array} \right\}. \end{aligned}$$

On the fourth iteration no new base pairs are produced — both $\Phi_{1,2}$ and $\Phi_{2,2}$ can be applied to the newly created base pair from the latest iteration, but the former yields an unsatisfiable Π and the latter reproduces the same base pair — so we have reached a fixed point, and $\mathit{base}^\Phi(P, Q) = (Y_1^3, Y_2^3)$.

The next two lemmas formalise the fact that $\mathit{base}^\Phi P(\mathbf{x})$ precisely characterises the satisfiability of $P(\mathbf{x})$.

Lemma 3.7 (Soundness)**.** *Given a base pair* $(V, \Pi) \in (\mathit{base}^\Phi P_i)(\mathbf{t})$*, a stack* s *such that* $s \models \Pi$*, and a finite set* $W \subseteq \mathsf{Loc} \setminus s(V)$*, there exists a heap* h *such that* $s, h \models_\Phi P_i\mathbf{t}$ *and* $\mathrm{dom}(h) \cap W = \emptyset$*.*

Proof. We proceed by fixed point induction on the definition of $\mathit{base}^\Phi\mathbf{P}$. That is, we assume the lemma holds for a tuple of functions $\mathbf{Y} = (Y_1, \ldots, Y_n) \in \sigma_1 \times \cdots \times \sigma_n$, and we must show it also holds for $(\bigcup_j \Psi_{1,j}(\mathbf{Y}), \ldots, \bigcup_j \Psi_{n,j}(\mathbf{Y}))$. Thus, assume $(V, \Pi) \in \Psi_{i,j}(\mathbf{Y})(\mathbf{t})$ with $s \models \Pi$ and $s(V) \cap W = \emptyset$. We require to find an h with $s, h \models P_i\mathbf{t}$ and $\mathrm{dom}(h) \cap W = \emptyset$.

By assumption, there is a rule $\Phi_{i,j}$ of the form (IndRule) with $(V, \Pi) \in \Psi_{i,j}(\mathbf{Y})(\mathbf{t})$. Therefore $V = \mathit{alloced}(V', \mathbf{x}, \Pi')[\mathbf{t}/\mathbf{x}]$ and $\Pi = (\Pi' \upharpoonright \mathbf{x})[\mathbf{t}/\mathbf{x}]$, where the following hold:

$$\begin{aligned} &V' = V_1 \cup \cdots \cup V_m \cup \{y_1, \ldots, y_k\}, \\ &\Pi' = \textstyle\bigotimes V' \cup \Pi_0 \cup \Pi_1 \cup \cdots \cup \Pi_m \text{ is satisfiable,} \\ &\forall 1 \leq \ell \leq m.\ (V_\ell, \Pi_\ell) \in Y_{j_\ell}(\mathbf{x}_\ell). \end{aligned}$$

By the lemma assumption and usual substitution facts, we have that $s[\mathbf{x} \mapsto s(\mathbf{t})] \models \Pi' \upharpoonright \mathbf{x}$ and

$$s[\mathbf{x} \mapsto s(\mathbf{t})](\mathit{alloced}(V', \mathbf{x}, \Pi')) \cap W = \emptyset.$$

Since Π' is satisfiable, we can apply Lemma 3.4 to obtain s' with $s'(\mathbf{x}) = s[\mathbf{x} \mapsto s(\mathbf{t})](\mathbf{x}) = s(\mathbf{t})$ and $s' \models \Pi'$. Furthermore, the lemma tells us that for any non-observable $y \in V'$, we have $y \notin W$. Otherwise, if $y \in V'$ is observable from $\mathbf{x}$ w.r.t. Π', there is an $x \in \mathbf{x}$ such that y and x are in the same Π'-equivalence class. It follows that $s'(y) = s'(x) = s[\mathbf{x} \mapsto s(\mathbf{t})](x)$ and, as $x \in \mathit{alloced}(V', \mathbf{x}, \Pi')$ but $s[\mathbf{x} \mapsto s(\mathbf{t})](\mathit{alloced}(V', \mathbf{x}, \Pi')) \cap W = \emptyset$, it must be the case that $s'(y) \notin W$. Thus, for all $y \in V'$ we have $s'(y) \notin W$, that is $s'(V') \cap W = \emptyset$.

We now show that there are heaps $h_1, \ldots, h_m$ such that, for all $1 \leq \ell \leq m$, we have both $s', h_\ell \models_\Phi P_{j_\ell}\mathbf{x}_\ell$ and $\mathrm{dom}(h_\ell) \cap W_\ell = \emptyset$, where W_ℓ is defined as follows:

$$W_\ell =_{\text{def}} W \cup s'(\{y_1, \ldots, y_k\}) \cup \bigcup_{p<\ell} \mathrm{dom}(h_p) \cup \bigcup_{\ell<q\leq m} s'(V_q) \ .$$

To do this, we inductively assume that we have constructed the chain of heaps $(h_p)_{1\leq p<\ell}$, and show how to construct h_ℓ.

Inductive construction of h_ℓ from $(h_p)_{1\leq p<\ell}$: We claim and prove that $s'(V_\ell) \cap W_\ell = \emptyset$. We have shown that $s'(V') \cap W = \emptyset$; since $V_\ell \subseteq V'$, it follows that $s'(V_\ell) \cap W = \emptyset$. Next, let $p < \ell$ and note that the induction hypothesis implies $\mathrm{dom}(h_p) \cap W_p = \emptyset$, which in turn implies $\mathrm{dom}(h_p) \cap s'(V_\ell) = \emptyset$ by definition of W_p. Thus $s'(V_\ell) \cap \bigcup_{p<\ell} \mathrm{dom}(h_p) = \emptyset$. Next, let $q > \ell$ and notice that $s'(V_\ell) \cap s'(V_q) = \emptyset$ because $s' \models \bigotimes V'$, which guarantees that s' is injective on $V' \supseteq V_\ell, V_q$. It therefore follows that $s'(V_\ell) \cap \bigcup_{\ell<q\leq m} s'(V_q) = \emptyset$. Also, for a similar reason, $s'(V_\ell) \cap s'(\{y_1, \ldots, y_k\}) = \emptyset$.

Now $(V_\ell, \Pi_\ell) \in Y_{j_\ell}(\mathbf{x}_\ell)$, where $s' \models \Pi_\ell$ and $s'(V_\ell) \cap W_\ell = \emptyset$, so by the main induction hypothesis (of the fixed point induction for the present lemma) there is a heap h_ℓ such that $s', h_\ell \models_\Phi P_{j_\ell}\mathbf{x}_\ell$ and $\mathrm{dom}(h_\ell) \cap W_\ell = \emptyset$. This completes the construction of h_ℓ.

We continue with the main proof. Note that $h_1 \circ \ldots \circ h_m$ is defined because $\mathrm{dom}(h_\ell) \cap W_\ell = \emptyset$ for all $1 \leq \ell \leq m$ implies that $\mathrm{dom}(h_1), \ldots, \mathrm{dom}(h_m)$ are all disjoint from each other. Now we define a heap h' whose domain is $s'(\{y_1, \ldots, y_k\})$ as follows: $h'(s'(y_i)) =_{\text{def}} s'(\mathbf{u_i})$ for each $1 \leq i \leq k$. Note that $h' \circ (h_1 \circ \ldots \circ h_m)$ is defined because $s' \models \bigotimes V'$ ensures that s' is injective on $\{y_1, \ldots, y_k\} \subseteq V'$ and $\mathrm{dom}(h_\ell) \cap W_\ell = \emptyset$ ensures that $\mathrm{dom}(h_\ell) \cap \mathrm{dom}(h') = \emptyset$ for each $1 \leq \ell \leq m$. Thus, defining $h =_{\text{def}} h' \circ h_1 \circ \ldots \circ h_m$, we obtain:

$$s', h \models_\Phi \Pi_0 : y_1 \mapsto \mathbf{u_1} * \ldots * y_k \mapsto \mathbf{u_k} * P_{j_1}\mathbf{x_1} * \ldots * P_{j_m}\mathbf{x_m}$$

which implies that $s', h \models_\Phi P_i\mathbf{x}$ (by applying the operator $\varphi_{i,j}$). Thus, since $s'(\mathbf{x}) = s(\mathbf{t})$, we obtain $s, h \models_\Phi P_i\mathbf{t}$.

We have $\mathrm{dom}(h) \cap W = \emptyset$ as required because

$$\mathrm{dom}(h_\ell) \cap W \subseteq \mathrm{dom}(h_\ell) \cap W_\ell = \emptyset,$$

for each $1 \leq \ell \leq m$, and $\mathrm{dom}(h') = s'(\{y_1, \ldots, y_k\}) \subseteq s'(V')$ while $s'(V') \cap W = \emptyset$. □

Lemma 3.8 (Completeness)**.** *If $s, h \models_\Phi P_i\mathbf{t}$, there is a base pair $(V, \Pi) \in (\mathit{base}^\Phi P_i)(\mathbf{t})$ such that $s(V) \subseteq \mathrm{dom}(h)$ and $s \models \Pi$.*

Proof. We have that $(s(\mathbf{t}), h) \in [\![P_i]\!]^\Phi$, and apply fixed point induction on the definition of $[\![\mathbf{P}]\!]^\Phi$. That is, we assume the lemma holds for $\mathbf{X} = (X_1, \ldots, X_n) \in \tau_1 \times \ldots \times \tau_n$ and must show that it holds for $(\bigcup_j \varphi_{1,j}(\mathbf{X}), \ldots, \bigcup_j \varphi_{n,j}(\mathbf{X}))$. Thus, assuming that $(s(\mathbf{t}), h) \in \varphi_{i,j}(\mathbf{X})$, we must then find a pair $(V, \Pi) \in (\mathit{base}^\Phi P_i)(\mathbf{t})$ with $s(V) \subseteq \mathrm{dom}(h)$ and $s \models \Pi$.

By assumption, there is an inductive rule $\Phi_{i,j}$ of the form (IndRule) such that $(s(\mathbf{t}), h) \in \varphi_{i,j}(\mathbf{X})$. By construction this means that $s(\mathbf{t}) = s'(\mathbf{x})$ for some stack s', and we have

$$s', h \models_{\mathbf{X}} \Pi_0 : y_1 \mapsto \mathbf{u_1} * \ldots * y_k \mapsto \mathbf{u_k} * P_{j_1}\mathbf{x_1} * \ldots P_{j_m}\mathbf{x_m}$$

Thus $s' \models \Pi_0$ and $h = h_1 \circ \ldots \circ h_k \circ h'_1 \circ \ldots \circ h'_m$, where $s', h_\ell \models_\Phi y_\ell \mapsto \mathbf{u}_\ell$ for all $1 \leq \ell \leq k$, and $(s'(\mathbf{x}_\ell), h'_\ell) \in X_{j_\ell}$ for all $1 \leq \ell \leq m$. Then, by the induction hypothesis, for all $1 \leq \ell \leq m$ there are pairs $(V_\ell, \Pi_\ell) \in (\mathit{base}^\Phi P_{j_\ell})(\mathbf{x}_\ell)$, such that $s'(V_\ell) \subseteq \mathrm{dom}(h'_\ell)$ and $s' \models \Pi_\ell$. Now we define

$$V =_{\text{def}} \bigcup_{1\leq\ell\leq m} V_\ell \cup \{y_1, \ldots, y_k\} \ .$$

As $\mathrm{dom}(h_1), \ldots, \mathrm{dom}(h_k), \mathrm{dom}(h'_1), \ldots, \mathrm{dom}(h'_m)$ are all disjoint, where $\mathrm{dom}(h_\ell) = \{s'(y_\ell)\}$ for each $1 \leq \ell \leq k$ and $\mathrm{dom}(h'_\ell) \supseteq s'(V_\ell)$ for each $1 \leq \ell \leq m$, we have $s' \models \bigotimes V$ (i.e., h would be undefined if s' were not injective on V). Putting everything together, we have

$$s' \models \Pi_0 \cup \bigotimes V \cup \Pi_1 \cup \cdots \cup \Pi_m \text{ and } s'(V) \subseteq \mathrm{dom}(h) \ .$$

Thus $\Pi = \Pi_0 \cup \bigotimes V \cup \Pi_1 \cup \cdots \cup \Pi_m$ is satisfiable, and so

$$\big((V \cap \mathbf{x})[\mathbf{t}/\mathbf{x}], (\Pi \upharpoonright \mathbf{x})[\mathbf{t}/\mathbf{x}]\big) \in (\mathit{base}^\Phi P_i)(\mathbf{t})$$

where the substitution $[\mathbf{t}/\mathbf{x}]$ is well defined as $\mathbf{x}$ is a tuple of distinct variables. From the fact that $s'(\mathbf{x}) = s(\mathbf{t})$, it also follows that $s'(\mathbf{x}) = s[\mathbf{x} \mapsto s(\mathbf{t})](\mathbf{x})$. By Lemma 3.3, this gives us

$$s[\mathbf{x} \mapsto s(\mathbf{t})] \models \Pi \upharpoonright \mathbf{x} \text{ and } s[\mathbf{x} \mapsto s(\mathbf{t})](V \cap \mathbf{x}) \subseteq \mathrm{dom}(h) \ .$$

Thus, from usual facts about substitution, we obtain

$$s \models (\Pi \upharpoonright \mathbf{x})[\mathbf{t}/\mathbf{x}] \text{ and } s(V \cap \mathbf{x})[\mathbf{t}/\mathbf{x}] \subseteq \mathrm{dom}(h) \ .$$

Thus there exists $(V, \Pi) \in (\mathit{base}^\Phi P_i)(\mathbf{t})$ such that $s \models \Pi$ and $s(V) \subseteq \mathrm{dom}(h)$ as required. □

We are now positioned to state our main decidability results.

Theorem 3.9. *A formula $P_i\mathbf{t}$ is satisfiable w.r.t. an inductive rule set Φ iff there is a base pair $(V, \Pi) \in (\mathit{base}^\Phi P_i)(\mathbf{t})$ such that Π is satisfiable.*

Furthermore, for any given Φ, the fixed point computation of $\mathit{base}^\Phi\mathbf{P}$ terminates after a finite number of steps.

Thus it is decidable whether a formula of the form $P_i\mathbf{t}$ is satisfiable w.r.t. Φ.

Proof. Regarding correctness, the "if" direction is immediate from Lemma 3.7 by taking $W = \emptyset$; the "only if" direction follows from Lemma 3.8.

For termination, each predicate P_i has a finite arity a_i and, since each of the pairs $(V, \Pi) \in \mathit{base}^\Phi P_i(\mathbf{t})$ may only use terms from the finitely many in $\mathbf{t}$, the final total number of pairs in $\mathit{base}^\Phi P_i(\mathbf{t})$ is bounded by a finite constant. All of the functions in the tuple $\mathit{base}^\Phi\mathbf{P}$, mapping terms to base pairs, are thus finitely described in a straightforward way. □

Corollary 3.10. *It is decidable whether an arbitrary symbolic heap is satisfiable w.r.t. an inductive rule set Φ.*

Proof. Let $\Pi : F$ be a symbolic heap, and let Ψ be the inductive rule set $\Phi \cup \{\Pi : F \Rightarrow Q\}$, where Q is a new 0-ary predicate not occurring in Φ or $\Pi : F$. Clearly the symbolic heap $\Pi : F$ is satisfiable w.r.t. Φ if and only if Q is satisfiable w.r.t. Ψ, which is a decidable problem according to Theorem 3.9. □

4. Complexity

In this section, we investigate the complexity of the satisfiability problem for inductive predicates in separation logic, which is addressed by the decision procedure in the previous section.

Definition 4.1. We define the decision problem PREDSAT as having instances (Φ, P) where Φ is an inductive rule set and P is a predicate defined in Φ. The pair (Φ, P) is a yes-instance of PREDSAT if $P\mathbf{x}$ is satisfiable w.r.t. Φ, where $\mathbf{x}$ is an arbitrary tuple of distinct variables.

For any $k \in \mathbb{N}$, define k-PREDSAT to be PREDSAT restricted to the situation where all predicates in Φ have arity $\leq k$.

In the following, the length of the encoding of a mathematical object o (e.g., set, pair, formula, etc.) under some reasonable (non-unary) encoding scheme will be denoted by $\|o\|$. Clearly, $\|(\Phi, P)\| = O(\|\Phi\|)$, as P can be represented by a pointer of $\lceil \log_2 |\Phi| \rceil$ bits. Finally we fix $\mathcal{B} =_{\text{def}} \{\top, \bot\}$.

Let α be the maximum arity of any predicates in Φ plus one; clearly, $\alpha = O(\|\Phi\|)$ as the maximum arity must be of the order of the length of the entire object. The length of a base pair is bounded by $\alpha + 2\alpha^2 = O(\|\Phi\|^2)$ (size of the variable set plus the size of the longest pure formula), and the number of distinct base pairs for any predicate in Φ is bounded by

$$N =_{\text{def}} 2^{\alpha + 2\alpha^2}.$$

Finally L is the maximum number of predicate occurrences in any rule; as such, $L = O(\|\Phi\|)$.

Lemma 4.2. *Let $\Phi_{i,j}$ be a rule in the form of (IndRule) and let $T = \langle B_1, \ldots, B_m \rangle$ be a tuple of base pairs for the formulas $P_{j_1}(\mathbf{x_1}), \ldots, P_{j_m}(\mathbf{x_m})$ respectively. Computing whether a base pair B' can be generated from T and $\Phi_{i,j}$ via Definition 3.5 takes time polynomial in $\|\Phi_{i,j}\|$ and $\|T\|$.*

Proof. Immediate from Definition 3.5 and the fact that satisfiability of conjunctions of pure formulas is in P (see, e.g., [3]). □

Definition 4.3. Let Φ be a set of inductive rules. A *rule instance* $r = (B, i, j, r_1, \ldots, r_m)$, for $i, j, m \geq 0$ is such that: (a) the rule $\Phi_{i,j}$ has m predicates in its body; (b) B is a base pair for P_i; (c) $r_1, \ldots, r_m$ are rule instances; and (d) B can be produced by running the body of the loop in Fig. 1 using $\Phi_{i,j}$ and the base pairs associated with $r_1, \ldots, r_m$. We say that the rule instance r defined as above *supports* predicate P_i.

In effect, a rule instance is a base pair together with the witnessing information for its production via our algorithm.

Lemma 4.4. *A predicate P defined in Φ is satisfiable iff there is a rule instance that supports P.*

Proof sketch. It is simple to modify the algorithm in Definition 3.5 so that instead of base pairs, it generates rule instances. Note that the rule instances within a rule instance r must be generated prior to r and thus can be represented by pointers, therefore not altering the space complexity of the algorithm. □

Lemma 4.5. *k-PREDSAT is in NP, for any $k \in \mathbb{N}$.*

Proof. We define a non-deterministic algorithm as follows. Let (Φ, P) be the input instance. As stated in the beginning of this section, there can be up to N distinct base pairs for each predicate in Φ. Thus, for each predicate P_i in Φ we guess a set of up to N entries of the form $\langle B, i, j, \overrightarrow{E_1}, \ldots, \overrightarrow{E_m} \rangle$, where B is a base pair, $i, j, m \geq 0$, and $\overrightarrow{E_\ell}$ is a pointer to an entry for $1 \leq \ell \leq m$. Each entry requires $O(\|\Phi\|^3)$ time to generate: B takes $O(\alpha + 2\alpha^2) = O(\|\Phi\|^2)$ time; pointers i, j take $2\lceil \log_2 |\Phi| \rceil = O(\|\Phi\|)$ time; pointers $\overrightarrow{E_1}, \ldots, \overrightarrow{E_m}$ take $m\lceil \log_2 N \rceil = O(L(\alpha + 2\alpha^2)) = O(\|\Phi\|^3)$ time. By assumption, the arity of all predicates in Φ is bounded by a constant k, thus N is also constant. This means the total number of guesses is polynomial in the input size.

We now check that this set of entries represents a rule instance supporting P. To do this we check: (a) that each entry is well-formed, i.e., that rule $\Phi_{i,j}$ involves m predicates, and that each $\overrightarrow{E_\ell}$ points to an entry for predicate P_{j_ℓ}; and (b) that each entry E can be generated through our algorithm using entries $E_1, \ldots, E_m$ and rule $\Phi_{i,j}$. Then, we check that the set of entries represents a directed acyclic graph when we view the pointers E_ℓ as edges. This will guarantee that there are no cycles and that, therefore, the set of entries represents a set of rule instances. These operations take polynomial time in N and $|\Phi|$ by Lemma 4.2 and standard facts from graph theory. Finally, we check for an entry (rule instance) supporting P.

Since we can check the guessed entries in polynomial time, we can decide satisfiability in non-deterministic polynomial time. □

Lemma 4.6. PREDSAT *is in* EXPTIME.

Proof. Each step of the algorithm in Definition 3.5 picks a rule from Φ and looks for a combination of base pairs that yields a new base pair. Thus it may go through up to N^L combinations of base pairs. In the worst case all rules have to be scanned, checking up to $|\Phi|N^L$ combinations. If no new base pair can be generated then the algorithm has reached a fixed point.

As argued above, there can be at most $|\Phi|N$ distinct base pairs. In the worst case one pair is generated in each step. Thus $|\Phi|N$ steps are required with total time $|\Phi|N|\Phi|N^L p(\|\Phi\|) = O(2^{\text{poly}(\|\Phi\|)})$ where $p(\|\Phi\|)$ is the time taken to check a single combination of base pairs according to Lemma 4.2. □

Our lower bound results use standard facts about boolean circuits, with and without inputs. We make a few common simplifying assumptions: (a) inputs and constants are gates with no inputs and one output; (b) every circuit includes exactly two constants, $\top$ and $\bot$; (c) there is one more kind of gate, the NAND gate (denoted by $\uparrow$); (d) gates are ordered so that if gate i has inputs l, r, then $i > l$ and $i > r$; (e) inputs, $\top$, $\bot$ and NAND gates appear in this order; (f) the output of the maximal gate is the output of the circuit. We first define a mapping from circuits to inductive rule sets.

Definition 4.7. Let C be a boolean circuit with n gates and k inputs. Thus, the constant gates are $k+1$ and $k+2$ ($\top$ and $\bot$ respectively). Define a set of rules Ψ as follows.

$$\Psi =_{\text{def}} \left\{ \begin{array}{r} x \neq \mathsf{nil} \Rightarrow T(x) \\ x = \mathsf{nil} \Rightarrow F(x) \\ F(x) * T(z) \Rightarrow N(x, y, z) \\ F(y) * T(z) \Rightarrow N(x, y, z) \\ T(x) * T(y) * F(z) \Rightarrow N(x, y, z) \end{array} \right\}$$

Also define the set of rules Φ_C as follows.

$$\Phi_C =_{\text{def}} \left\{ \begin{array}{r} T(x_{k+1}) * F(x_{k+2}) * \\ \quad *_{i=k+3}^{n} N(x_{l_i}, x_{r_i}, x_i) \Rightarrow P(x_n) \\ P(x_n) * T(x_n) \Rightarrow Q_\top \\ P(x_n) * F(x_n) \Rightarrow Q_\bot \end{array} \right\} \cup \Psi$$

Above, l_i (resp. r_i) denotes the left (right) input of gate i. Clearly, $\|\Phi_C\| = O(\|C\|)$ and the maximum arity is 3.

We will often have to go from boolean tuples to stacks and back. The following definition provides appropriate mappings.

Definition 4.8. Fix some $\tau \in \mathsf{Val}$ such that $\tau \neq nil$. Let $b \in \mathcal{B}$, $\mathbf{B} \in \mathcal{B}^n$ and $\mathbf{x} \in \mathsf{Var}^n$. Define functions $\mathrm{sval} : \mathcal{B} \to \mathsf{Val}$, $\mathrm{bval} : \mathsf{Val} \to \mathcal{B}$, stack $s^{\mathbf{x}}_{\mathbf{B}}$ and boolean n-tuple $\mathbf{B}^s_{\mathbf{x}}$ as

$$\mathrm{sval}(b) =_{\text{def}} \begin{cases} \tau & \text{if } b = \top \\ nil & \text{if } b = \bot \end{cases} \qquad \mathrm{bval}(v) =_{\text{def}} \begin{cases} \top & \text{if } v \neq nil \\ \bot & \text{if } v = nil \end{cases}$$

$$s^{\mathbf{x}}_{\mathbf{B}}(x_i) =_{\text{def}} \mathrm{sval}(B_i) \qquad (\mathbf{B}^s_{\mathbf{x}})_i =_{\text{def}} \mathrm{bval}(s(x_i))$$

where $i = 1, \ldots, n$.

Theorem 4.9. *k-PREDSAT is NP-complete for $k \geq 3$.*

Proof. Membership in NP follows from Lemma 4.5. Hardness follows by reducing from CIRCUITSAT via Definition 4.7. We show that there is a boolean k-tuple $\mathbf{B}$ such that $C(\mathbf{B}) = \top$ iff $(\Phi_C, Q_\top) \in k$-PREDSAT, by proving that for any b there exists $\mathbf{B}$ such that $C(\mathbf{B}) = b$ iff $(\Phi_C, Q_b) \in k$-PREDSAT.

($\Rightarrow$) Suppose there exists $\mathbf{B}$ such that $C(\mathbf{B}) = b$. Extend $\mathbf{B}$ to the n-tuple $\mathbf{B'}$ such that for all $i = 1, \ldots, n$, B'_i is equal to the output of gate i. Clearly, $\mathbf{B'}$ is well defined. Let $\hat{s} =_{\text{def}} s^{\mathbf{x}}_{\mathbf{B'}}$. It is easy to see that if $b = \top = B'_n$ then $\hat{s} \models_\Psi T(x_n)$ and that if $b = \bot$ then $\hat{s} \models_\Psi F(x_n)$. Thus in order to show that $\hat{s} \models_{\Phi_C} Q_b$ it remains to show that $\hat{s} \models_{\Phi_C} P(x_n)$. If $n \leq k+2$ this is immediate as the output of C is either a constant or an input. If n is a NAND gate then we must show that for every $i = k+3, \ldots, n$, $B'_{l_i} \uparrow B'_{r_i} = B'_i$ implies $\hat{s} \models_{\Phi_C} N(x_{l_i}, x_{r_i}, x_i)$, where l_i, r_i are the inputs of gate i. This holds by the definitions of predicate N and stack $\hat{s}$.

($\Leftarrow$) Suppose there is a stack s such that $s \models_{\Phi_C} Q_b$. We assume w.l.o.g. that $s \models_{\Phi_C} P(x_n)$ also. Let $\hat{\mathbf{B}} =_{\text{def}} \mathbf{B}^s_{\mathbf{x}}$. Set $\mathbf{B}$ as the k-prefix of $\hat{\mathbf{B}}$. We use induction to prove that for every $i = 1, \ldots, n$, if the inputs are set to $\mathbf{B}$ then the output of gate i is $\hat{B}_i$. The case where $i = 1, \ldots, k+2$ is trivial. If $i = k+3, \ldots, n$ then it is a NAND gate and has inputs $l_i, r_i < i$ whose values we know are equal to $\hat{B}_{l_i}, \hat{B}_{r_i}$. It is then simple to verify that if $s \models_\Psi N(x_{l_i}, x_{r_i}, x_i)$ then it follows that $\hat{B}_{l_i} \uparrow \hat{B}_{r_i} = \hat{B}_i$, by the definitions of N and $\hat{\mathbf{B}}$.

If $b = \top$ then by assumption $s \models_{\Phi_C} P(x_n) * T(x_n)$, so $\hat{B}_n = \top$ and thus $C(\mathbf{B}) = \top$. The case $b = \bot$ is similar. □

For EXPTIME-hardness we will encode natural numbers as boolean tuples as well as formulas, as follows.

Definition 4.10. We use $i^{\mathcal{B}}$ to denote the boolean n-tuple encoding an integer $1 \leq i \leq 2^n$. Note that, for convenience, we set $1^{\mathcal{B}} = \bot^n$ and $(2^n)^{\mathcal{B}} = \top^n$. In addition, we use a formula encoding in terms of the predicates T and F as seen above.

$$\mathrm{frm}_b(x) =_{\text{def}} \begin{cases} T(x) & \text{if } b = \top \\ F(x) & \text{if } b = \bot \end{cases}$$

$$\mathrm{bool}_{\mathbf{B}}(\mathbf{x}) =_{\text{def}} \mathrm{frm}_{B_1}(x_1) * \ldots * \mathrm{frm}_{B_n}(x_n)$$

$$\mathrm{num}_i(\mathbf{x}) =_{\text{def}} \mathrm{bool}_{\left(i^{\mathcal{B}}\right)}(\mathbf{x})$$

The *circuit value problem* (CVP) has as instances input-free circuits. A circuit C is a yes-instance of CVP iff $C() = \top$. The *succinct circuit value problem* (SCVP) has as instances input-free circuits generated by a pair of circuits as explained below. An instance of SCVP is a yes-instance iff for the generated circuit C, $C() = \top$. Formally, an instance $S_C = (L, R)$ of SCVP consists of two circuits, each of $2n$ inputs. In the represented circuit C, for all $i, j \in \{1, \ldots, 2^n\}$ such that $L(i^{\mathcal{B}}, j^{\mathcal{B}}) = \top$ (resp. $R(i^{\mathcal{B}}, j^{\mathcal{B}}) = \top$), it holds that $i > j$, i is a NAND gate and its left (right) input is j. For simplicity we assume that circuits always use 2^n gates and that their output is that of gate 2^n. Also recall that for a k-input circuit, gates $k+1$ and $k+2$ are $\top$ and $\bot$ respectively.

Definition 4.11. Let C be a k-input circuit with n gates. Then, define Φ^r_C as follows, where Ψ is as in Definition 4.7.

$$\Phi^r_C =_{\text{def}} \Psi \cup \left\{ \begin{array}{r} T(x_{k+1}) * F(x_{k+2}) * T(x_n) * \\ *^n_{i=k+3} N(x_{l_i}, x_{r_i}, x_i) \Rightarrow P(x_1, \ldots, x_k) \end{array} \right\}$$

This definition turns the circuit C into a relation P over the variables representing the inputs of C. This will simplify presentation when we are only interested for inputs that make the output equal to $\top$. In particular, the following lemma holds.

Lemma 4.12. *For any circuit C on k-inputs and any $\mathbf{B} \in \mathcal{B}^k$, $C(\mathbf{B}) = \top$ iff $s \models_{\Phi^r_C} P(\mathbf{x}) * \mathrm{bool}_{\mathbf{B}}(\mathbf{x})$ for some stack s.*

Proof. Analogous to the proof of Theorem 4.9. □

Lemma 4.13. *For any k-input circuit C and stacks s, s', if $\mathbf{B}^s_{\mathbf{x}} = \mathbf{B}^{s'}_{\mathbf{x}}$ then $s \models_{\Phi^r_C} P(\mathbf{x})$ if and only if $s' \models_{\Phi^r_C} P(\mathbf{x})$.*

Proof. Observe that $\mathbf{B}^s_{\mathbf{x}} = \mathbf{B}^{s'}_{\mathbf{x}}$ entails that for any variable x_i, $s(x_i) = nil$ iff $s'(x_i) = nil$. The result follows by verifying that satisfaction of predicates T, F, N by a stack s depends only on whether their arguments evaluate to nil or not. □

We now present the reduction from SCVP to PREDSAT.

Definition 4.14. Let $S_C = (L, R)$ be an instance of SCVP. Let Φ^r_L, Φ^r_R be as in Definition 4.11 (we assume names P_L, P_R for the corresponding P predicate). Define rules (U1), (U2), (U3), and inductive rule sets X and Φ_{S_C} as follows.

$$\mathrm{num}_1(\mathbf{x}) * T(v) \Rightarrow U(\mathbf{x}, v) \qquad \text{(U1)}$$

$$\mathrm{num}_2(\mathbf{x}) * F(v) \Rightarrow U(\mathbf{x}, v) \qquad \text{(U2)}$$

$$\begin{array}{r} P_L(\mathbf{x}, \mathbf{l}) * P_R(\mathbf{x}, \mathbf{r}) * \\ U(\mathbf{l}, w) * U(\mathbf{r}, z) * N(w, z, v) \end{array} \Rightarrow U(\mathbf{x}, v) \qquad \text{(U3)}$$

$$X =_{\text{def}} \{\text{(U1)}, \text{(U2)}, \text{(U3)}\} \cup \Phi^r_L \cup \Phi^r_R$$

$$\Phi_{S_C} =_{\text{def}} \left\{ \begin{array}{r} U(\mathbf{x}, v) * T(v) \Rightarrow Q_\top(\mathbf{x}) \\ U(\mathbf{x}, v) * F(v) \Rightarrow Q_\bot(\mathbf{x}) \\ \mathrm{num}_{2^n}(\mathbf{x}) * Q_\top(\mathbf{x}) \Rightarrow R \end{array} \right\} \cup X$$

Thus S_C is mapped to the PREDSAT instance (Φ_{S_C}, R).

Theorem 4.15. *PREDSAT is EXPTIME-complete.*

Proof. By Lemma 4.6, PREDSAT is in EXPTIME. Hardness follows by exhibiting a reduction from SCVP (which is EXPTIME-complete [23]) via Definition 4.14. This follows from the stronger fact that for all $i \in \{1, \ldots, 2^n\}$ and for any boolean b, the output of gate i is b iff there is a stack s such that $s \models_{\Phi_{S_C}} \mathrm{num}_i(\mathbf{x}) * Q_b(\mathbf{x})$. We use induction on i.

($\Rightarrow$) Assume $i = 1$. By assumption, gate 1 is the constant $\top$ thus its output is also $b = \top$. Let $\mathbf{B} =_{\text{def}} i^{\mathcal{B}}$. Define the stack $\hat{s} =_{\text{def}} s^{\mathbf{x}}_{\mathbf{B}}[v \mapsto \mathrm{sval}(\top)]$. By applying (U1) we obtain that $\hat{s} \models_X U(\mathbf{x}, v)$ and as $\hat{s} \models_{\Phi_{S_C}} T(v)$ by construction, it must be that $\hat{s} \models_{\Phi_{S_C}} \mathrm{num}_i(\mathbf{x}) * Q_\top(\mathbf{x})$. The case where $i = 2$ is similar.

Suppose $i > 2$ and that the output of gate i is b. Gate i is a NAND gate thus there are $l, r < i$ such that $L(i^{\mathcal{B}}, l^{\mathcal{B}}) = \top$, $R(i^{\mathcal{B}}, r^{\mathcal{B}}) = \top$, the values of gates l, r are c, d and $c \uparrow d = b$. By the inductive hypothesis we obtain two stacks s_l, s_r with $s_l \models_{\Phi_{S_C}} \mathrm{num}_l(\mathbf{l}) * Q_c(\mathbf{l})$ and $s_r \models_{\Phi_{S_C}} \mathrm{num}_r(\mathbf{r}) * Q_d(\mathbf{r})$. In particular, regardless of the values c, d take, $s_l \models_{\Phi_{S_C}} U(\mathbf{l}, w)$

and $s_r \models_{\Phi_{S_C}} U(\mathbf{r}, z)$ (we set $s_l(w) = s_l(v)$ and $s_r(z) = s_r(v)$ w.l.o.g.). By Lemma 4.12, there exist two stacks s'_l, s'_r such that

$$s'_l \models_{\Phi^r_L} P_L(\mathbf{x}, \mathbf{l}) * \mathrm{num}_i(\mathbf{x}) * \mathrm{num}_l(\mathbf{l})$$
$$s'_r \models_{\Phi^r_R} P_R(\mathbf{x}, \mathbf{r}) * \mathrm{num}_i(\mathbf{x}) * \mathrm{num}_r(\mathbf{r})$$

where Lemma 4.13 lets us assume $s'_l(\mathbf{x}) = s'_r(\mathbf{x})$, $s'_l(\mathbf{l}) = s_l(\mathbf{l})$ and $s'_r(\mathbf{r}) = s_r(\mathbf{r})$. Thus there exists $\hat{s}$ such that $\hat{s}(w) = s_l(w)$, $\hat{s}(z) = s_r(z)$ and

$$\hat{s} \models_{\Phi_{S_C}} P_L(\mathbf{x}, \mathbf{l}) * P_R(\mathbf{x}, \mathbf{r}) * U(\mathbf{l}, w) * U(\mathbf{r}, z) * N(w, z, v)$$

and by (U3), $\hat{s} \models U(\mathbf{x}, v)$. A case analysis on the definition of N allows us to conclude that $\hat{s} \models_{\Phi_{S_C}} \mathrm{num}_i(\mathbf{x}) * Q_b(\mathbf{x})$.

($\Leftarrow$) Let s be a stack such that $s \models_{\Phi_{S_C}} \mathrm{num}_i(\mathbf{x}) * Q_b(\mathbf{x})$. Thus $s \models_{\Phi_{S_C}} U(\mathbf{x}, v)$. If $i \in \{1, 2\}$ then gate i is a constant and the result follows from (U1), (U2).

If $i > 2$ then gate i is a NAND gate and rule (U3) applies:

$$s \models_{\Phi_{S_C}} P_L(\mathbf{x}, \mathbf{l}) * P_R(\mathbf{x}, \mathbf{r}) * U(\mathbf{l}, w) * U(\mathbf{r}, z) * N(w, z, v).$$

Let $l, r \in \{1, \ldots, 2^n\}$ be such that $l^{\mathcal{B}} = \mathbf{B}^s_{\mathbf{l}}$ and $r^{\mathcal{B}} = \mathbf{B}^s_{\mathbf{r}}$. By Lemma 4.12 we can conclude that $L(i^{\mathcal{B}}, l^{\mathcal{B}}) = \top$ and $R(i^{\mathcal{B}}, r^{\mathcal{B}}) = \top$, meaning that l, r are the inputs of gate i. As such, there exist booleans c, d such that $s \models_{\Phi_{S_C}} Q_c(\mathbf{l}) * Q_d(\mathbf{r})$ and by the inductive hypothesis we obtain that the outputs of gates l, r are c, d respectively. It remains to show that $c \uparrow d = b$ which follows by a case analysis on the definition of N. □

The worst case can be exhibited easily through counting.

Proposition 4.16. *There is a family of predicates Φ_n of size $O(n)$ such that the algorithm in Defn. 3.5 runs in $\Omega(2^n)$ time and space.*

Proof. It should be clear that circuits can be rendered as predicates in linear space. In particular, the successor relation over two n-bit numbers can be encoded as a predicate $succ(\mathbf{x}, \mathbf{y})$, for n-variable tuples $\mathbf{x}, \mathbf{y}$. Consider the set of predicates Φ_n.

$$\Phi_n =_{\text{def}} \left\{ \begin{array}{r} \mathrm{num}_1(\mathbf{y}) \Rightarrow Q(\mathbf{y}) \\ succ(\mathbf{x}, \mathbf{y}) * Q(\mathbf{x}) \Rightarrow Q(\mathbf{y}) \\ \mathrm{num}_{(2^n)}(\mathbf{x}) * Q(\mathbf{x}) \Rightarrow P \end{array} \right\}$$

Given (Φ_n, P), the algorithm starts with the base case for Q and counts from 1 to 2^n, creating a base pair encoding each number in between. This will happen irrespective of which strategy is used to select rules or base pairs for instantiation. Clearly, the algorithm will take $\Omega(2^n)$ time. □

5. Implementation and experiments

We implemented our decision procedure as a straightforward rendition of Definition 3.5 (in about 1000 lines of OCaml code) in the theorem proving framework CYCLIST [11], which provides support for logics with inductive predicates. Here we report on the performance of this algorithm on various data sets.

The code, the tool and test files are available online at [1].

5.1 Benchmarks on handwritten predicates

First, we collected a set of 17 standard predicates (defined by 36 rules) from the separation logic literature, defining data structures such as singly- or doubly-linked list segments, possibly-cyclic lists, trees and tree segments. Our decision procedure takes just 4 ms to prove satisfiability of this set of definitions. This is rather to be expected, since all predicates in this test set are easily seen to be satisfiable.

In order to exhibit the worst-case behaviour of our algorithm, we also tested our tool on the family Φ_n of predicates given in Proposition 4.16, using a binary adder implementation of the successor relation. Runtimes, as expected, are exponential in n: cases with $n \leq 4$ are solved in a fraction of a second, the case $n = 5$ is solved in about half a minute, $n = 6$ takes almost 16 minutes, and $n = 7$ times out after 40 minutes.

(a)	Solved in		
Vars	≤ 1 s	≤ 30 s	Sat.
20	94	99	9
30	88	96	17
40	86	95	20
50	**89**	**98**	**37**
60	91	97	28
70	85	97	46
80	89	97	41

(b)	Solved in		
Rules	≤ 1 s	≤ 30 s	Sat.
2×2	100	100	34
3×2	100	100	19
2×3	91	98	32
3×3	**89**	**98**	**37**
4×3	86	96	31
3×4	57	81	24
4×4	47	75	23

(c)	Solved in		
Arity	≤ 1 s	≤ 30 s	Sat.
1	99	100	22
2	96	100	24
3	**89**	**98**	**37**
4	80	90	30
5	72	80	24

(d)	Solved in		
Recs	≤ 1 s	≤ 30 s	Sat.
0	100	100	21
1	100	100	42
2	**89**	**98**	**37**
3	88	96	13
4	85	93	9

Figure 2. Synthetic benchmark performance of our algorithm.

5.2 Benchmarks on automatically abduced predicates

Next, we tested our algorithm on a large collection of predicates generated automatically by the CABER tool, which attempts to abduce inductively defined safety and/or termination preconditions in our logical fragment for pointer programs [10]. We recorded all candidate preconditions generated by CABER over 29 test programs, including back-tracking attempts. This produced 45 945 syntactically unique inductive rule sets, defining from 1–37 predicates with up to 11 parameters and two recursive invocations each. The majority of these definition sets (94%) had no more than 20 predicates; sets with 19 predicates were the most numerous (15%).

We found that our algorithm terminates in less than 50 ms on all tests in this suite, and most definition sets (83%) were satisfiable. We believe this is probably due to the relatively simple recursive structure of the predicates produced by CABER.

5.3 Synthetic benchmarks

To evaluate the scalability of our procedure, we generated test cases drawn from a random distribution with parameters:

- *Vars*: the number of variables in Var;
- $Rules = Preds \times Cases$: the number of predicates and inductive rules for each predicate;
- *Arity*: the arity of all predicates;
- *Eqs*, *Neqs*, *Points*, and *Recs*: the average numbers of equalities, disequalities, points-to literals, and recursive predicate calls, respectively, in each rule.

Each instance consists of $Rules = Preds \times Cases$ independently generated inductive definitions, with all predicates of the same *Arity*. On the rule body, the number for each kind of literal is drawn from a Poisson distribution where the parameter λ is set to, respectively, *Eqs*, *Neqs*, *Points*, and *Recs*. This yields a mix of short and long rule bodies with a specified average length. Furthermore, to have on average one base case for the rule system, with probability $p = 1/Rules$ all the recursive calls in the body of a rule are discarded.

Figure 2 reports our algorithm's performance on instances drawn from this distribution. Each row collects the results of 100 instances randomly generated with various parameter values; the second and third columns report the number of tests solved in under 1 and 30 seconds, while the last one shows the number of solved instances found to be satisfiable. Each of the four sub-tables shows how the statistics vary as one parameter changes while all the rest remain fixed. Although most cases are fairly easy to solve, a few hard instances consistently show up on all parameter settings. For example, with parameters $Vars = 50$, $Rules = 3{\times}3$, $Arity = 3$, and all Eqs, $Neqs$, $Points$, and $Recs$ set to 2 (that is the bold line repeated on all four tables), we find that two very hard instances remain unsolved after the 30 s timeout.

6. Related work

Here we survey the main categories of work related to the contribution of this paper.

6.1 Satisfiability in separation logic with inductive predicates

Recently, interesting progress has been made on satisfiability and entailment problems in various fragments of separation logic with user-defined inductive predicates as considered here (see Section 2). In particular, Iosif et al. [20] propose a sub-fragment of our setting, imposing significant syntactic restrictions on the inductive definitions to enforce *bounded treewidth* for all models of a predicate. Here the treewidth of a heap h is determined by a corresponding graph structure. These restrictions allow them to use a reduction to monadic second-order logic (MSO) for their proof that both satisfiability and entailment are decidable in their fragment of separation logic with inductive definitions. However, these syntactic restrictions disallow many natural inductive predicates; in particular, predicates describing structures with dangling data pointers, such as list or tree segments with extra arbitrary data fields, or structures where potentially no memory is allocated.[1]

In this paper, we only consider satisfiability, not entailment (which is undecidable [2]), but we are not restricted to bounded-treewidth predicates. In addition, we provide a direct decision procedure for satisfiability (as opposed to a reduction proof of decidability) and contribute an analysis of the complexity of satisfiability checking for our fragment.

Considerable research effort has also been expended on the symbolic heap fragment of separation logic with a list segment predicate only. Berdine et al. [3] provided the first decidability result for satisfiability and entailment in this fragment, with a polynomial time procedure given later by Cook et al. [15]. Piskac et al. recently presented another decision procedure, which also works for structures such as sorted list segments and doubly linked lists, based on translating entailments to an intermediate logic that can be handled by an SMT solver [24]. (In very recent work [25], Piskac et al. extended their approach to support also tree-shaped data structures.) Finally, Navarro Pérez and Rybalchenko [22] provided an SMT encoding for satisfiability and entailment in another extension of the fragment allowing pure formulas from arbitrary SMT theories, rather than simple (dis)equalities.

6.2 Satisfiability in other logics with inductive definitions

The consistency of inductive predicates defined by first-order Horn clauses has been widely studied in different contexts. One well-known application of such clausal definitions is Datalog, a rule-based query language for relational databases with ties to logic programming. Datalog rules roughly correspond to our inductive rules where the spatial component of rule bodies is always emp. The evaluation of Datalog queries was shown to be decidable by Shmueli [27] and EXPTIME-complete (see e.g. [16, Theorem 4.5]). However, the interaction between spatial conjunction ($*$), allocation ($\mapsto$) and recursion makes a direct translation to Datalog impractical because it imposes a global constraint; at the same time we found that reducing from Datalog for our lower bounds is possible but no simpler than our circuit-based method.

More recently, Hoder et al. [19] describe a Datalog-based engine, μZ, for the fixed point computation of inductive definitions. In contrast to our approach, they compute concrete fixed points based on explicit underlying ground facts (i.e., they focus on *model checking*, as opposed to satisfiability checking).

More generally, there has been some interest from the program analysis and verification community in the satisfiability of Horn and Horn-like clauses. Bjørner et al. [7] advocate such clauses as an interchange format for software model checking tools, while Grebenshchikov et al. [18] describe an abstraction-based procedure for finding models and checking satisfiability of Horn-like clauses in the context of program analysis.

6.3 Separation logic tools with general inductive predicates

Several analysis tools based on separation logic allow the user to provide their own inductive definitions for spatial predicates. We have already mentioned in our evaluation (Section 5) the theorem prover CYCLIST [11], which treats separation logic with user-defined inductive predicates, and the related abductive prover CABER [10], which automatically infers inductive predicate definitions as safety/termination preconditions for `while` programs. Another such tool is THOR [21], which proves memory safety and generates sound arithmetic abstractions of heap programs (w.r.t. safety and termination). In THOR the specification and predicate definitions must be entered manually. User-defined inductive predicates are also employed in HIP/SLEEK [14], a combined theorem prover and verification system for a C-like language. Finally, the shape analysis in [13] infers inductive definitions based on "structural invariant checkers" provided by the user.

7. Conclusion and future work

The decidability status of satisfiability in the symbolic heap fragment of separation logic with general inductive predicates has stood open for some time. Following the recent achievement of a partial positive answer to this question in [20], here we resolve the general case affirmatively. Our decidability proof has the advantage of being constructive: we give a decision procedure for checking the satisfiability of inductively defined predicates and of individual rules defining those predicates.

We show that our satisfiability problem is EXPTIME-complete in the general case, and that it is still NP-complete when the inductive predicates are restricted to at most $k \geq 3$ arguments. Despite these high complexities, our experiments indicate that, for predicate definitions arising in practice, our prototype implementation is typically able to solve the decision problem in a matter of milliseconds. This opens up a number of interesting potential applications in the automatic verification of programs based on our fragment of separation logic (which could be used, e.g., to consider heap-based programs employing arbitrary data structures, rather than just lists):

- First, automatic verification or *shape analysis* based on separation logic (cf. [4, 5, 12]) typically employs symbolic states based on disjunctions of symbolic heaps. Our algorithm could thus be used to quickly eliminate unsatisfiable disjuncts from symbolic states; it seems plausible that this might yield non-trivial reductions in the time and space costs of such analyses.
- Second, although our decision procedure for satisfiability cannot be used to *decide* the validity of entailments (which is im-

[1] For further discussion of the limitations of this fragment of separation logic we refer to [20].

possible in general), it does nevertheless provide a quick *partial* method for proving or disproving such entailments. On the one hand, any entailment $F \vdash G$ with an unsatisfiable antecedent F is trivially valid. On the other hand, $F \vdash G$ must be invalid if the base pairs for F and G imply the existence of a model satisfying F but not G. To illustrate this point, consider the usual "list segment" predicate of separation logic, defined by

$$x = y : \mathsf{emp} \Rightarrow \mathsf{ls}(x, y)$$
$$x \mapsto z * \mathsf{ls}(z, y) \Rightarrow \mathsf{ls}(x, y)$$

Now, consider the (invalid) entailment $\mathsf{ls}(x, y) \vdash \mathsf{ls}(y, x)$. Computing the base pairs for both sides yields

$$(base\ \mathsf{ls})(x, y) = \{(\emptyset, \{x = y\}),\ (\{x\}, \emptyset)\}$$
$$(base\ \mathsf{ls})(y, x) = \{(\emptyset, \{y = x\}),\ (\{y\}, \emptyset)\}$$

The second base pair for $\mathsf{ls}(x, y)$ implies the existence of a model for $\mathsf{ls}(x, y)$ in which x is allocated and y is not, and hence $x \neq y$ in this model. However, the base pairs for $\mathsf{ls}(y, x)$ tell us that y must be allocated in any model of $\mathsf{ls}(y, x)$ where $x \neq y$. That is, there is a model of $\mathsf{ls}(x, y)$ that is not a model of $\mathsf{ls}(y, x)$. We believe that this sort of reasoning should be quite straightforward to automate.

- Third, our satisfiability procedure can be used to check the sanity of, and/or remove unsatisfiable clauses from, the definitions of inductive predicates either written by programmers, or inferred automatically (cf. [10]).
- Finally, beyond deciding the satisfiability of inductive predicates, the set of base pairs computed by our procedure also provides a useful partition of the space to search for models. While Gherghina et al. [17] have shown that a manual case analysis of inductive definitions can guide proof techniques and improve their efficiency; our decision procedure could automate the case analysis required to do so.

Taking a different direction for future work, one could investigate whether our techniques for deciding satisfiability extend to more general variants of separation logic, where general inductive predicates are still allowed, but the general format of formulas is less restricted. Possible candidates for such extensions include: higher-order separation logic [6]; the fragment in which formulas may contain pure assertions beyond (dis)equalities [22]; and separation logic with fractional permissions [8].

Acknowledgements. We wish to thank the anonymous reviewers for their valuable comments, which have helped us greatly in improving the presentation of the paper.

References

[1] Satisfiability checker for separation logic with inductive definitions. `https://github.com/ngorogiannis/cyclist/releases/tag/CSL-LICS14`.

[2] T. Antonopoulos, N. Gorogiannis, C. Haase, M. Kanovich, and J. Ouaknine. Foundations for decision problems in separation logic with general inductive predicates. In *FoSSaCS'14*, volume 8412 of *LNCS*, pages 411–425, 2014.

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS'04*, volume 3328 of *LNCS*, pages 97–109, 2004.

[4] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV'07*, volume 4590 of *LNCS*, pages 178–192, 2007.

[5] J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory safety for systems-level code. In *CAV'11*, volume 6806 of *LNCS*, pages 178–183, 2011.

[6] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM TOPLAS*, 29(5), 2007.

[7] N. Bjørner, K. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. In *SMT'12*, pages 3–11, 2012.

[8] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *POPL'05*, pages 259–270, 2005.

[9] J. Brotherston. Formalised inductive reasoning in the logic of bunched implications. In *SAS'07*, volume 4634 of *LNCS*, pages 87–103, 2007.

[10] J. Brotherston and N. Gorogiannis. Cyclic abduction of inductively defined safety and termination preconditions. Technical Report RN/13/14, University College London, 2013.

[11] J. Brotherston, N. Gorogiannis, and R. L. Petersen. A generic cyclic theorem prover. In *APLAS'12*, volume 7705 of *LNCS*, pages 350–367, 2012.

[12] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. of the ACM*, 58(6): 26, 2011.

[13] B.-Y. E. Chang, X. Rival, and G. Necula. Shape analysis with structural invariant checkers. In *SAS'07*, volume 4634 of *LNCS*, pages 384–401, 2007.

[14] 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.

[15] B. Cook, C. Haase, J. Ouaknine, M. J. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR'11*, volume 6901 of *LNCS*, pages 235–249, 2011.

[16] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3): 374–425, 2001.

[17] C. Gherghina, C. David, S. Qin, and W.-N. Chin. Structured specifications for better verification of heap-manipulating programs. In *FM'11*, volume 6664 of *LNCS*, pages 386–401, 2011.

[18] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI'12*, pages 405–416, 2012.

[19] K. Hoder, N. Bjørner, and L. de Moura. μZ– an efficient engine for fixed points with constraints. In *CAV'11*, volume 6806 of *LNCS*, pages 457–462, 2011.

[20] R. Iosif, A. Rogalewicz, and J. Simacek. The tree width of separation logic with recursive definitions. In *CADE'13*, volume 7898 of *LNAI*, pages 21–38, 2013.

[21] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL'10*, pages 211–222, 2010.

[22] J. A. Navarro Pérez and A. Rybalchenko. Separation logic modulo theories. In *APLAS'13*, volume 8301 of *LNCS*, pages 90–106, 2013.

[23] C. H. Papadimitriou and M. Yannakakis. A note on succinct representations of graphs. *Inf. Control*, 71(3):181–185, Dec. 1986.

[24] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using SMT. In *CAV'13*, volume 8044 of *LNCS*, pages 773–789, 2013.

[25] R. Piskac, T. Wies, and D. Zufferey. Enabling automated reasoning about separation logic of trees with data. In *CAV'14*, 2014. To appear.

[26] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74, 2002.

[27] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *PODS'87*, pages 237–249, 1987.

[28] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV'08*, volume 5123 of *LNCS*, pages 385–398, 2008.