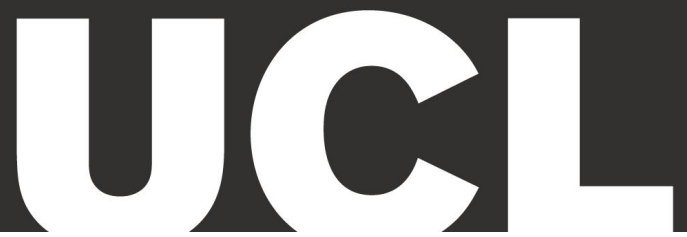
UCL DEPARTMENT OF
COMPUTER SCIENCE



Research Note
RN/13/15

A Decision Procedure for Satisfiability in Separation Logic with Inductive Predicates

July 12, 2013

James Brotherston Carsten Fuhs Nikos Gorogiannis
Juan Navarro Perez

Abstract

In this paper we show that the satisfiability problem for a fragment of separation logic with general inductively defined predicates, commonly employed in program verification, is decidable. Our decision procedure computes a fixed point corresponding to the "base" of an inductive predicate that exactly characterises its satisfiability. The decision procedure then extends from inductive predicates to arbitrary separation logic formulas in our fragment in a straightforward manner.

A complexity analysis of our decision procedure shows that it runs, in the worst case, in exponential time. This is optimal since we also show the satisfiability problem for our inductive predicates to be EXPTIME-complete (by reduction from the succinct circuit value problem). In addition, we show that when the arity of predicates is bounded by a constant, the problem becomes NP-complete.

Finally, we provide an implementation of our decision procedure, and analyse its performance on a range of formulas that are either automatically generated or else encountered in the separation logic literature. For the large majority of these test cases, our tool reports times in the low milliseconds.

A Decision Procedure for Satisfiability in Separation Logic with Inductive Predicates

James Brotherston * Carsten Fuhs † Nikos Gorogiannis ‡
Juan Navarro Pérez
Dept. of Computer Science, University College London, UK

Abstract

In this paper we show that the satisfiability problem for a fragment of separation logic with general inductively defined predicates, commonly employed in program verification, is decidable. Our decision procedure computes a fixed point corresponding to the "base" of an inductive predicate that exactly characterises its satisfiability. The decision procedure then extends from inductive predicates to arbitrary separation logic formulas in our fragment in a straightforward manner.

A complexity analysis of our decision procedure shows that it runs, in the worst case, in exponential time. This is optimal since we also show the satisfiability problem for our inductive predicates to be EXPTIME-complete (by reduction from the succinct circuit value problem). In addition, we show that when the arity of predicates is bounded by a constant, the problem becomes NP-complete.

Finally, we provide an implementation of our decision procedure, and analyse its performance on a range of formulas that are either automatically generated or else encountered in 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 [20, 23] is an established and fairly popular formalism for verifying imperative, heap-manipulating programs. At the time of writing, there is a number of automatic program verification tools based on separation logic, such as SLAYER [2] and ABDUCTOR [8], capable of establishing memory safety properties of code bases extending into millions of lines [25]. 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 [6]. In this situation, however, it becomes much more difficult to determine whether a given formula (containing arbitrary inductive predicates) is *consistent* or not, which can lead to considerable performance problems as time is spent on the analysis of scenarios which are, in fact, unsatisfiable.

In this paper we address the latter problem by showing that the satisfiability problem for the most commonly considered 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 possible satisfying model, the subset of its arguments that are required to be allocated in memory, as well as the equalities and disequalities that must hold between its 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. Furthermore, having computed the base of all required predicates, our procedure is also able to effectively decide the satisfiability of other formulas in the fragment in which these predicates may occur.

We undertake a complexity analysis of our decision procedure which shows that, in the worst case, it runs in exponential time in the size of the underlying set of inductive definitions. This is, essentially, because our inductive definition schema allows us to construct sets of definitions that admit an exponential number of base pairs. Indeed, we show that the satisfiability problem for our inductive predicates is EXPTIME-complete. Additionally, if the maximum number of arguments among all inductive predicates is bounded, then the satisfiability problem becomes NP-complete.

We also provide an implementation of our decision procedure that is capable of taking as input separation logic formulas and inductive definition sets in the format employed by existing tools such as CYCLIST [7]. We evaluate the performance of this algorithm on a large number of inductive predicates automatically generated by the predicate inference tool in [6], and on a number of other examples drawn from the literature on separation logic veri-

* Research supported by an EPSRC Career Acceleration Fellowship.
† Research supported by EPSRC "Resource Reasoning" grant
‡ Research supported by EPSRC grant EP/H008373/1.

fication. To evaluate the scalability of the approach, we also benchmark the implementation against synthetically generated examples with varying parameters. Although it is possible to produce exponential time performance by a suitable choice of parameters, we find that for the vast majority of examples in our test suite, the algorithm terminates in a matter of milliseconds. This suggests that our decision procedure might have significant applications as a black-box satisfiability tool in automated separation logic verification.

We remark that the problem of satisfiability for separation logic with inductive definitions was recently considered by Iosif et al. [16]. Their paper establishes decidability results for both satisfiability *and* entailment problems via an embedding into monadic second order logic, but is restricted to inductive predicates of bounded *treewidth* which, for instance, disallows structures with dangling data pointers. Compared to their work, while we do not consider entailments between formulas, our decidability result for satisfiability holds for a much larger class of inductive predicates; in addition, we provide complexity results and an implementation for our decision procedure.

The remainder of this paper is structured as follows. First, in Section 2, we present an extended example illustrating how to compute the base of an inductive predicate, which is the main idea underlying our decision procedure for satisfiability. We formally introduce our fragment of separation logic with inductive definitions in Section 3, and then present our decision procedure and the proof of its correctness in Section 4. Section 5 contains our analysis of the complexity of our algorithm and of the complexity of the satisfiability problem for inductive predicates. Section 6 describes the implementation of our decision procedure and its evaluation. Section 7 surveys related work in the area, and Section 8 concludes.

2. Illustration

Before venturing into the formal details of our development, in this section we motivate the satisfiability problem under consideration, as well as our method for solving it, with a simple example.

Consider the following four *inductive rules* defining two mutually recursive predicates P and Q:

$$x = \mathsf{nil} \Rightarrow P(x) \qquad \text{(P1)}$$

$$x \neq \mathsf{nil} : Q(x, x) \Rightarrow P(x) \qquad \text{(P2)}$$

$$y = \mathsf{nil} : x \mapsto (d, c) * P(d) \Rightarrow Q(x, y) \qquad \text{(Q1)}$$

$$y \mapsto (d, c) * Q(x, c) \Rightarrow Q(x, y) \qquad \text{(Q2)}$$

This set of rules, in fact, has been automatically inferred by the program analysis tool in [6] as a tentative definition of the safety precondition $P(x)$ for a program that traverses a list of lists. The body of each rule (on the left of $\Rightarrow$) has a *pure* part, containing only equalities and disequalities, and a *spatial* part that symbolically describes a heap. A *points-to* formula, e.g. $x \mapsto (d, c)$, denotes a heap with exactly one memory cell, the address of which is x and contains the data pair (d, c). A separating conjunction $A * B$ of formulas is true on those heaps that can be partitioned into two disjoint subheaps, one satisfying A and the other B. A missing (empty) spatial part denotes the empty heap.

In each rule, variables occurring on the left- but not the right-hand side are implicitly existentially quantified. Often, informally, we say that variables on the right-hand side are "externally visible", while all others are not. These rules can be applied in a bottom-up manner to determine variable and heap assignments that satisfy each one of the predicates. We are interested, thus, in discovering whether some of the rules, or in fact some of the predicates, are *unsatisfiable* and may be safely discarded from the rule set.

From rule (P1), for example, we know that $P(x)$ is satisfied when $x \coloneqq nil$ and the heap does not contain any memory cells. Similarly, following rule (Q1), $Q(x, y)$ is satisfied when, for instance, $x \coloneqq 3$, $y \coloneqq nil$ and the heap consists of a single cell allocated at address 3 with content $(nil, 7)$. This follows since, under this assignment, the pure part of the rule ($y = \mathsf{nil}$) is true and, letting the existentially quantified $c \coloneqq 7$ and $d \coloneqq nil$, we know that $x \mapsto (d, c)$ is satisfied by the heap $h \coloneqq [3 \mapsto (nil, 7)]$, while $P(d)$ is satisfied by the empty heap. More generally, $Q(x, y)$ is always satisfied when x is any (non-nil) location, $y \coloneqq nil$ and the heap is $h \coloneqq [x \mapsto (\mathsf{nil}, c)]$ where c holds an arbitrary value. Armed with this information, and following a similar line of reasoning, we can recursively apply rule (Q2) to discover infinitely many new satisfying assignments for $Q(x, y)$, namely:

$$\begin{array}{llll} x \coloneqq 3 & y \coloneqq 5 & h \coloneqq [5 \mapsto (1, nil), 3 \mapsto (nil, 7)] \\ x \coloneqq 3 & y \coloneqq 4 & h \coloneqq [4 \mapsto (1, 5), 5 \mapsto (1, nil), 3 \mapsto (nil, 7)] \\ \vdots \end{array}$$

So far, we have not had a chance to satisfy the body of rule (P2), since $x \neq y$ in all models of $Q(x, y)$ found so far. But how can we prove, without computing the infinitely many models satisfying Q, that no suitable assignment will be eventually found? For this we introduce the concept of the *base* of an inductively defined predicate. Intuitively, the base keeps track of: (1) the externally visible variables that are necessarily allocated in any heap satisfying the body of a rule and (2) the set of equalities and disequalities that must be satisfied among these variables. Two key results show that the base is indeed an useful device to answer questions about satisfiability of inductive predicate definitions:

1. The base computation eventually reaches a fix-point and terminates after a finite number of steps.
2. The base of a predicate (resp. a rule) exactly characterises its satisfiability. That is: a predicate (resp. a rule) is unsatisfiable if and only if its base is empty.

Starting again on the same example, from rule (P1) we initially discover that the pair $(\emptyset, x = \mathsf{nil})$ is in the base of $P(x)$. That is, nothing is necessarily allocated by the predicate, but the equality $x = \mathsf{nil}$ must hold for it to be satisfied. From rule (Q1) we discover that the pair $(\{x\}, \{y = \mathsf{nil}, x \neq \mathsf{nil}\})$ is in the base of $Q(x, y)$. That is, in models of this rule x must definitely be allocated, and both $y = \mathsf{nil}$, $x \neq \mathsf{nil}$ must hold. Similarly, from rule (Q2) we now discover that

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

is also in the base of $Q(x, y)$. The variable x must be allocated because, so far, it *is* allocated in all base pairs of Q, while y is explicitly allocated by the body of the rule. Also from the previous base we inherit the fact that $x \neq \mathsf{nil}$, while $y \neq \mathsf{nil}$ follows because y is allocated and, similarly, $x \neq y$ is deduced because *both* x and y must be allocated on *disjoint* portions of the heap. Note that facts about non-externally visible variables, such as the inherited $c = \mathsf{nil}$, are not recorded in the base. For the technical details of this computation we refer the reader to Example 4.5 developed later.

Recursively applying rule (Q2) on the newly discovered base pair, and after discarding facts about non-externally visibles, we reproduce exactly the same base pair. So infinitely many satisfying assignments for $Q(x, y)$ have been condensed into a single element in its base. Now it is also clear that $Q(x, x)$ is unsatisfiable as, in the only two available pairs in its base, the values of the first and second argument *must* be distinct from each other.

3. Inductive definitions in separation logic

Here we present our fragment of inductive definitions in separation logic, following the approach in [5].

3.1 Syntax

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; in particular, we abbreviate by $\mathbf{P}$ the tuple $(P_1, \ldots, P_n)$. 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 $\mathbf{x}$.

Definition 3.1. *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 $P_i\mathbf{t}$).

A *symbolic heap* is given by $\Pi : F$, where F is a spatial formula and Π is a finite set of pure formulas. Whenever one of Π, F is empty, we will omit the semicolon.

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 3.2. 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.

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.

3.2 Semantics

We use a typical RAM model employing heaps of records. We fix an infinite set Val of *values*, 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 $nil \in \mathsf{Val} \setminus \mathsf{Loc}$ 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 tuples of values; we fix

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

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. Finally, a model is a stack and heap pair.

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. We write $s, h \models_\Phi \Pi : F$, where $\Pi : F$ is a symbolic heap, to mean that $s, h \models_\Phi F$ and $s, h \models_\Phi G$ for all $G \in \Pi$. We say that $\Pi : F$ is *satisfiable*, for a fixed Φ, if there is some stack s and heap h such that $s, h \models_\Phi \Pi : F$.

We remark that satisfaction of a pure formula Π does not depend on either the heap or the inductive rules: we write $s \models \Pi$ to mean that $s, h \models_\Phi \Pi$ for any heap h and inductive definition set Φ. Furthermore, we note that Π determines an equivalence relation among terms. In particular we write $t \simeq_\Pi t'$ if the set $\Pi \cup \{t \neq t'\}$ is unsatisfiable and $\langle t \rangle_\Pi$ to denote the equivalence class of t, i.e., the set of all terms t' such that $t \simeq_\Pi t'$. In a slight abuse of notation, we also write $\langle \mathbf{t} \rangle_\Pi$ to denote the union of the equivalence classes of all terms $t \in \mathbf{t}$.

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

Definition 3.3. For each predicate P_i with arity a_i, $1 \leq i \leq n$, we define $\tau_i = \mathrm{Pow}(\mathsf{Val}^{a_i} \times \mathsf{Heap})$ where $\mathrm{Pow}(-)$ is powerset.

Partition Φ into $\Phi_1, \ldots, \Phi_n$, where $\Phi_i \subseteq \Phi$ is the set of inductive rules 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:

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

where $\models_{\mathbf{x}}$ is the satisfaction relation defined above, except that $[\![P_i]\!]^{\mathbf{X}} =_{\text{def}} X_i$, where $\mathbf{X} = (X_1, \ldots, X_n)$. We then finally define the tuple $[\![\mathbf{P}]\!]^\Phi \in \tau_1 \times \ldots \times \tau_n$ by:

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

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

Example 3.4. Consider again the rules from the example in Section 2, where the rules in Φ are now partitioned as follows:

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

Here $\tau_1 = \mathrm{Pow}(\mathsf{Val} \times \mathsf{Heap})$ and $\tau_2 = \mathrm{Pow}(\mathsf{Val} \times \mathsf{Val} \times \mathsf{Heap})$ correspond, respectively, to all sets of potential models of the two predicates $\mathbf{P} = \{P, Q\}$. The function $\varphi_{i,j} : \tau_1 \times \tau_2 \mapsto \tau_i$ maps known models of P and Q to a set of new models deduced by the rule $\Phi_{i,j}$. Initially $\mathbf{X}^0 = (\emptyset, \emptyset)$, that is no known models.

After the first iteration $\varphi_{1,1}(\mathbf{X}^0) = \{(nil, e)\}$ contains the only model of P generated by the rule $\Phi_{1,1}$, while $\varphi_{i,j}(\mathbf{X}^0) = \emptyset$ for all three other rules. Thus $\mathbf{X}^1 = (\{(nil, e)\}, \emptyset)$. On the second iteration we now discover

$$\varphi_{2,1}(\mathbf{X}^1) = \{(x, nil, [x \mapsto (nil, c)]) \mid x \in \mathsf{Loc} \text{ and } c \in \mathsf{Val}\}\ ,$$

all new models of Q; while $\varphi_{i,j}(\mathbf{X}^1) = \varphi_{i,j}(\mathbf{X}^0)$ remain for all other three rules and thus $\mathbf{X}^2$ becomes $(\{(nil, e)\}, \varphi_{2,1}(\mathbf{X}^1))$.

Then, on the third iteration,

$$\varphi_{2,2}(\mathbf{X}^2) = \{(x, y, [y \mapsto (d, nil), x \mapsto (nil, c)]) \mid x \neq y \in \mathsf{Loc} \text{ and } c, d \in \mathsf{Val}\} ,$$

while $\varphi_{i,j}(\mathbf{X}^2) = \varphi_{i,j}(\mathbf{X}^1)$ remain for all other three rules and thus $\mathbf{X}^3 = (\{(nil, e)\}, \varphi_{2,1}(\mathbf{X}^1) \cup \varphi_{2,2}(\mathbf{X}^2))$.

Further applications of the rule $\Phi_{2,2}$, via $\varphi_{2,2}$, yield larger and larger models for Q, *ad infinitum*.

The fixed point $[\![\mathbf{P}]\!]^{\Phi}$ contains all possible models of P and Q, each found after applying a finite but unbounded number of derivations using rules in Φ.

4. A decision procedure for satisfiability of inductive predicates

Throughout this section we assume, without loss of generality, that sets of pure formulas are ordered as a list of equality formulas $t = t'$ followed by a list of disequality formulas $t \neq t'$. We use the following definition to restrict the variables occurring in a pure formula to only those which are "externally visible". For example, given $\Pi = \{x = c, c \neq y, c \neq d\}$ and the externally visible variables $\mathbf{x} = (x, y)$, then the restricted $\Pi \upharpoonright \mathbf{x}$ is $\{x \neq y\}$.

Definition 4.1. Let Π be a finite set of pure formulas and let $\mathbf{x}$ be a tuple of variables. We define the set of pure formulas $\Pi \upharpoonright \mathbf{x}$ by induction on Π as follows, where comma is to be read as set union in the obvious way:

$$\emptyset \upharpoonright \mathbf{x} =_{\text{def}} \emptyset$$

$$(t_1 \neq t_2, \Pi) \upharpoonright \mathbf{x} =_{\text{def}} \begin{cases} t_1 \neq t_2, (\Pi \upharpoonright \mathbf{x}) & \text{if } t_1, t_2 \in \mathbf{x} \cup \{\mathsf{nil}\} \\ \Pi \upharpoonright \mathbf{x} & \text{otherwise} \end{cases}$$

$$(t_1 = t_2, \Pi) \upharpoonright \mathbf{x} =_{\text{def}} \begin{cases} t_1 = t_2, (\Pi \upharpoonright \mathbf{x}) & \text{if } t_1, t_2 \in \mathbf{x} \cup \{\mathsf{nil}\} \\ \Pi[t_1/t_2] \upharpoonright \mathbf{x} & \text{if } t_2 \notin \mathbf{x} \cup \{\mathsf{nil}\} \\ \Pi[t_2/t_1] \upharpoonright \mathbf{x} & \text{otherwise} \end{cases}$$

The following pair of lemmas formalise the notion that there is a close connection between models of Π and those of $\Pi \upharpoonright \mathbf{x}$. More specifically, given a model s of Π it is possible to modify s, as long as the values assigned to $\mathbf{x}$ are not changed, and still remain a model of $\Pi \upharpoonright \mathbf{x}$. Conversely, given a model s of $\Pi \upharpoonright \mathbf{x}$, assuming that Π was satisfiable to begin with, we can always tweak s without changing values assigned to $\mathbf{x}$ and recover a model for Π.

Lemma 4.2. *Let Π be a finite set of pure formulas. If $s \models \Pi$ and $s(\mathbf{x}) = s'(\mathbf{x})$ then $s' \models \Pi \upharpoonright \mathbf{x}$.*

Proof. We proceed by induction on the size of the set Π. If Π is empty, we are trivially done. Otherwise, we distinguish cases on the first formula in Π.

Case $\Pi = (t_1 \neq t_2, \Pi')$. By assumption, we have $s \models t_1 \neq t_2$ and $s \models \Pi'$. Thus, by induction hypothesis, $s' \models \Pi' \upharpoonright \mathbf{x}$. We first consider the subcase where $t_1, t_2 \in \mathbf{x} \cup \{\mathsf{nil}\}$. Then, since $s'(\mathbf{x}) = s(\mathbf{x})$, we easily have $s' \models t_1 \neq t_2$ and so $s' \models t_1 \neq t_2, \Pi' \upharpoonright \mathbf{x}$ as required.

If one of $t_1, t_2 \notin \mathbf{x} \cup \{\mathsf{nil}\}$, then we just have to prove $s' \models \Pi' \upharpoonright \mathbf{x}$, which is immediate by the induction hypothesis.

Case $\Pi = (t_1 = t_2, \Pi')$. By assumption, we have $s \models t_1 = t_2$ and $s \models \Pi'$. We first consider the subcase where $t_1, t_2 \in \mathbf{x} \cup \{\mathsf{nil}\}$. By the induction hypothesis, we have that $s' \models \Pi' \upharpoonright \mathbf{x}$. Since $s'(\mathbf{x}) = s(\mathbf{x})$, we have $s' \models t_1 = t_2$. Thus $s' \models t_1 = t_2, \Pi \upharpoonright \mathbf{x}$ as required.

Next we consider the subcase where $t_2 \notin \mathbf{x} \cup \{\mathsf{nil}\}$ (the subcase where $t_2 \in \mathbf{x} \cup \{\mathsf{nil}\}$ and $t_1 \notin \mathbf{x} \cup \{\mathsf{nil}\}$ is similar). Since $s \models t_1 = t_2$ and $s \models \Pi'$, we have $s \models \Pi'[t_1/t_2]$. Thus, by induction hypothesis, we have $s' \models \Pi'[t_1/t_2] \upharpoonright \mathbf{x}$ as required. □

Lemma 4.3. *Let Π be a finite set of pure formulas. If Π is satisfiable and $s \models \Pi \upharpoonright \mathbf{x}$ then there exists a stack s' with $s'(\mathbf{x}) = s(\mathbf{x})$ such that $s' \models \Pi$.*

Furthermore, for any finite set of locations $W \subseteq \mathsf{Loc}$, and for any variable y we can choose $s'(y)$ such that if $y \notin \langle \mathbf{x} \rangle_\Pi$ then $s'(y) \notin W$.

Proof. By induction on the size of the set Π, i.e. the number of pure formulas it contains. If Π is empty we define

$$s'(y) =_{\text{def}} \begin{cases} s(y) & \text{if } y \in \mathbf{x} \\ nil & \text{otherwise} . \end{cases}$$

It is easy to verify that s' satisfies the conditions of the lemma: by construction $s'(\mathbf{x}) = s(\mathbf{x})$; also if $y \notin \langle \mathbf{x} \rangle_\Pi$ then necessarily $s'(y) = nil \notin W$, since nil is not a location. Otherwise, when Π is not empty, we distinguish the following two cases.

Case $\Pi = (t_1 \neq t_2, \Pi')$. First we examine the subcase where $t_1, t_2 \in \mathbf{x} \cup \{\mathsf{nil}\}$. By assumption, we have $s \models t_1 \neq t_2$ and $s \models (\Pi' \upharpoonright \mathbf{x})$. Thus, by induction hypothesis, there is an s' such that $s'(\mathbf{x}) = s(\mathbf{x})$ and $s' \models \Pi'$. Furthermore, as $t_1, t_2 \in \mathbf{x} \cup \{\mathsf{nil}\}$ by assumption and since $s \models t_1 \neq t_2$, we have $s' \models t_1 \neq t_2$, which gives us $s' \models t_1 \neq t_2, \Pi'$ as required.

Now we examine the subcase where both $t_1, t_2 \notin \mathbf{x} \cup \{\mathsf{nil}\}$ (the subcase where only one of t_1, t_2 is not in $\mathbf{x} \cup \{\mathsf{nil}\}$ is similar). By assumption, we have $s \models \Pi' \upharpoonright \mathbf{x}$. Thus by induction hypothesis, there is a s'' with $s''(\mathbf{x}) = s(\mathbf{x})$ and $s'' \models \Pi'$. Since $t_1, t_2 \neq \mathsf{nil}$, both t_1 and t_2 are variables, and they cannot be the same variable, otherwise Π is unsatisfiable, contradicting the lemma assumption. Thus we define

$$s' =_{\text{def}} s''[t_1 \mapsto \ell_1, t_2 \mapsto \ell_2]$$
$$\text{where } \ell_1, \ell_2 \in \mathsf{Val} \setminus (W \cup s(\mathbf{x}) \cup \{v \mid \exists x \in \Pi' .\ s''(x) = v\})$$
$$\text{and } \ell_1, \ell_2 \text{ distinct}$$

Such ℓ_1 and ℓ_2 can always be found since Val is assumed to be infinite, while the set of disallowed values is finite. Note that our choice of ℓ_1, ℓ_2 ensures that $s'(t_1), s'(t_2) \notin W$ so the second part of the lemma is satisfied. Furthermore, we have $s' \models t_1 \neq t_2$.

It just remains to show that $s' \models \Pi'$. This is not quite trivial because, although we have $s'' \models \Pi'$, it might be that either t_1 or t_2 occurs in Π'. However, because of our assumption about the format of sets of pure formulas (all equalities come before disequalities), Π' must contain only disequality formulas. Thus we have only to ensure that t_1, t_2 are not equal to any other term in Π under s'; but this is precisely guaranteed by our choice of ℓ_1, ℓ_2 above.

Case $\Pi = (t_1 = t_2, \Pi')$. First we examine the subcase where $t_1, t_2 \in \mathbf{x} \cup \{\mathsf{nil}\}$. By assumption, $s \models t_1 = t_2$ and $s \models \Pi' \upharpoonright \mathbf{x}$. Thus, by induction hypothesis, there is an s' such that $s' \models \Pi'$ and $s'(\mathbf{x}) = s(\mathbf{x})$. Since $t_1, t_2 \in \mathbf{x} \cup \{\mathsf{nil}\}$ and $s \models t_1 = t_2$, we have $s' \models t_1 = t_2, \Pi$ as required.

Next we examine the subcase where t_1 is in $\mathbf{x} \cup \{\mathsf{nil}\}$ but t_2 is not. By assumption, $s \models \Pi'[t_1/t_2] \upharpoonright \mathbf{x}$. Thus, by induction hypothesis, there is a stack s'' such that both $s''(\mathbf{x}) = s(\mathbf{x})$ and $s'' \models \Pi'[t_1/t_2]$. As t_2 is assumed to be a variable, the stack given by $s' =_{\text{def}} s''[t_2 \mapsto s''(t_1)]$ is well defined and, by usual facts about substitution, $s' \models \Pi'$. Thus the first part of the lemma holds. For the second part of the lemma, when $y = t_2$, we note that then either $t_1 \in \mathbf{x}$ so that $t_2 \in \langle \mathbf{x} \rangle_\Pi$ and the condition does not apply, or $t_1 = \mathsf{nil}$ so that $s'(t_2) = nil \notin W$.

The other subcases are similar, we just have to make sure to choose $s' =_{\text{def}} s''[t_1 \mapsto s''(t_2)]$ to satisfy $s' \models \Pi'[t_2/t_1]$. Also

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 $\Phi_{i,j} \in \Phi$
    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 $\big((\langle V\rangle_\Pi \cap \mathbf{x})[\mathbf{t}/\mathbf{x}], (\Pi \restriction \mathbf{x})[\mathbf{t}/\mathbf{x}]\big)$ to $Y_i(\mathbf{t})$
  return $\mathbf{Y}$

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

we have that either both $t_1, t_2 \in \langle \mathbf{x}\rangle_\Pi$, and the second part of the lemma does not apply, or both $t_1, t_2 \notin \langle \mathbf{x}\rangle_\Pi$ and in any case we pick a value $s'(t_1) = s'(t_2) \notin W$. This completes the proof. □

The following definition is central to the main contribution of this paper. It constitutes the definition of the *base* operation which is used to capture the satisfiability status of inductive rules and predicates. The pseudocode in Figure 1 is provided as an informal aid to navigate the steps of the computation, but formal and notation details are only explained in the definition.

Definition 4.4. First, for each $1 \le i \le n$ we define

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

where a_i is the arity of the predicate symbol P_i, and $\mathrm{MPow}(X)$ is the set of all multisets over X.

Next, for a multiset V of variables we define $\bigotimes V$ to be the set containing:

- 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.

We partition the inductive rule set Φ into $\Phi_1, \ldots, \Phi_n$ with each Φ_i further indexed by j as in Definition 3.3. Without loss of generality, we consider each inductive rule $\Phi_{i,j} \in \Phi$ to be written in the following 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 Π_0 is a set of pure formulas. 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 including all pairs:

$$\big((\langle V\rangle_\Pi \cap \mathbf{x})[\mathbf{t}/\mathbf{x}], (\Pi \restriction \mathbf{x})[\mathbf{t}/\mathbf{x}]\big)$$

such that:

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

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 3.2. The intersection $\langle V\rangle_\Pi \cap \mathbf{x}$ denotes the *multiset* of variables contained in V which, modulo the equivalence relation induced by Π, also occur in $\mathbf{x}$; that is all $y \in V$ such that $y \equiv_\Pi x$ for some $x \in \mathbf{x}$.

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

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

where by slight abuse notation $\bigcup_j \Psi_{i,j}(\mathbf{Y})$ denotes the function that maps a tuple of terms $\mathbf{t}$ to the set $\bigcup_j \Psi_{i,j}(\mathbf{Y})(\mathbf{t})$. We also write $base^{\Phi}P_i$ as an abbreviation for $\pi_i(base^{\Phi}\mathbf{P})$.

Example 4.5. As an example consider again the set of inductive rules Φ from the previous Example 3.4. In this case each

$$Y_1 \in \sigma_1 = \mathsf{Term} \to \mathrm{Pow}(\mathrm{MPow}(\mathsf{Var}) \times \mathsf{Pure})$$
$$Y_2 \in \sigma_2 = \mathsf{Term} \times \mathsf{Term} \to \mathrm{Pow}(\mathrm{MPow}(\mathsf{Var}) \times \mathsf{Pure})$$

map a corresponding number of terms to respective sets of base pairs for the predicates P and Q.

To compute $base^{\Phi}\mathbf{P}$, initially we start with $\mathbf{Y}^0 = (Y_1^0, Y_2^0)$, where $Y_1^0 = \lambda x.\ \emptyset$ and $Y_2^0 = \lambda x, y.\ \emptyset$. That is, initially the base of both P and Q is empty.

On the first iteration, we want to compute $\Psi_{i,j}(Y_1^0, Y_2^0)$, where both i and j range over $\{1, 2\}$. For the inductive rule $\Phi_{1,1}$, the function $\Psi_{1,1}(Y_1^0, Y_2^0)$ computes $V = \emptyset$ and $\Pi = \{x = \mathsf{nil}\}$. The set Π is clearly satisfiable and, since both V and Π remain the same when restricting variables to the only visible $\{x\}$, we have

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

Since $\mathbf{Y}^0$ does not contain any base pairs yet, all other rules map their respective input terms to $\emptyset$. Thus we finally obtain the new updated $\mathbf{Y}^1 = (Y_1^1, Y_2^1)$ where

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

That is, the base of $P(x)$ now contains the pair $(\emptyset, \{x = \mathsf{nil}\})$ while the base of $Q(x, y)$ remains empty.

On the second iteration, the rule $\Phi_{1,1}$ will generate (as always in the future) the same pair, while both $\Phi_{1,2}$ and $\Phi_{2,2}$ remain unsatisfiable because the base of Q contains no pairs. This time, however, the inductive rule $\Phi_{2,1}$ becomes active, as there is now a pair $(\emptyset, \{d = \mathsf{nil}\}) \in Y_1^1(d)$ produced by the current base of P. We thus have

$$V = \{x\} \qquad \Pi = \{y = \mathsf{nil}, x \neq \mathsf{nil}\} \cup \{d = \mathsf{nil}\}$$

with satisfiable Π. After projecting to variables in $\{x, y\}$

$$\Psi_{2,1}(Y_1^1, Y_2^1) = \lambda x, y.\ \big\{(\{x\}, \{y = \mathsf{nil}, x \neq \mathsf{nil}\})\big\}\ ,$$

and, at the end of the iteration, $\mathbf{Y}^2 = (Y_1^2, Y_2^2)$ where

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

On the third iteration, the rule $\Phi_{1,2}$ becomes interesting, since now $(\{x\}, \{x = \mathsf{nil}, x \neq \mathsf{nil}\}) \in Y_2^2(x, x)$ and, for this rule,

$$V = \{x\} \qquad \Pi = \{x \neq \mathsf{nil}\} \cup \{x = \mathsf{nil}, x \neq \mathsf{nil}\}\ .$$

The resulting Π, however, is not satisfiable and so no new base pair is produced by this rule. The rule $\Phi_{2,1}$ produces the same base pair as before, but the rule $\Phi_{2,2}$ has some new activity since now we have the new pair $(\{x\}, \{c = \mathsf{nil}, x \neq \mathsf{nil}\}) \in Y_2^2(x, c)$. Thus

$$V = \{x, y\}$$
$$\Pi = \textstyle\bigotimes\{x, y\} \cup \{y \neq \mathsf{nil}\} \cup \{c = \mathsf{nil}, x \neq \mathsf{nil}\}$$
$$= \{c = \mathsf{nil}, x \neq y, y \neq \mathsf{nil}, x \neq \mathsf{nil}\}$$

whose projection with respect to the set of variables $\{x, y\}$ only removes the equality $c = \mathsf{nil} \in \Pi$. At the end of the iteration we then have $\mathbf{Y}^3 = (Y_1^3, Y_2^3)$ where

$$Y_1^3 = \lambda x.\ \big\{(\emptyset, \{x = \mathsf{nil}\})\big\}$$
$$Y_2^3 = \lambda x, y.\ \big\{(\{x\}, \{y = \mathsf{nil}, x \neq \mathsf{nil}\}),$$
$$(\{x, y\}, \{x \neq y, y \neq \mathsf{nil}, x \neq \mathsf{nil}\})\big\}\ .$$

On the fourth iteration both $\Phi_{1,1}$ and $\Phi_{2,1}$ produce the same base pairs as before. For the rule $\Phi_{1,2}$ we now have a new pair $(\{x, x\}, \{x \neq x, x \neq \mathsf{nil}, x \neq \mathsf{nil}\}) \in Y_2^3(x, x)$ which, however, one can quickly see that also generates an unsatisfiable set Π. Similarly, for the rule $\Phi_{2,2}$ we also need to consider the new pair $(\{x, c\}, \{x \neq c, c \neq \mathsf{nil}, x \neq \mathsf{nil}\}) \in Y_2^3(x, c)$ which yields

$$V = \{x, y, c\}$$
$$\Pi = \bigotimes\{x, y, c\} \cup \{y \neq \mathsf{nil}\} \cup \{x \neq c, c \neq \mathsf{nil}, x \neq \mathsf{nil}\}$$
$$= \{x \neq y, x \neq c, y \neq c, y \neq \mathsf{nil}, c \neq \mathsf{nil}, x \neq \mathsf{nil}\}\ .$$

Furthermore, the projections with respect to $\{x, y\}$ yield

$$\langle V\rangle_\Pi \cap \{x, y\} = \{x, y\}$$
$$\Pi \restriction \mathbf{x} = \{x \neq y, y \neq \mathsf{nil}, x \neq \mathsf{nil}\}\ .$$

This produces exactly the same base pair as before, so no new pairs are generated, $\mathbf{Y}^4 = \mathbf{Y}^3$ and a fixed point is reached:

$$\mathit{base}^\Phi P(x) = \big\{(\emptyset, \{x = \mathsf{nil}\})\big\}$$
$$\mathit{base}^\Phi Q(x, y) = \big\{(\{x\}, \{y = \mathsf{nil}, x \neq \mathsf{nil}\}),$$
$$(\{x, y\}, \{x \neq y, y \neq \mathsf{nil}, x \neq \mathsf{nil}\})\big\}\ .$$

The following pair of lemmas formalise the stated intuition that $\mathit{base}^\Phi P(\mathbf{x})$, as defined, exactly characterises the satisfiability of an inductively defined predicate $P(\mathbf{x})$. The first of these lemmas, stated in a slightly more general form, shows that if we are given a stack $s \models \Pi$ for a pair $(V, \Pi) \in \mathit{base}^\Phi P(\mathbf{x})$ then it is possible to find a heap h such that $s, h \models_\Phi P(\mathbf{x})$ and, necessarily, all of the values $s(V) \subseteq \mathrm{dom}\, h$ are allocated in the heap. Here, and in the following, we write $s(V)$ where s is a stack and V a multiset of variables to mean $\{s(v) \in \mathsf{Val} \mid v \in V\}$.

Lemma 4.6. *If $(V, \Pi) \in (\mathit{base}^\Phi P_i)(\mathbf{t})$, $s \models \Pi$, and $W \subseteq \mathsf{Loc}$ satisfies $s(V) \cap W = \emptyset$, then 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 $\mathit{base}^\Phi\mathbf{P}$.

We assume that the lemma already 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 that it also holds for $(\bigcup_j \Psi_{1,j}(\mathbf{Y}), \ldots, \bigcup_j \Psi_{n,j}(\mathbf{Y}))$. Thus, we assume $(V, \Pi) \in \Psi_{i,j}(\mathbf{Y})(\mathbf{t})$ with $s \models \Pi$ and $s(V) \cap W = \emptyset$. Our task is to find a heap h with $s, h \models P_i\mathbf{t}$ and $\mathrm{dom}(h) \cap W = \emptyset$.

By assumption, there is an inductive rule $\Phi_{i,j}$ of the form (IndRule) above such that $(V, \Pi) \in \Psi_{i,j}(\mathbf{Y})(\mathbf{t})$. Therefore we have $V = (\langle V'\rangle_{\Pi'} \cap \mathbf{x})[\mathbf{t}/\mathbf{x}]$ and $\Pi = (\Pi' \restriction \mathbf{x})[\mathbf{t}/\mathbf{x}]$, where the following hold:

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

By the lemma assumption and usual substitution facts, we have $s[\mathbf{x} \mapsto s(\mathbf{t})] \models \Pi' \restriction \mathbf{x}$ and $s[\mathbf{x} \mapsto s(\mathbf{t})](\langle V'\rangle_{\Pi'} \cap \mathbf{x}) \cap W = \emptyset$. Since Π' is satisfiable, we can apply Lemma 4.3 to obtain s' with $s'(\mathbf{x}) = s[\mathbf{x} \mapsto s(\mathbf{t})](\mathbf{x}) = s(\mathbf{t})$ and $s' \models \Pi'$. Furthermore, for any variable $y \in V'$, if $y \notin \langle\mathbf{x}\rangle_\Pi$ from the lemma application it follows that $y \notin W$. Alternatively, if $y \simeq_\Pi x$ for some $x \in \mathbf{x}$, then $s'(y) = s'(x) = s[\mathbf{x} \mapsto s(\mathbf{t})](x)$ and since $x \in \langle V'\rangle_{\Pi'} \cap \mathbf{x}$ but $s[\mathbf{x} \mapsto s(\mathbf{t})](\langle V'\rangle_{\Pi'} \cap \mathbf{x}) \cap W = \emptyset$, it must be the case that $s'(y) \notin W$. Thus, in either case, we have $s'(V') \cap W = \emptyset$.

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

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

We show inductively how to construct h_ℓ given the chain of heaps $(h_p)_{1\leq p<\ell}$.

***Inductive construction of** $h_1, \ldots, h_m$.* We state and prove the claim 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 by the induction hypothesis we have $\mathrm{dom}(h_p) \cap W_p = \emptyset$, which 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 $V' \supseteq V_\ell, V_q$. Thus $s'(V_\ell) \cap \bigcup_{\ell<q\leq m} s'(V_q) = \emptyset$. Finally, for a similar reason, $s'(V_\ell) \cap s'(\{y_1, \ldots, y_k\}) = \emptyset$.

Now since $(V_\ell, \Pi_\ell) \in Y_{j_\ell}(\mathbf{x}_\ell)$, $s' \models \Pi_\ell$, and $s'(V_\ell) \cap W_\ell = \emptyset$, we can apply the main induction hypothesis of the lemma to obtain 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.

Now we continue with the main proof. We 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 $s'(\{y_1, \ldots, y_k\})$ by: $h'(s'(y_i)) =_{\text{def}} s'(\mathbf{u_i})$ 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:

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

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

Furthermore, 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 $s'(V') \cap W = \emptyset$. This completes the proof. □

The following lemma proves the inverse direction of the former. Here, we show that if the inductive predicate $P(\mathbf{x})$ is satisfiable, then there must be a satisfiable witness base pair in $\mathit{base}^\Phi P(\mathbf{x})$.

Lemma 4.7. *If $s, h \models_\Phi P_i\mathbf{t}$ then there is $(V, \Pi) \in (\mathit{base}^\Phi P_i)(\mathbf{t})$ such that $s(V) \subseteq \mathrm{dom}(h)$ and $s \models \Pi$.*

Proof. We have $(s(\mathbf{t}), h) \in [\![P_i]\!]^\Phi$, 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, we assume $(s(\mathbf{t}), h) \in \varphi_{i,j}(\mathbf{X})$ and must find a pair $(V, \Pi) \in (\mathit{base}^\Phi P_i)(\mathbf{t})$ such that $s(V) \subseteq \mathrm{dom}(h)$ 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

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

Thus $s' \models \Pi_R$ 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 $1 \leq \ell \leq m$. By the induction hypothesis, we thus have for all

$1 \leq \ell \leq m$ that there are pairs $(V_\ell, \Pi_\ell) \in (base^\Phi P_{j_\ell})(\mathbf{x}_\ell)$, 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

$$s' \models \Pi_R \cup \bigotimes V \cup \{\Pi_1, \ldots, \Pi_m\} \text{ and } s'(V) \subseteq \mathrm{dom}(h)$$

Thus $\Pi = \Pi_R \cup \bigotimes V \cup \{\Pi_1, \ldots, \Pi_m\}$ is satisfiable, and so

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

where the substitution $[\mathbf{t}/\mathbf{x}]$ is well defined as $\mathbf{x}$ is a tuple of distinct variables. Since $s'(\mathbf{x}) = s(\mathbf{t})$, also $s'(\mathbf{x}) = s[\mathbf{x} \mapsto s(\mathbf{t})](\mathbf{x})$. By Lemma 4.2, 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, using the 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 $(V, \Pi) \in (base^\Phi P_i)(\mathbf{t})$ such that $s \models \Pi$ and $s(V) \subseteq \mathrm{dom}(h)$ as required. □

Putting all of the ingredients together, the following pair of theorems establish the correctness and termination of the approach, both for individual inductive predicates and for separation logic formulas containing those predicates. The focus of the next section is to determine exact bounds for the complexity of the approach.

Theorem 4.8. *A formula of the form $P_i\mathbf{t}$ is satisfiable if and only if there exists a pair $(V, \Pi) \in (base^\Phi P_i)(\mathbf{t})$ such that Π is satisfiable. Furthermore, for any given rule set Φ, the fixed point computation $base^\Phi \mathbf{P}$ terminates after a finite number of steps.*

Proof. The "if" direction follows immediately from Lemma 4.6 by taking $W = \emptyset$. The "only if" direction follows immediately from Lemma 4.7.

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

Corollary 4.9. *Satisfiability checking of symbolic heaps of the form $\Pi : F$, with respect to a collection of inductive predicates defined by a rule set Φ, is a decidable problem.*

Proof. Consider the rule set $\Psi = \Phi \cup \{\Pi : F \Rightarrow Q\}$ where Q is a new 0-ary predicate not already defined in Φ. From the previous theorem it follows that the symbolic heap $\Pi : F$ is satisfiable if and only if $base^\Psi Q(\,)$ is not empty. □

5. Complexity

In the previous section we showed that our satisfiability checking algorithm terminates and, thus, it constitutes a decision procedure. Now this section investigates, in further depth, its theoretical complexity. We define the decision problem PREDSAT as having instances of the form (Φ, P) where Φ is a set of rules and P a predicate defined in Φ. A yes-instance of PREDSAT is one where there exists a model s, h such that $s, h \models_\Phi P(\mathbf{x})$. We also introduce k-PREDSAT, a restricted form of PREDSAT where all predicates defined in Φ have arity bounded by the constant k. We use $\|o\|$ to denote the length of the encoding of an object o under some reasonable encoding scheme. Clearly, $O(\|(\Phi, P)\|) = O(\|\Phi\|)$. Finally we denote by $\mathcal{B}$ the set $\{\top, \bot\}$.

For a set of rules Φ let α be the maximum arity of any predicate in Φ plus one; clearly, α has order $O(\|\Phi\|)$. Thus, the length of a base pair is bounded by $\alpha + 2\alpha^2$ of order $O(\|\Phi\|^2)$ (size of the variable set plus the size of the longest pure formula). Consequently, the number of distinct base pairs for any given predicate in Φ is bounded by $N =_{\text{def}} 2^{\alpha+2\alpha^2}$. Finally we use L to denote the maximum length of any rule; as such, L is of order $O(\|\Phi\|)$.

Lemma 5.1. *Let $\Phi_{i,j}$ be a rule in the form of (IndRule) and a tuple $T = \langle B_1, \ldots, B_m \rangle$ of base pairs. Computing whether a base pair B' can be generated from T and $\Phi_{i,j}$ via Definition 4.4 takes time polynomial in $\|\Phi_{i,j}\|$ and $\|T\|$.*

Proof. Immediate from Definition 4.4 and the fact that satisfiability of pure formulas is decidable in polynomial time (see, e.g., [1]). □

Lemma 5.2. k-PREDSAT *is in* NP.

Proof. Suppose (Φ, P) is the input instance. We define a non-deterministic algorithm as follows. For each predicate P_i in Φ we guess a set of up to N entries each comprising:

$$\langle B, j, \overrightarrow{B_1}, \ldots, \overrightarrow{B_m} \rangle$$

where B is a base pair and j is an integer such that rule $\Phi_{i,j}$ (below) defines P_i.

$$\Pi * y_1 \mapsto \mathbf{u_1} * \ldots * y_k \mapsto \mathbf{u_k} * P_{j_1}(\mathbf{x_1}) * \ldots * P_{j_m}(\mathbf{x_m}) \Rightarrow P_i\mathbf{x}$$

In addition, $\overrightarrow{B_\ell}$ is a pointer to a base pair for predicate P_{j_ℓ}. Each entry requires $O(\|\Phi\|^2 + \log|\Phi| + L \log N) = O(\|\Phi\|^3)$ time and space. Constructing all entries takes $O(|\Phi| N \|\Phi\|^3) = O(N\|\Phi\|^4)$ time and space.

We will now show that this set of sets of entries represents a possible state in the execution of the algorithm in Definition 4.4. To do this we need to ensure two things. Firstly, it must be possible to generate each base pair according to the information contained in its corresponding entry (this is clearly possible in polynomial time by Lemma 5.1). Secondly, that each such base pair is ultimately derivable from base cases, i.e., rules with no predicate occurrences. It is simple to show that if this is possible then the algorithm must generate all these base pairs at some point in its execution. To ensure this, we need only show that all maximal paths starting at a base pair and formed by following the pointers $\overrightarrow{B_\ell}$, are acyclic. This guarantees that they terminate in sink base pairs, i.e., ones where the generating rule involves no other base pairs ($m = 0$, above). This can be checked using standard graph-search algorithms in polynomial time in the size of the graph, i.e., in $N|\Phi|$. Finally, to show that a predicate in Φ is satisfiable, it suffices to have a base pair for that predicate, which can be trivially checked.

If certifying a pair takes time $O((N\|\Phi\|)^c)$ for some constant c, then certifying all pairs takes $O(|\Phi|N(N\|\Phi\|)^c) = O((N\|\Phi\|)^{c+1})$ time. Consequently, the total time for guessing and certifying will be $O((N\|\Phi\|)^{c+1})$ without loss of generality.

If the arity of any predicate in Φ is bounded by a constant then N also becomes a constant, and thus we can decide consistency in non-deterministic $O(\|\Phi\|^{c+1})$ time. □

Lemma 5.3. PREDSAT *is in* EXPTIME.

Proof. Each step of the algorithm scans all rules in Φ and for each one looks for a combination of base pairs that yields a new base pair. Thus it may have to go through N^L combinations of base pairs. In the worst case all rules have to be scanned, in up to $|\Phi|N^L$

time. If no new base pair can be generated for any rule 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 exactly one pair is generated in each step. Thus $|\Phi|N$ steps are required with total time $|\Phi|N|\Phi|N^L p(\|\Phi\|) = O(2^{\mathrm{poly}(\|\Phi\|)})$ where $p(\|\Phi\|)$ is the time taken to check a single combination of base pairs according to Lemma 5.1. □

Our lower bound results use standard facts about boolean circuits, with and without inputs. We will 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 $\bot$, even if it uses none; (c) there is one more kind of gate, that representing the NAND connective (denoted by $\uparrow$); (d) gates are numbered sequentially, so that if gate i has inputs l, r, then $i > l$ and $i > r$; (e) inputs precede $\top$ which precedes $\bot$ which precedes NAND gates in this order; (d) the output of the maximal gate is the output of the circuit. We begin by defining a mapping from circuits to sets of inductive rules, which will provide the basis for NP-hardness.

Definition 5.4. 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 the set of rules Φ_C as follows:

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

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

In the definition of Φ_C, x_{l_i} (resp. x_{r_i}) denote the left (right) input of gate i, where $i > k+2$. Clearly, $\|\Phi_C\| = O(\|C\|)$ and the maximum predicate arity is 3.

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

Definition 5.5. 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

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

$$s^{\mathbf{x}}_{\mathbf{B}}(x_i) =_{\text{def}} \mathrm{sval}(B_i) \text{ for } i = 1, \ldots, n$$

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

$$(\mathbf{B}^s_{\mathbf{x}})_i =_{\text{def}} \mathrm{bval}(s(x_i)) \text{ for } i = 1, \ldots, n$$

Now we can state the first hardness result of this section.

Theorem 5.6. *k-PREDSAT is NP-complete for $k \geq 3$.*

Proof. Membership in NP follows from Lemma 5.2. Hardness will follow by reducing from CIRCUITSAT via Definition 5.4. Formally, we show that there exists a boolean k-tuple $\mathbf{B}$ such that $C(\mathbf{B}) = \top$ if and only if $(\Phi_C, Q_\top) \in k$-PREDSAT. This will be achieved by showing that for any b there exists appropriate $\mathbf{B}$ such that $C(\mathbf{B}) = b$ if and only if $(\Phi_C, Q_b) \in k$-PREDSAT.

Left-to-right: suppose there exists $\mathbf{B}$ such that $C(\mathbf{B}) = b$. Extend the tuple $\mathbf{B}$ to the n-tuple $\mathbf{B}'$ such that for all $i = 1, \ldots, n$, B'_i is equal to the output of gate i. $\mathbf{B}'$ is clearly 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 $n \leq k+2$ then 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 follows from the definitions of predicate N and stack $\hat{s}$.

Right-to-left: suppose there is a stack s such that $s \models_{\Phi_C} Q_b$. We assume without loss of generality 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 two inputs $l_i, r_i < i$ for which we know that their values are equal to $\hat{B}_{l_i}, \hat{B}_{r_i}$ respectively. It is then a simple matter of verifying 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 $s \models_{\Phi_C} P(x_n) * T(x_n)$, thus $\hat{B}_n = \top$ and therefore $C(\mathbf{B}) = \top$. The other case where $b = \bot$ is similar. □

For EXPTIME hardness we will encode natural numbers as boolean tuples as well as formulas. The following definition formalises how we will do this.

Definition 5.7. We use $i^{\mathcal{B}}$ to denote the boolean n-tuple encoding an integer $i = 1, \ldots, 2^n$ as a binary number on $\mathcal{B}$. Note that, for notational 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. Formally,

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

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

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

The *circuit value problem* (CVP) has as instances circuits without inputs. A circuit C is a yes-instance of CVP if and only if $C() = \top$. The *succinct circuit value problem* (SCVP) has as instances input-free circuits that are generated by a pair of circuits in a way that is explained below. Again, an instance of SCVP is a yes-instance if and only if for the generated circuit C it is the case that $C() = \top$.

Formally, an instance $S_C = (L, R)$ of SCVP consists of two circuits, each of $n+n$ inputs. In the represented circuit C, for every $i, j = 1, \ldots, 2^n$ if $L(i^{\mathcal{B}}, j^{\mathcal{B}}) = \top$ (resp. $R(i^{\mathcal{B}}, j^{\mathcal{B}}) = \top$) then $i > j$, 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.

Definition 5.8. Let C be a k-input circuit and n gates. Then, define Φ^i_C as follows, where Ψ is as in Definition 5.4.

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

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, we have the following lemma.

Lemma 5.9. *For any circuit C on k-inputs and any $\mathbf{B} \in \mathcal{B}^k$, $C(\mathbf{B}) = \top$ if and only if $\exists s \models_{\Phi^r_C} P(\mathbf{x}) * \mathrm{bool}_{\mathbf{B}}(\mathbf{x})$.*

Proof. Analogous to the proof of Theorem 5.6. □

Lemma 5.10. *For any k-input circuit C and stacks s, s', if it is the case that $\mathbf{B}^s_{\mathbf{x}} = \mathbf{B}^{s'}_{\mathbf{x}}$ 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, $s(x_i) = nil$ if and only if $s'(x_i) = nil$. The result then follows by verifying that the satisfaction of predicates T, F, N by a stack s depends only on whether their arguments evaluate to nil or not in the stack s. □

We can now define a mapping from succinctly represented circuits to sets of inductive rules.

Definition 5.11. Let $S_C = (L, R)$ be an instance of SCVP. Let Φ^i_L, Φ^i_R be as in Definition 5.8 (we assume names P_L, P_R for the corresponding P predicate). Define rules U1, U2, U3 as follows.

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

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

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

Then define the following sets of rules.

$$X =_{\text{def}} \{\text{U1}, \text{U2}, \text{U3}\} \cup \Phi^i_L \cup \Phi^i_R$$

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

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

Using Definition 5.11, we can state our EXPTIME-hardness result.

Theorem 5.12. PREDSAT *is* EXPTIME*-complete.*

Proof. Membership follows from Lemma 5.3. Hardness follows by reducing from SCVP (which is EXPTIME-complete [21]) through Definition 5.11. This follows from the stronger fact that for all gates $i = 1, \ldots, 2^n$ and for any boolean b, the output of gate i is b if and only if there is stack s such that $s \models_{\Phi_{S_C}} \mathrm{num}_i(\mathbf{x}) * Q_b(\mathbf{x})$. We use induction on i.

Left-to-right: Let $i = 1$. Gate 1 is the constant $\top$ by assumption thus its output is also $b = \top$. Let $\mathbf{B} =_{\text{def}} i^{\mathcal{B}}$. Define $\hat{s}$ as

$$\hat{s} =_{\text{def}} s^{\mathbf{x}}_{\mathbf{B}}[v \mapsto \mathrm{sval}(\top)].$$

Clearly as $\hat{s} \models_X U(\mathbf{x}, v)$ through (U1) and $\hat{s} \models_{\Phi_{S_C}} T(v)$ 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.

Let $i > 2$ and suppose the output of gate i is b. That gate must be a NAND gate thus there exist $l, r < i$ 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 such that $s_l \models_{\Phi_{S_C}} \mathrm{num}_l(\mathbf{l}) * Q_c(\mathbf{l})$ and $s_r \models_{\Phi_{S_C}} \mathrm{num}_r(\mathbf{r}) * Q_d(\mathbf{r})$. In particular, regardless of which values c, d have, $s_l \models_{\Phi_{S_C}} U(\mathbf{l}, w)$ and $s_r \models_{\Phi_{S_C}} U(\mathbf{r}, z)$ (we set $s_l(w) = s_l(v)$ and $s_r(z) = s_r(v)$ without loss of generality). By Lemma 5.9, we also conclude that there are stacks s'_l, s'_r such that $s'_l \models_{\Phi^r_L} P_L(\mathbf{x}, \mathbf{l}) * \mathrm{num}_i(\mathbf{x}) * \mathrm{num}_l(\mathbf{l})$ and $s'_r \models_{\Phi^r_R} P_R(\mathbf{x}, \mathbf{r}) * \mathrm{num}_i(\mathbf{x}) * \mathrm{num}_r(\mathbf{r})$, where Lemma 5.10 allows us to 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 $\hat{s} \models_{\Phi_{S_C}} \mathrm{num}_i(\mathbf{x}) * Q_b(\mathbf{x})$.

Right-to-left: Suppose now that there is a stack s 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 $i = 1$ or $i = 2$ then gate i is a constant and the result follows trivially from rules (U1) or (U2).

Let $i > 2$. Then, gate i is a NAND gate, thus 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 $l, r = 1, \ldots, 2^n$ be the unique integeres such that $l^{\mathcal{B}} = \mathbf{B}^s_{\mathbf{l}}$ and $r^{\mathcal{B}} = \mathbf{B}^s_{\mathbf{r}}$. By Lemma 5.9 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. □

Exponential run-time can be exhibited easily through counting.

Proposition 5.13. *There exists a family of predicates Φ_n of size $O(n)$ such that the algorithm runs in $\Omega(2^n)$ time and space.*

Proof. It should be clear from the above results that it is possible to render circuits as predicates in linear space. In particular, the successor relation over two n-bit numbers is trivial to encode as a predicate $succ(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}, \mathbf{y}$ are two n-variable tuples. Consider then the following set of predicates Φ_n.

$$\mathrm{num}_1(\mathbf{y}) \Rightarrow Q(\mathbf{y})$$

$$succ(\mathbf{x}, \mathbf{y}) * Q(\mathbf{x}) \Rightarrow Q(\mathbf{y})$$

$$\mathrm{num}_{(2^n)}(\mathbf{x}) * Q(\mathbf{x}) \Rightarrow P$$

The set Φ_n has size linear in n, including all auxiliary predicates such as $succ$.

Given Φ_n, the algorithm will start with the base case for Q and effectively count up to $2^n - 1$, creating a base pair encoding each number in between. Note that this will happen irrespective of which strategy is used to select predicates or base pairs for instantiation. Clearly, the algorithm will take $\Omega(2^n)$ time and space before it finds a fixed point. □

It is notable that Definitions 5.4 and 5.11 describe predicates that are pure, in that no points-to subformulas ($x \mapsto \mathbf{t}$) are used. This fact affirms the intuition obtained from the algorithm, i.e., that heap allocation can be treated effectively (with respect to consistency) through computations on pure formulas, and thus that it does not add to the complexity of the pure problem.

6. Implementation and experiments

We implemented the decision procedure for satisfiability of inductive definitions in separation logic in CYCLIST [7], an open-source framework for constructing cyclic theorem provers with support for logics with inductive predicates. Our implementation is a straightforward rendition of Definition 4.4 in about 1000 lines of OCaml code. In the rest of this section we detail our efforts to evaluate the performance of the algorithm in various contexts.

6.1 Benchmarks on automatically abduced predicates

The first of our experiments concerns the abduction of inductive definitions in separation logic. The CABER tool described in [6]

(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

Table 1. Each row reports the number of test cases, out of 100 randomly generated instances for each parameter combination, that were solved by our tool.

takes a program as input and attempts to invent an inductively defined precondition such that the program, when started from a state that satisfies this precondition, is guaranteed to run without faulting (i.e., it is *memory-safe*). It returns as output a set of inductive rules, but does not guarantee that the defined predicates are satisfiable. Of course, ensuring that the abduced precondition is indeed satisfiable is important because an unsatisfiable precondition, though practically useless, nevertheless trivially satisfies the requirements of abduction: any program is memory-safe under this precondition for the simple reason that no program state satisfies the precondition.

We modified this abductive prover so that whenever it finds a candidate set of inductive definitions, it records it as a test case and then fails the search, triggering back-tracking and, thus, continuing to generate candidate definitions. Twenty-nine test programs were used, producing 45 945 syntactically unique inductive rule sets. The number of predicates ranged between one and 37, with the majority (94%) having 20 predicates, and with sets with 19 predicates being the most numerous (15%). Predicates had up to 11 parameters and up to two recursive invocations. Most sets were satisfiable (83%) and no test took more than 50 ms. We conjecture that the strong performance of our algorithm in these tests results from the relatively simple recursive structure that the predicates produced by the prover have.

6.2 Benchmarks on predicates from the literature

We have collected standard predicates for e.g. singly- or doubly-linked list segments, (possibly circular) lists, tree segments and trees from the literature. This collection consists of in total 17 predicates defined by 36 rules. With a runtime of 4 ms to prove satisfiability of this set of definitions, also here our decision procedure performs extremely well. This is rather expected, since all predicates in this test set are quite easily seen to be satisfiable.

6.3 Synthetic benchmarks

To experimentally evaluate the scalability of our procedure, we designed a random distribution from which to draw hard synthetically generated test cases. The distribution is determined by the following parameters:

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

For each instance, a total of $Rules = Preds \times Cases$ inductive definitions are independently generated, where all predicates have the same $Arity$. On each rule body, the exact number of literals of each of the four available kinds is drawn from a Poisson distribution where the parameter λ is set to, respectively, Eqs, $Neqs$, $Points$, or $Recs$. This produces a mix of short and long rule bodies with a specified average length. Furthermore, to produce 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.

The terms occurring on each literal are randomly drawn from a set containing nil and the specified number of variables $Vars$. To avoid trivially unsatisfiable rule bodies, the two terms in an equality or disequality are required to be distinct, while all terms allocated by points-to literals are required to be non-nil and distinct from each other. Moreover, when randomly choosing a term, with probability $p = 1/(Points + Recurs)$ the choice is limited to arguments in the head of the rule. This has the effect, roughly, of making sure that on average one of the arguments of the predicate being defined is allocated—either directly or indirectly by recursion—in the body of the rule.

Table 1 reports statistics gathered when solving instances drawn from the above described distribution. Each row collects the results from 100 instances randomly generated with different parameter values. The two middle columns report, for each parameter combination, the number of test cases that were solved in, respectively, not more than 1 second and not more than half a minute; while the last column reports the total number of satisfiable instances, i.e. when all predicates in the rule set are satisfiable, that were found before the half a minute timeout.

Although most cases are fairly easy to solve, a few hard instances consistently show up on all parameter settings. For example, the tool takes one second per instance or less to decide 89 of the 100 test cases generated with $Vars = 50$, $Rules = 3{\times}3$, $Arity = 3$, and all Eqs, $Neqs$, $Points$, and $Recs$ set to 2 (that is the bold line repeated on all four tables). Given half a minute per instance 9 more test cases can be solved, while 2 very hard instances remain unsolved after this hard timeout. Each of the four tables show, respectively, how these statistics change as one of the parameters changes while all the others remain fixed. As expected, more variables result in more satisfiable instances (a), and systems with more rules (b) or predicates with higher arity (c) are more difficult to solve. Rules with more recursive predicate calls (d) also have a slight increase in difficulty but not as much, this can explained by the quick drop on satisfiable instances found: if the number of variables remains fixed, adding more predicates to the body of a rule will tend to make it unsatisfiable and thus easier to decide.

6.4 Exponential-time benchmarks

We also report on a special class of hand-crafted synthetic benchmarks, namely the family Φ_n of predicates of size $O(n)$ given in Proposition 5.13 to show that the algorithm can have runtime exponential in n. Here we used a binary adder-based implementation for the successor relation. Table 2 reports on the corresponding runtimes. As expected, the runtime indeed deteriorates dramatically with growing values for n in this artificial family of recursive specifications.

n	Runtime
1	< 0.01 s
2	0.05 s
3	0.07 s
4	0.84 s
5	28.94 s
6	15 m 49.25 s
7	> 40 m

Table 2. Runtimes of our tool on the predicate family Φ_n from Proposition 5.13

7. Related work

We identify three main categories of related work: (i) work on satisfiability for fragments of separation logic; (ii) work on satisfiability for other logical settings featuring inductive definitions; and finally (iii) automated verification tools for separation logic with support for general inductive predicates, which might benefit from our satisfiability decision procedure.

7.1 Satisfiability in separation logic with inductive predicates

Some interesting progress has been made recently on satisfiability checking for certain fragments of separation logic with user-defined inductive predicates. In [16], Iosif et al. consider a restriction of our fragment of separation logic (see Definition 3.1) which only allows structures with *bounded treewidth*. They prove decidability for satisfiability and entailment of inductive definitions in this fragment via a reduction to monadic second-order logic over graphs. However, the restriction to structures of bounded treewidth rules out many natural inductive predicates; in particular, structures with dangling data pointers, such as list or tree segments with extra arbitrary data fields. In the present paper, we only consider satisfiability, not entailment, 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 fragment of separation logic given by symbolic heaps with a single fixed inductive predicate lseg, denoting linked list segments. Berdine et al. [1] provided the first decidability result for satisfiability and entailment of this fragment. Cook et al. [11] improved on this result by giving a graph-based decision procedure which operates in polynomial time, implemented in the tool SELOGER [14]. Recently, Piskac et al. [22] presented another decision procedure based on translating entailments to an intermediate logic of "graph reachability and stratified sets" which, given suitable axioms, is then decided by an SMT solver. The technique works as well for slightly more general structures, such as sorted list segments and doubly linked lists. Finally, Navarro Pérez and Rybalchenko [19] provided an SMT encoding for satisfiability and entailment in another extension of the fragment, allowing the pure part of the symbolic heaps to contain SMT formulas for arbitrary theories, rather than simple (dis)equalities.

7.2 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 satisfiability of Datalog queries was shown to be decidable by Shmueli [24]. More recently, Hoder et al. [15] 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 (which is to say that 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. [4] advocate the use of such clauses as an interchange format for software model checking tools, while Grebenshchikov et al. [13] describe an abstraction-based procedure for finding models and checking the satisfiability of Horn-like clauses in the context of program analysis.

An interesting line of research for future work would be to extend the current tools and techniques for first-order Horn clauses to the situation in which separation logic operators are permitted in the bodies of rules.

7.3 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 6) the theorem prover CYCLIST [7], which treats separation logic with user-defined inductive predicates, and the related abductive prover CABER [6], which automatically infers inductive predicate definitions as safety/termination preconditions for `while` programs. Another such tool is THOR [18, 17], which allows for proofs of memory safety and for generating sound arithmetic abstractions of heap programs (with respect to safety and termination). In THOR the specification must be entered manually, including the definitions of any inductive predicates; the consistency of such definitions is the responsibility of the user. Other automated separation logic tools allowing the user to define their own inductive predicates include: HIP/SLEEK [10], a combined theorem prover and verification system for a simple C-like language; the JSTAR [12] prover for Java programs, which requires that the user provides rules for, as well as the definitions of, any inductive predicates; and the automated shape analysis described in [9], which infers inductive predicate definitions based on "structural invariant checkers", which are essentially user-provided inductive definitions. We believe that our decision procedure could be of assistance in such settings, by warning the user about unintentionally inconsistent predicate definitions.

8. Conclusion and future work

The question of whether or not satisfiability of the symbolic heap fragment of separation logic with general inductive predicates is decidable has stood open for some time. Following the recent achievement of a partial positive answer to this question in [16], in this paper we resolve this question affirmatively for the general case.

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. In addition, we provide complexity results stating that the general case of the satisfiability problem is EXPTIME-complete, and its restriction to predicates with at most $k \geq 3$ arguments is still NP-complete. Despite these high complexities, our experiments indicate that, for realistic predicate definitions arising in practice, our prototype implementation is already able to solve the decision problem in milliseconds.

Our solution to the satisfiability decision problem for general inductive predicates in separation logic opens up possible applications in symbolic execution and verification of heap-based programs with *arbitrary* data structure shapes. For example, problems

such as deciding whether a given symbolic heap describes any concrete memory state or deciding the consistency of abduced preconditions in separation logic [6] have now come within reach of mechanical solutions.

Taking a different direction, one could investigate whether our techniques for deciding satisfiability of inductive definitions extends to more general variants of separation logic, where general inductive predicates are still allowed, but formulas are less restricted. Possible candidates for such extensions include: higher-order separation logic [3]; the fragment in which formulas may contain pure assertions beyond (dis)equalities as in [19]; and fragments in which $*$ and $\wedge$ may be nested.

References

[1] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *Proc. FSTTCS'04*, volume 3328 of *LNCS*, pages 97–109, 2004.

[2] Josh Berdine, Byron Cook, and Samin Ishtiaq. Slayer: Memory safety for systems-level code. In *Proc. CAV'11*, volume 6806 of *LNCS*, pages 178–183, 2011.

[3] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM TOPLAS*, 29(5), 2007.

[4] Nikolaj Bjørner, Kenneth McMillan, and Andrey Rybalchenko. Program verification as satisfiability modulo theories. In *Proc. SMT'12*, pages 3–11, 2012.

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

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

[7] James Brotherston, Nikos Gorogiannis, and Rasmus Lerchedahl Petersen. A generic cyclic theorem prover. In *Proc. APLAS'12*, volume 7705 of *LNCS*, pages 350–367, 2012.

[8] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6):26, 2011.

[9] Bor-Yuh Evan Chang, Xavier Rival, and George Necula. Shape analysis with structural invariant checkers. In *Proc. SAS'07*, volume 4634 of *LNCS*, pages 384–401, 2007.

[10] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao 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.

[11] Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. Tractable reasoning in a fragment of separation logic. In *Proc. CONCUR'11*, volume 6901 of *LNCS*, pages 235–249, 2011.

[12] Dino Distefano and Matthew J. Parkinson. jStar: towards practical verification for Java. In *Proc. OOPSLA'08*, pages 213–226, 2008.

[13] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *Proc. PLDI'12*, pages 405–416, 2012.

[14] Christoph Haase, Samin Ishtiaq, Joël Ouaknine, and Matthew J. Parkinson. Seloger: A tool for graph-based reasoning in separation logic. In *Proc. CAV'13*, LNCS, 2013. To appear.

[15] Krystof Hoder, Nikolaj Bjørner, and Leonardo de Moura. μZ– an efficient engine for fixed points with constraints. In *Proc. CAV'11*, volume 6806 of *LNCS*, pages 457–462, 2011.

[16] Radu Iosif, Adam Rogalewicz, and Jiri Simacek. The tree width of separation logic with recursive definitions. In *Proc. CADE'13*, volume 7898 of *LNAI*, pages 21–38, 2013.

[17] Stephen Magill. *Instrumentation Analysis: An Automated Method for Producing Numeric Abstractions of Heap-Manipulating Programs*. PhD thesis, CMU Pittsburgh, PA, USA, 2010.

[18] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proc. POPL'10*, pages 211–222, 2010.

[19] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic modulo theories. *CoRR*, abs/1303.2489, 2013.

[20] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proc. CSL'01*, volume 2142 of *LNCS*, pages 1–19, 2001.

[21] Christos H. Papadimitriou and Mihalis Yannakakis. A note on succinct representations of graphs. *Inf. Control*, 71(3):181–185, December 1986.

[22] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using SMT. In *Proc. CAV'13*, volume 8044 of *LNCS*, 2013. To appear.

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

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

[25] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *Proc. CAV'08*, volume 5123 of *LNCS*, pages 385–398, 2008.