A Decidable Fragment in Separation Logic with Inductive Predicates and Arithmetic (Technical Report)

Quang Loc Le[1], Makoto Tatsuta[2], Jun Sun[3], and Wei-Ngan Chin[4]

[1] School of Computing, Teesside University, UK
[2] National Institute of Informatics / Sokendai, Tokyo, Japan
[3] Singapore University of Technology and Design, Singapore
[4] National University of Singapore, Singapore

Abstract. We consider the satisfiability problem for a fragment of separation logic including inductive predicates with shape and arithmetic properties. We show that the fragment is decidable if the arithmetic properties can be represented as *semilinear* sets. Our decision procedure is based on a novel algorithm to infer a finite representation for each inductive predicate which precisely characterises its satisfiability. Our analysis shows that the proposed algorithm runs in exponential time in the worst case. We have implemented our decision procedure and integrated it into an existing verification system. Our experiment on benchmarks shows that our procedure helps to verify the benchmarks effectively.

Keywords: Satisfiability Solving · Decidability · Separation Logic · Inductive Predicates

1 Introduction

Separation logic [16,29] is a well-established assertion language designed for reasoning about heap-manipulating programs. Combined with inductive predicates, separation logic has been shown to capture semantics of loops and recursive procedures naturally and succinctly. A decision procedure for satisfiability of separation logic with inductive predicates could be useful for multiple analysis problems associated with heap-manipulating programs, e.g., compositional verification [9,28,21], shape analysis [17], termination analysis [7] as well as to uncover reachability in bug finding tools [19]. It has been shown that the satisfiability of the fragment of separation logic which does not include inductive (user-defined) predicates is decidable [8,25,24,19]. The main challenge on satisfiability checking of separation logic with inductive predicates is that it often requires reasoning about infinite heaps as well as infinite integer domain. Indeed, the problem in the full fragment of inductive predicates with shape and arithmetic properties is shown to be undecidable [31]. One research goal is thus to identify decidable yet expressive fragment of the logic, based on which we can have precise and always-terminating reasoning over heap-manipulating programs.

One way to show that a fragment of separation logic with inductive predicates is decidable is to infer, for each inductive predicate, a finite representation without any inductive predicates which precisely characterizes its satisfiability. For example, the authors

in [1] showed that inductive predicates on linked lists can be precisely characterised by models of length zero or two and thus concludes that the fragment of separation logic with inductive predicates on linked lists only is decidable. Later, Brotherston *et. al.* proposed SLSAT [6], a decision procedure to compute for every arbitrary heap-only inductive predicate a finite (disjunctive) set of base formulas which exactly characterises its satisfiability, and consequently showed that the fragment of separation logic with heap-only inductive predicates is decidable. Finally, the work in [31] extended SLSAT to show that a fragment of separation logic with inductive predicates and arithmetic properties under several restrictions is decidable. In particular, their fragment only allows inductive predicates satisfying the following conditions: for each inductive predicate, its heap part has two disjuncts and the arithmetic part is restricted in DPI predicates.

In this work, we present a decidable fragment of separation logic including inductive predicates with shape and arithmetic properties, which is more expressive than all fragments which have been shown to be decidable previously. The decidability is shown through a novel algorithm which computes for each inductive predicate a base formula (i.e. one without inductive predicates) which exactly characterizes its satisfiability. The idea is to compute for each heap-only inductive predicate a non-recursive base formula regardless of the infinite domains. In the case that the inductive predicate includes shape and arithmetic properties, if the arithmetical properties can be precisely computed in the form of arithmetic closures, we derive a combination of the base formula and the arithmetic closures which precisely characterises satisfiability for the inductive predicate.

In particular, we show how to derive a disjunctive base formula for each inductive predicate based on *flat formulas*, which are designed to capture the notion of a (infinite) set of formulas which can be represented by the same base formula (allocated memory, (dis)equalities and arithmetic closures). First, we describe a novel algorithm to derive for each inductive predicate a cyclic unfolding tree prior to flattening the tree into a disjunctive set of *regular* formulas. Every regular formula in this set has the same base pair of the allocated memory and (dis)equalities over a set of free variables (similar to [6]). Secondly, we define a decidable fragment where every regular formula derived for inductive predicates is flattable i.e., its arithmetic part is a conjunction of periodic constraints and the closure of the union of these conjunctions can be represented by some semilinear sets and thus is Presburger-definable (similar to [5,31]). As a result, our algorithm derives for each inductive predicate a disjunctive set of flat formulas, and then a disjunctive set of base formulas.

Contributions We make the following technical contributions.

- Firstly, we present a novel algorithm to generate cyclic unfolding trees for inductive predicates with shape and arithmetic properties. Our complexity analysis shows that the proposed algorithm runs in exponential time in the worst case.
- Secondly, based on the algorithm, we present a decision procedure for satisfiability checking of the fragment of separation logic with inductive predicates where arithmetic properties can be represented as semilinear sets.
- Thirdly, we have implemented our algorithm and applied it to verify several benchmark programs. In our implementation, we generate under-/over-approximated bases for those inductive predicates beyond the decidable fragment systematically.

Predicates	$Pred$	$::= \texttt{pred}\ \texttt{P}_1(\bar{v}){\equiv}\Phi_1;\ \cdots;\texttt{pred}\ \texttt{P}_\texttt{n}(\bar{v}){\equiv}\Phi_n$
Formula	Φ	$::= \Delta \mid \Phi_1 \vee \Phi_2 \qquad \Delta ::= \exists\bar{v}{\cdot}\ (\kappa{\wedge}\alpha{\wedge}\phi)$
Spatial formula	κ	$::= \texttt{emp} \mid x{\mapsto}c(f_i{:}v_i) \mid \texttt{P}(\bar{v})^o_u \mid \kappa_1{*}\kappa_2$
Ptr (Dis)Equality	α	$::= \texttt{true} \mid \texttt{false} \mid v_1{=}v_2 \mid v{=}\texttt{null} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{null} \mid \alpha_1{\wedge}\alpha_2$
Presburger arith.	ϕ	$::= \texttt{true} \mid i \mid \exists v{\cdot}\ \phi \mid \neg\phi \mid \phi_1{\wedge}\phi_2 \mid \phi_1{\vee}\phi_2$
Linear arithmetic	i	$::= a_1{=}a_2 \mid a_1{\leq}a_2$
	a	$::= k^{\texttt{int}} \mid v \mid k^{\texttt{int}}{\times}a \mid a_1{+}a_2 \mid -a \mid max(a_1,a_2) \mid min(a_1,a_2)$

$\mathcal{P}{=}\{\texttt{P}_1, ..., \texttt{P}_\texttt{n}\}\quad c \in Node\quad f_i \in Fields\quad v,v_i,x,y \in Var\quad \bar{v} \equiv v_1,\ldots,v_n$

Fig. 1: Syntax.

Organization The rest of the paper is organized as follows. Sect 2 presents relevant definition. Sect 3 shows an overview of our approach through an example. We show how to compute bases of regular formulas in Sect 4 and subsequently compute regular formulas of inductive predicates in Sect 5. Sect 6 describes a decision procedure. Our implementation and evaluation are presented in Sect 7. Sect 8 reviews related work and lastly Sect 9 concludes. All missing proofs are presented in Appendix.

2 Preliminaries

We use $\bar{x}$ to denote a sequence of variables and x_i to denote its ith element. We write $\bar{x}^N$ and $\bar{x}^S$ to denote the sequence of integer variables and pointer variables in $\bar{x}$, resp.

Syntax A formula is defined by the syntax presented in Fig. 1. A symbolic heap Δ is an existentially quantified conjunction of some spatial formula κ, some pointer (dis)equality α and some formula in Presburger arithmetic ϕ. All free variables in Δ, denoted by function $FV(\Delta)$, are implicitly universally quantified at the outermost level. The spatial formula κ may be conjoined ($*$) by emp predicate, points-to predicates $x{\mapsto}c(f_i{:}v_i)$ and inductive predicate $\texttt{P}(\bar{v})^o_u$ where o and u are labels used for constructing unfolding trees in a breadth-first manner. While o captures the ordering number, u is the number of unfolding. We occasionally omit these numbers if there is no ambiguity. Whenever possible, we discard f_i of the points-to predicate and use its short form as $x{\mapsto}c(\bar{v})$. We often use π to denote a conjunction of α and ϕ formulas. Note that $v_1{\neq}v_2$ and $v{\neq}\texttt{null}$ are short forms for $\neg(v_1{=}v_2)$ and $\neg(v{=}\texttt{null})$ respectively. _ is used to denote a "don't care" term.

We write $\mathcal{P}$ to denote a set of n predicates in our system. Each inductive predicate is defined by a disjunction Φ using the key word pred. In each disjunct, we require that variables which are not formal parameters must be existentially quantified.

Example 1. We define an increasingly sorted list using the fragment above.

$$\texttt{pred}\ \texttt{sortll}(\texttt{root},n,mi) \equiv \texttt{root}{\mapsto}node2(mi,\texttt{null}) \wedge n{=}1$$
$$\vee\ \exists\ q,n_1,mi_1{\cdot}\ \texttt{root}{\mapsto}node2(mi,q) * \texttt{sortll}(q,n_1,mi_1) \wedge n{=}n_1{+}1 \wedge mi{\leq}mi_1;$$

where the data structure $node2$ is declared as: *data node2 { int val; node2 next;}*. In the sorted list $\texttt{sortll}(\texttt{root},n,mi)$, root is the pointer pointing to the head of the list, n is the length of the list and mi is the minimal value stored in the list.

We use $\Delta[t_1/t_2]$ for a substitution of all occurrences of t_2 in Δ to t_1. Note that we always apply the following normalization after predicate unfolding: $(\exists \bar{w}_1\cdot\ \kappa_1{\wedge}\pi_1) * (\exists \bar{w}_2\cdot\ \kappa_2{\wedge}\pi_2){\equiv}(\exists \bar{w}_1,\bar{v}_2\cdot\ \kappa_1{*}(\kappa_2\rho){\wedge}\pi_1{\wedge}(\pi_2\rho))$ where $\bar{v}_2$ is a vector of fresh variables and has the same length n as $\bar{w}_2$; and ρ is a substitution: $\rho{=}\circ\{[v_i/w_i]\mid\forall i\in\{1...n\}\}$.

Our proposal relies on the following definitions. $\mathtt{P}(\bar{v})$ is called (heap) observable if there is at least one free *pointer-typed* variable in $\bar{v}$. Otherwise, it is called *unobservable*. $v{\mapsto}c(\bar{t})$ is called (heap) observable if v is free. Otherwise, it is *unobservable*.

- **(base formula)** Φ is a *base* formula (or base for short) if it does not include any occurrences of inductive predicates. Otherwise, it is an inductive formula.
- **($\Delta^\exists$ formula)** Let $\Delta^\exists$ be a base formula and is of the form:

$$\Delta^\exists{\equiv}\exists\bar{w}{\cdot}x_1{\mapsto}c_1(\bar{v}_1){*}...{*}x_n{\mapsto}c_n(\bar{v}_n){\wedge}\alpha{\wedge}\phi$$

  $\Delta^\exists$ is a totally (existentially) quantified heap base formula if $x_i{\in}\bar{w}$ for all $i{\in}\{1,..,n\}$ and $FV(\alpha){\subseteq}\bar{w}$. We show that existentially quantified pointer-typed variables are not externally visible wrt. the satisfiability problem (Sec. 4). This is the fundamental for the transformation of an inductive predicate into an equi-satisfiable set of base formulas.
- **(regular formula)** Φ is a regular formula if it is of the form: $\Phi{\equiv}\Delta^b{*}\Upsilon^\exists$ where Δ^b is a base formula and $\Upsilon^\exists$ is a disjunctive (possibly infinite) set of $\Delta^\exists$ formulas. For example $\Delta^b{*}(\exists\bar{w}{\cdot}\mathtt{P_1}(\bar{v}_1){*}...{*}\mathtt{P_n}(\bar{v}_n))$, where $\bar{v}_i^S{\subseteq}\bar{w}\ \forall i{\in}\{1...n\}$, is a regular formula.
- **(flat formula)** Φ is a flat formula if it is a regular formula and is flattable, i.e. can be represented by a base formula.

We use Δ^b to denote a conjunctive base formula, Δ^{re} a regular formula and Δ^{flat} a flat formula. The following definition is critical for the computation of base formulas.

Definition 1 *The numeric projection $(\Phi)^N$ is defined inductively as follows.*

$$\begin{array}{lll} (\Delta_1 \vee \Delta_2)^N \equiv (\Delta_1)^N \vee (\Delta_2)^N & (\kappa_1{*}\kappa_2)^N & \equiv (\kappa_1)^N{\wedge}(\kappa_2)^N \\ (\exists\bar{x}\cdot\Delta)^N \quad \equiv \exists\bar{x}^N\cdot(\Delta)^N & (\mathtt{P}(\bar{v}))^N & \equiv \mathtt{P^N}(\bar{v}^N) \\ (\kappa{\wedge}\alpha{\wedge}\phi)^N \quad \equiv (\kappa)^N{\wedge}\phi & (x{\mapsto}c(\bar{v}))^N{\equiv}(\mathtt{emp})^N & \equiv \mathtt{true} \end{array}$$

For each inductive predicate $\mathtt{P}(\bar{t}){\equiv}\Phi$, we assume the inductive predicate symbols $\mathtt{P^N}$ and predicate $\mathtt{P^N}(\bar{t}^N)$ for its numeric projection satisfy $\mathtt{P^N}(\bar{t}^N){\equiv}\Phi^N$. The semantics of the numeric projection $\mathtt{P^N}(\bar{t}^N)$ is as follows. Let Υ_P^b be a (infinite) set base formulas derived from $\mathtt{P}(\bar{t})$. If all variables in $\bar{t}$ are pointer-typed, then $\mathtt{P^N}(\bar{t}^N){\equiv}\mathtt{true}$. Otherwise, $\mathtt{P^N}(\bar{t}^N){\equiv}\bigvee\{(\Delta^b)^N\mid\Delta^b\in\Upsilon_P^b\}$.

Example 2. The numeric definition $\mathtt{sortll}^N$ corresponding to the above increasingly sorted list `sortll` is defined as follows.

$$\begin{array}{l} \mathtt{pred\ sortll}^N(n{,}mi) \equiv n{=}1 \\ \quad \vee\ \exists\, n_1{,}mi_1{\cdot}\ \mathtt{sortll}^N(n_1, mi_1) \wedge n{=}n_1{+}1 \wedge mi{\leq}mi_1; \end{array}$$

$$
\begin{array}{lll}
s,h \models \mathtt{emp} & \mathtt{iff} & h{=}\emptyset \\
s,h \models v{\mapsto}c(f_i:v_i) & \mathtt{iff} & l{=}s(v), \mathrm{dom}(h){=}\{l \rightarrow r\} \text{ and } r(c,f_i){=}s(v_i) \\
s,h \models \mathtt{P}(\bar{t}) & \mathtt{iff} & s,h \models \mathtt{P}(\bar{t})^m \text{ for some } m{\geq}0 \\
s,h \models \mathtt{P}(\bar{t})^{k+1} & \mathtt{iff} & s,h \models \Delta[\mathtt{P}(\bar{t})^k/P(\bar{t})] \text{ for some definition branch } \Delta \text{ of } \mathtt{P} \\
s,h \models \mathtt{P}(\bar{t})^0 & \mathtt{iff} & \text{never} \\
s,h \models \kappa_1 * \kappa_2 & \mathtt{iff} & \exists h_1,h_2{\cdot}\ h_1\#h_2 \text{ and } h{=}h_1{\cdot}h_2 \text{ and} \\
 & & s,h_1 \models \kappa_1 \text{ and } s,h_2 \models \kappa_2 \\
s,h \models \mathtt{true} & \mathtt{iff} & \text{always} \qquad s,h \models \mathtt{false} \ \ \mathtt{iff} \ \ \text{never} \\
s,h \models \exists v_1,...,v_n{\cdot}(\kappa{\wedge}\pi) & \mathtt{iff} & \exists \alpha_1...\alpha_n \cdot s(v_1{\mapsto}\alpha_1{*}...{*}v_n{\mapsto}\alpha_n), h \models \kappa \\
 & & \text{and } s(v_1{\mapsto}\alpha_1{*}...{*}v_n{\mapsto}\alpha_n) \models \pi \\
s,h \models \Phi_1 \vee \Phi_2 & \mathtt{iff} & s,h \models \Phi_1 \text{ or } s,h \models \Phi_2
\end{array}
$$

Fig. 2: Semantics.

Semantics Concrete heap models assume a fixed finite collection *Node*, a fixed finite collection *Fields*, a disjoint set *Loc* of locations (heap addresses), a set of non-address values *Val*, such that $\mathtt{null}{\in}Val$ and $Val \cap Loc = \emptyset$. Further, we define:

$$
\begin{array}{l}
Heaps \stackrel{\text{def}}{=} Loc {\rightharpoonup}_{fin} (Node \rightarrow Fields \rightarrow Val \cup Loc) \\
Stacks \stackrel{\text{def}}{=} Var \rightarrow Val \cup Loc
\end{array}
$$

The semantics is given by a forcing relation: $s{,}h \models \Phi$ that forces the stack s and heap h to satisfy the constraint Φ where $h \in Heaps$, $s \in Stacks$, and Φ is a formula.

The semantics is presented in Fig. 2. $dom(f)$ is the domain of function f; $h_1\#h_2$ denotes that heaps h_1 and h_2 are disjoint, i.e., $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$; and $h_1{\cdot}h_2$ denotes the union of two disjoint heaps. Semantics of pure formulas depend on stack valuations. It is straightforward and omitted for simplicity.

3 Overview and Illustration

In this section, we illustrate how our decision procedure works through checking the satisfiability of the following inductive predicate over the data structure $node$ which is declared as: $data$ *node* { *node left; node right;*}.

$$
\begin{array}{l}
\mathtt{pred}\ \mathtt{Q}(x{,}y{,}n) \equiv \exists\ y_1{\cdot}x{\mapsto}node(\mathtt{null}{,}y_1){\wedge}y{=}\mathtt{null}{\wedge}x{\neq}\mathtt{null}{\wedge}n{=}1 \\
\quad \vee\ \exists\ x_1{,}y_1{,}n_1{\cdot}y{\mapsto}node(x_1{,}y_1) * \mathtt{Q}(x,y_1,n_1)^{0}_{_}{\wedge}y{\neq}\mathtt{null}{\wedge}n{=}n_1{+}2;
\end{array}
$$

First, we infer a disjunctive set of base formulas for the predicate Q which precisely characterizes Q's satisfiability. After that, we check satisfiability of each disjunct in the set. If one of the disjuncts is satisfied, so is Q. We remark that as the base formulas do not contain any occurrences of inductive predicates, their satisfiability is decidable [25,19]. We generate the base formulas for each inductive predicate by: (i) constructing a cyclic unfolding tree and (ii) extracting base formulas from the leaf nodes in the tree.

Constructing Cyclic Unfolding Tree We construct the cyclic unfolding tree for inductive predicate `Q` as shown in Fig. 3. In an unfolding tree, a node `v` is a conjunctive formula. An edge from $\mathtt{v}_1$ to $\mathtt{v}_2$ where $\mathtt{v}_2$ is a child of $\mathtt{v}_1$ is obtained by unfolding $\mathtt{v}_1$, i.e., substituting an occurrence of an inductive predicate in $\mathtt{v}_1$ with one disjunct in the predicate's definition (after proper actual/formal parameter substitutions). For instance, in Fig. 3, the root of the tree is $\Delta_2{\equiv}\mathtt{Q}(x{,}y{,}n)^0_0$. We remark that the ordering number and unfolding number of the root are initially set to 0. The root has two children, Δ_{21} and Δ_{22}, which are obtained by unfolding the occurrence of `Q` with its two branches.

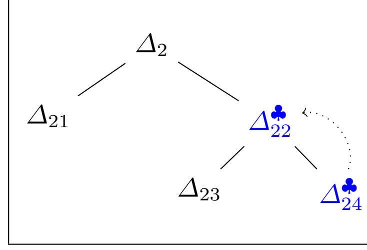


Fig. 3: Cyclic Unfolding Tree $\mathcal{T}^{Q}_{2}$.

$$\Delta_{21}{\equiv}\exists\, y_1{\cdot}x{\mapsto}node(\mathtt{null}, y_1) \wedge y{=}\mathtt{null}{\wedge}x{\neq}\mathtt{null}{\wedge}n{=}1$$
$$\Delta_{22}{\equiv}\exists\, x_1{,}y_1{,}n_1{\cdot}y{\mapsto}node(x_1{,}y_1) * \mathtt{Q}(x, y_1, n_1)^0_1{\wedge}y{\neq}\mathtt{null}{\wedge}n{=}n_1{+}2$$

In turn, Δ_{22} has two children, Δ_{23} and Δ_{24}, which are obtained by unfolding the occurrence of `Q` again.

$$\begin{aligned}\Delta_{23}{\equiv}&\exists\, x_1{,}y_1{,}n_1{,}y_2{\cdot}y{\mapsto}node(x_1{,}y_1) * x{\mapsto}node(\mathtt{null}, y_2) \wedge \\ &y_1{=}\mathtt{null}{\wedge}x{\neq}\mathtt{null}{\wedge}n_1{=}1{\wedge}y{\neq}\mathtt{null}{\wedge}n{=}n_1{+}2\\ \Delta_{24}{\equiv}&\exists\, x_1{,}y_1{,}n_1{,}x_2{,}y_2{,}n_2{\cdot}y{\mapsto}node(x_1{,}y_1){*}y_1{\mapsto}node(x_2{,}y_2){*}\mathtt{Q}(x, y_2, n_2)^0_2 \wedge \\ &y_1{\neq}\mathtt{null}{\wedge}n_1{=}n_2{+}2{\wedge}y{\neq}\mathtt{null}{\wedge}n{=}n_1{+}2\end{aligned}$$

We remark that unfolding numbers annotated for occurrences of recursive predicates (e.g., $\mathtt{Q}(x, y_1, n_1)^0_1$ in Δ_{22} and $\mathtt{Q}(x, y_2, n_2)^0_2$ in Δ_{24}) are increased by one after each unfolding.

A leaf in the unfolding tree is either a base formula (e.g., Δ_{21} and Δ_{23}), or one whose all occurrences of inductive predicates are unobservable, or one which is linked back to an interior node (e.g., Δ_{24}). Intuitively, a leaf node v is linked back to an interior node v' only if v is subsumed (wrt. the satisfiability problem) by v' in terms of the constraint on the heap. These back-links generate (virtual) cycles in the tree. A leaf is marked either closed or open. It is marked closed if it is either unsatisfiable or is linked back to some interior node. Otherwise, it is marked open. For instance, Δ_{24} is linked back to Δ_{22} and thus marked closed. These two nodes are labeled with the fresh symbol ♣ in Fig. 3. They are linked as they have (i) the same *observable* points-to predicate $y{\mapsto}node(_{,}_)$, (ii) the same *observable* occurrence of inductive predicate $\mathtt{Q}(x, _, _)$ and (iii) the same disequalities over free variables (i.e., $y \neq \mathtt{null}$).

Each path ending with a leaf node which is not involved in any back-link represents (a way to derive) a formula which can be obtained by unfolding the inductive predicates according to the edges in the path. A cycle in the tree thus represents an infinite set of formulas, since we can construct infinitely many paths by iterating through the cycle an unbounded number of times. For instance, in Fig. 3, we can obtain a different formula following the cycle from Δ_{22} to Δ_{24} and back to Δ_{22} for a different number of times and then following the edge from Δ_{22} to Δ_{23}. We show that all formulas obtained by iterating through the same cycle a different number of times have the same *spatial*

base. Furthermore, if the closure of the arithmetic part of these formulas is Presburger-definable, we can construct one formula to represent this infinite set of formulas.

Flattening Cyclic Unfolding Tree After constructing the tree, we derive the `base` for the inductive predicates, e.g. `Q` in this example. To do that, we flatten the tree iteratively until there is no cycle left. To flatten the tree iteratively, we keep flattening the minimal cyclic sub-trees, i.e. the sub-trees without nested cycles, in a bottom-up manner. For instance, in Fig. 3 the sub-tree in which Δ_{22} is the root is a minimal cyclic sub-tree. In principle, we can derive an infinite number of base formulae, each of which corresponds to the formula constructed by iterating the cyclic a different number of times. For instance, the following is the disjunctive set of the formulas obtained by following the cycle zero or more times (and then visiting Δ_{23}).

$$\begin{aligned}\Delta_{23}^{flat} \equiv \quad & \exists\, x_1,y_1,n_1,y_2{\cdot}(y{\mapsto}node(x_1,y_1){*}x{\mapsto}node(\texttt{null},y_2){\wedge}x{\neq}\texttt{null}\wedge \\ & \quad y{\neq}\texttt{null}{\wedge}n{=}n_1{+}1) \wedge (y_1{=}\texttt{null}{\wedge}n_1{=}1) \\ & \vee \exists\, x_1,y_1,n_1,x_2,y_2,n_2,y_3{\cdot}(y{\mapsto}node(x_1,y_1){*}x{\mapsto}node(\texttt{null},y_3){\wedge}x{\neq}\texttt{null}\wedge \\ & \quad y{\neq}\texttt{null}{\wedge}n{=}n_1{+}1) * (y_1{\mapsto}node(x_2,y_2){*}{\wedge}y_2{=}\texttt{null}{\wedge}n_1{=}n_2{+}2 \\ & \quad n_2{=}1) \\ & \vee \ldots\end{aligned}$$

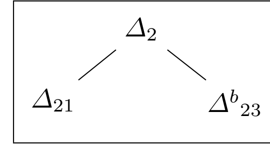


Fig. 4: Flattened Tree.

Notice that each iteration of this cycle results in a formula which conjuncts Δ_{23} with unobservable heaps (e.g., $y_1{\mapsto}node(_,y_2){\wedge}y_2{=}\texttt{null}$ where y_1, y_2 are existentially quantified variables) and a constraint which requires that the third parameter of `Q` is increased by two. We refer to Δ_{23}^{flat} as a flat formula. One of our main contribution in this work is to show that all formulae in the set have the same base. In particular, we state that a quantified heap base formula $\Delta^{\exists}{}_i$ is equi-satisfiable to its numeric projection, i.e. $(\Delta^{\exists}{}_i)^N$. As a result, a flat formula is equi-satisfiable to a conjunction of a base formula (i.e., Δ_{23}) and the set of the numeric projections. Furthermore, in the proposed decidable fragment, closure of this numeric set is Presburger-definable. In this example, this numeric set can be represented by the arithmetic predicate: $\texttt{P}_{\texttt{cyc}}(n_1){\equiv}n_1{=}1 \vee \exists n_2{\cdot}n_1{=}n_2{+}2{\wedge}\texttt{P}_{\texttt{cyc}}(n_2)$. Following [31], we can show that this predicate is equivalent to the following Presburger formula: $\exists k{\cdot}n_1{=}2k{+}1{\wedge}k{\geq}0$. As a result, Δ_{23}^{flat} is equi-satisfiable to the following base formula:

$$\begin{aligned}\Delta^b{}_{23}{\equiv}&\exists\, x_1,y_1,x_2,y_2,n_1{\cdot}(y{\mapsto}node(x_1,y_1){*}x{\mapsto}node(\texttt{null},y_2){\wedge}x{\neq}\texttt{null}\wedge \\ & y{\neq}\texttt{null}{\wedge}n{=}n_1{+}1){\wedge}(\exists k{\cdot}n_1{=}2k{+}1{\wedge}k{\geq}0)\end{aligned}$$

$\mathcal{T}_2^Q$ is flattened as the tree presented in Fig. 4 which has no cycle. Finally, the base of `Q` is computed based on the open leaf nodes of the tree shown in Fig. 4. It is the disjunction of Δ_{21} and $\Delta^b{}_{23}$ as:

$$\begin{aligned}&\{\exists\, y_1{\cdot}x{\mapsto}node(\texttt{null},y_1){\wedge}y{=}\texttt{null}{\wedge}x{\neq}\texttt{null}{\wedge}n{=}1; \\ &\exists\, x_1,y_1,y_2,k{\cdot}y{\mapsto}node(x_1,y_1){*}x{\mapsto}node(\texttt{null},y_2){\wedge}x{\neq}\texttt{null}{\wedge}y{\neq}\texttt{null}{\wedge}n{=}2k{+}2{\wedge}k{\geq}0\}\end{aligned}$$

Since either disjunct of the set above is satisfiable, so is `Q`.

4 Foundation of Base Computation

In this section, we show that existentially quantified pointer-typed variables are not externally visible wrt. the satisfiability problem. This finding is fundamental for the transformation of an inductive predicate into regular formulas and then flat formulas. The following two functions: $\mathbf{eXPure}(\Delta^b)$ and $\Pi(\pi, \bar{w})$, are relevant in our argument.

Reduction We first define a function called $\mathbf{eXPure}$, which transforms a base formula into an *equi-satisfiable* first-order formula. $\mathbf{eXPure}$ is defined as follows:

$$\mathbf{eXPure}(\exists \bar{w}\cdot\ x_1{\mapsto}c_1(\bar{v}_1){*}...{*}x_n{\mapsto}c_n(\bar{v}_n) \wedge \pi) \equiv$$
$$\exists\ \bar{w}\cdot\ \bigwedge\{x_i{\neq}\texttt{null} \mid i{\in}\{1...n\}\} \wedge \bigwedge\{x_i{\neq}x_j \mid i,j{\in}\{1...n\} \text{ and } i{\neq}j\} \wedge \pi$$

Proposition 1. *For all s such that $s \models \mathbf{eXPure}(\Delta^b)$, there exists s',h such that $s{\subseteq}s'$, $|dom(h)|{=}n{+}|\bar{w}|$, $(s(x_i) \rightarrow _) \in dom(h)\ \forall i{\in}\{1...n\}$, and $s', h \models \Delta^b$ where $|dom(h)|$ is the size of heap $dom(h)$ and $|\bar{w}|$ is the length of sequence $\bar{w}$.*

Proposition 2. *For all s, h such that $s, h \models \Delta^b$, $s \models \mathbf{eXPure}(\Delta^b)$.*

Lemma 1. *Δ^b is satisfiable if only if $\mathbf{eXPure}(\Delta^b)$ is satisfiable.*

Proof The "if" direction follows immediately from Prop. 1. The "only if" direction follows immediately from Prop. 2. □

We remark that the proposed function $\mathbf{eXPure}$ is similar to the well-formed function in [24]. Indeed, the well-formed function is more general than $\mathbf{eXPure}$ as it additionally supports singly-linked lists $lseg$.

Quantifier Elimination Function $\Pi(\pi, \bar{w})$ eliminates the existential quantifiers on pointer-typed variables $\bar{w}^S$. It is defined as follows.

Definition 2 $\Pi(\texttt{true}, \bar{w}) = \texttt{true}$, $\Pi(\texttt{false}, \bar{w}) = \texttt{false}$, $\Pi(v_1{\neq}v_1{\wedge}\pi_1, \bar{w}) = \texttt{false}$, $\Pi(\exists \bar{w}\cdot\ \alpha{\wedge}\phi, \bar{w}) = \exists \bar{w}\cdot\ \Pi(\alpha, \bar{w}){\wedge}\phi$. *Otherwise,*

$$\Pi(v_1{=}v_2{\wedge}\alpha_1, \bar{w}) = \begin{cases} \Pi(\alpha_1[v_1/v_2], \bar{w}) & \textit{if } v_1 \in \bar{w}^S \\ \Pi(\alpha_1[v_2/v_1], \bar{w}) & \textit{if } v_2 \in \bar{w}^S \textit{ and } v_1 \notin \bar{w}^S \\ v_1{=}v_2{\wedge}\Pi(\alpha_1, \bar{w}) & \textit{otherwise} \end{cases}$$
$$\Pi(v_1{\neq}v_2{\wedge}\alpha_1, \bar{w}) = \begin{cases} v_1{\neq}v_2{\wedge}\Pi(\alpha_1, \bar{w}) & \textit{if } v_i{\notin}\bar{w}^S, i{=}\{1,2\} \\ \Pi(\alpha_1, \bar{w}) & \textit{otherwise} \end{cases}$$

For soundness, we assume that α is sorted s.t. equality conjuncts are processed before disequality ones.

Lemma 2. *For all s, $s \models \exists\bar{w}{\cdot}\alpha$ iff there exists $s'{\subseteq}s$ and $s' \models \Pi(\alpha, \bar{w})$.*

We remark that quantifier elimination in equality logic has been studied well and can be done in SMT solvers (i.e., Z3 [11]). In this paper, we present a simplified implementation for efficiency.

Lemma 1 and Lemma 2 imply that it is sound and complete to discard existentially quantified heaps while solving satisfiability in our fragment. The base of a regular formula is computed as follows.

Lemma 3. *For all* s *and* h, $s,h \models \Delta^b{*}\Upsilon^\exists$ *iff there exist* $s'{\subseteq}s$, $h'{\subseteq}h$ *and* $s',h' \models \Delta^b{\wedge}\bigvee\{(\Delta^\exists)^N \mid \Delta^\exists \in \Upsilon^\exists\}$

The proof, based on structural induction on the number of base formulas $\Delta^\exists$ of $\Upsilon^\exists$, is presented in the Appendix. We remark that this result can be implicitly implied from the results presented in [7,19,20]. Now, the problem of base computation in separation logic is reduced to the problem of closure computation for arithmetic constraints. We formally define this reduction as follows.

Definition 3 (Base Computation) *Let* $\Delta^{re}{\equiv}\Delta^b{*}\Upsilon^\exists$ *be a regular formula.* Δ^{re} *is flattable, i.e. can be represented as a base formula, if* $\bigvee\{(\Delta^\exists)^N \mid \Delta^\exists \in \Upsilon^\exists\}$ *is equivalent to a Presburger formula.*

We note that the disjunction set $\Upsilon^\exists$ may be infinite. In the next section we transform each inductive predicate into a set of regular formulas; each of these regular formulas is of the form: $\Delta^{re} \equiv \Delta^b{*}(\exists\bar{w}{\cdot}\mathtt{P_1}(\bar{v}_1){*}...{*}\mathtt{P_n}(\bar{v}_n))$, where $\bar{v}_i^S{\subseteq}\bar{w}$ for all $i{\in}\{1...n\}$. Based on Definition 3, Δ^{re} is equivalent to $\Delta^{re} \equiv \Delta^b{*}(\exists\bar{w}{\cdot}\mathtt{P_1}^N(\bar{v}_1^N){\wedge}...{\wedge}\mathtt{P_n}^N(\bar{v}_n^N))$. Thus, the problem of base computation for inductive predicates is reduced to the problem of closure computation for numeric predicates.

5 Transformation of Inductive Predicates

In this section, we present an algorithm, named `pred2reg`, to transform each inductive predicate into a disjunctive set of regular formulas. Each of these regular formulas is of the form: $\Delta^{re} \equiv \Delta^b{*}(\exists\bar{w}{\cdot}\mathtt{P_1}(\bar{v}_1){*}...{*}\mathtt{P_n}(\bar{v}_n))$, where $\bar{v}_i^S{\subseteq}\bar{w}$ for all $i{\in}\{1...n\}$. For each inductive predicate in $\mathcal{P}$, `pred2reg` first uses procedure `utree` to construct a cyclic unfolding tree to characterise its satisfiability (Sec. 5.1). After that, `pred2reg` uses procedure `extract_regular` to flatten the tree into a set of regular formulas in a bottom-up manner (Sec. 5.2). The correctness of the transformation is presented in Sec. 5.3.

5.1 Constructing Cyclic Unfolding Tree

Procedure `utree` presented in Algorithm 1 aims to construct an unfolding tree given an inductive predicate. This algorithm is an instantiation of the S2SAT algorithm described in [19]. While S2SAT is designed for decision problems (`SAT` or `UNSAT`), `utree` works as a re-write procedure. It transforms an user-defined predicate into an unfolding tree with (virtual) cycles. Given an inductive predicate, say $\mathtt{P}(\bar{v})$, it constructs a cyclic unfolding tree for the formula $\Delta{\equiv}\mathtt{P}(\bar{v})^0_0$. Each iteration (lines 2-12) conducts one of the following four actions. Function `OA` over-approximates every leaf node and checks whether it is unsatisfiable. If it is the case, the function marks the leaf closed. Function `link_back` links a leaf back to an interior node if they have the same free (externally) pointer-based variables. In each such back-link, the leaf node is called a bud and the interior node is called a companion. Function `choose_bfs` chooses an open leaf for the unfolding with function `unfold`.

Algorithm 1: Procedure `utree`

```
   input  : Δ
   output: T_n
1  i←0; T_0←{Δ} ;                                  /* initialize */
2  while true do
3      T_i←OA(T_i) ;                      /* mark unsat and closed */
4      T_i←link_back(T_i) ;                   /* detect similarly */
5      (is_exists, Δ_i)←choose_bfs(T_i) ;  /* open leaf for unfolding */
6      if not is_exists then
7          return T_i;
8      else
9          i←i+1 ;
10         T_i←unfold(Δ_i);
11     end
12 end
```

Over-approximation Given an input tree $\mathcal{T}_i$, for each its leaf node Δ, function `OA` obtains the over-approximation Δ' by substituting all occurrences of inductive predicates appearing in Δ with `true` prior to transforming Δ' into an equi-satisfiable first-order formula π' using function $\mathtt{eXPure}$ (defined in Section 4). Finally, π' is discharged using an SMT solver. If π' is unsatisfiable, so is Δ.

Unfolding In each iteration, our algorithm selects one open leaf node including some occurrences of inductive predicates to expand the tree. The node is selected in a breadth-first manner. Among all open leaf nodes, a node is selected if it contains at least one *observable* occurrence of an inductive predicate, e.g. $\mathtt{P}(\bar{v})^o_u$, where u is the smallest unfolding number. If there are more than one such occurrences, the one with the smallest ordering number is chosen. We remark that a leaf node whose occurrences of inductive predicates are all unobservable is never unfolded as this leaf is already a regular formula. For each new node derived, `unfold` marks it open and creates a new edge accordingly. Let $\mathtt{Q}(\bar{t})^{o_l}_{_}$ denote a predicate occurrence of the derived node, its unfolding number is set to `u+1` if it is (not necessary directly) recursive. Otherwise, it is `u`. Its sequence number is set to $o_l{+}o$.

Linking back Function `link_back` connects an open leaf with at least one observable occurrence of inductive predicate (say, $\exists \bar{w}_1{\cdot}\kappa_1{\wedge}\alpha_1{\wedge}\phi_1$) to an interior node (say, $\exists \bar{w}_2{\cdot}\kappa_2{\wedge}\alpha_2{\wedge}\phi_2$) as follows.

1. First, it discards all unobservable points-to predicates and all unobservable inductive predicates, and then eliminates existentially quantified variables for pointer equalities and disequalities in the two formulas. Afterwards, the two formulas become $\kappa'_1{\wedge}\alpha_{1a}{\wedge}\phi_1$ and $\kappa'_2{\wedge}\alpha_{2a}{\wedge}\phi_2$ where $\alpha_{1a}{\equiv}\Pi(\alpha_1, \bar{w}_1^S)$, $\alpha_{2a}{\equiv}\Pi(\alpha_2, \bar{w}_2^S)$.
2. Secondly, it constructs α_{1b} (resp., α_{2b}) by augmenting the closure for equalities on pointers into α_{1a} (resp., α_{2a}): if $x{=}y \in \alpha$ and $y{=}z \in \alpha$ then $x{=}z \in \alpha$.
3. Thirdly, it builds a set of addresses, i.e. $\mathtt{B}_1$, $\mathtt{B}_2$, for each formula. Given a formula $\exists \bar{w}{\cdot}\kappa{\wedge}\alpha{\wedge}\phi$, its set of addresses `B` is collected as follows. If $x{\mapsto}c(_){\in}\kappa$ then $x \in \mathtt{B}$; if $x \in \mathtt{B}$ and $x{=}y \in \alpha$ then $y \in \mathtt{B}$.

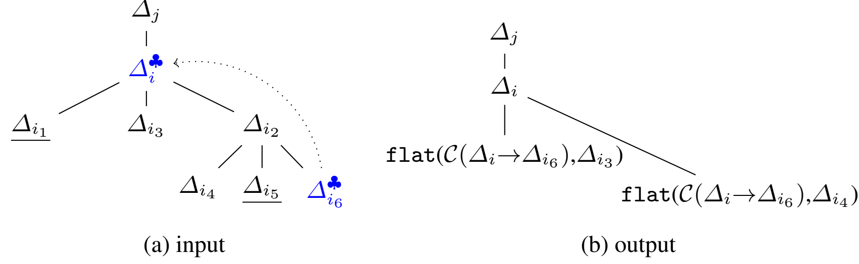


Fig. 5: Flattening minimal cyclic sub-tree.

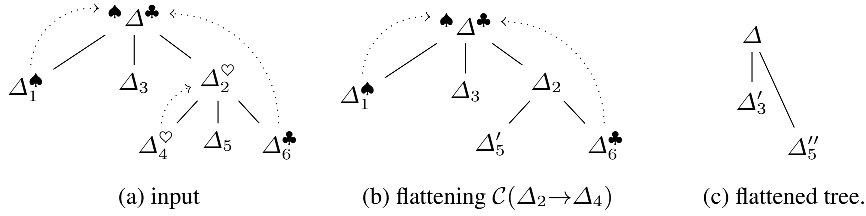
4. Next, it adds into α_{1b} (resp. α_{2b}) the Boolean abstraction of separating predicates, e.g. $\alpha_{1c}{\equiv}\alpha_{1b}{\wedge}\bigwedge\{x{\neq}\texttt{null} \mid x \in \texttt{B}_1\}{\wedge}\bigwedge\{x{\neq}y \mid x{,}y \in \texttt{B}_1\}$ and similarly for α_{2c}. Note that we assume redundant constraints in α_{1c} and α_{2c} are discarded.
5. Finally, $\kappa_1'{\wedge}\alpha_{1c}{\wedge}\phi_1$ is linked to $\kappa_2'{\wedge}\alpha_{2c}{\wedge}\phi_2$ if the following conditions hold:
   - i) $\texttt{B}_1$ and $\texttt{B}_2$ are identical; and
   - ii) α_{1c} and α_{2c} are identical; and
   - iii) For all occurrence $\texttt{P}_\texttt{i}(\bar{t})^{o_2}_{u_2}$ in κ_2', there exists one occurrence $\texttt{P}_\texttt{i}(\bar{v})^{o_1}_{u_1}$ in κ_1' such that $u_1{>}u_2$ and for all free variable $v_i \in \bar{v}$, t_i is a free variable and $\alpha_{1c} \implies t_i{=}v_i$.

5.2 Flattening Cyclic Unfolding Tree

To compute a set of regular formulas for a cyclic tree, procedure `extract_regular` flattens its cycles using procedure `flat_tree` iteratively in a bottom-up manner until there is no cycle left. Afterward, the set is derived from the disjunctive set of flattened open leaf nodes. In particular, it repeatedly applies `flat_tree` on minimal cyclic sub-trees. A cyclic sub-tree is minimal if it does not include any (nested) cyclic sub-trees and among other companion nodes, its companion node is the one which is closest to a leaf node. We use $\mathcal{C}(\Delta_c{\rightarrow}\{\Delta_b^1, .., \Delta_b^n\})$ to denote a minimal cyclic sub-tree where back-links are formed between companion Δ_c and buds Δ_b^i. If there is only one bud in the tree, we write $\mathcal{C}(\Delta_c{\rightarrow}\Delta_b)$ for simplicity. Function `flat_tree` takes a minimal cyclic sub-tree as an input and returns a set of regular formulas, each of them corresponds to an open leaf node in the tree.

We illustrate procedure `flat_tree` through the example in Fig. 5 where the tree in the left (Fig. 5(a)) is a minimal cyclic sub-tree $\mathcal{C}(\Delta_i{\rightarrow}\Delta_{i_6})$ and is the input of `flat_tree`. For a minimal cyclic sub-tree, `flat_tree` first eliminates all closed leaf nodes (e.g., Δ_{i_1} and Δ_{i_5}). We remark that if all leaf nodes of a cyclic sub-tree are unsatisfiable, the whole sub-tree is pruned i.e. replaced by a closed node with `false`. After that, the open leaf nodes (e.g., Δ_{i_3} and Δ_{i_4}) are flattened by the function `flat`. Finally, flattened nodes (e.g., $\texttt{flat}(\mathcal{C}(\Delta_i{\rightarrow}\Delta_{i_6}), \Delta_{i_3})$ and $\texttt{flat}(\mathcal{C}(\Delta_i{\rightarrow}\Delta_{i_6}), \Delta_{i_5})$) are connected directly to the root of the minimal cyclic sub-tree (e.g., Δ_i); all other nodes (e.g., Δ_{i_2} and Δ_{i_6}) are discarded. The result is presented in Fig 5(b).

Function `flat` takes a minimal cyclic sub-tree, e.g. $\mathcal{C}(\Delta_c{\rightarrow}\Delta_b)$, and an open leaf node in the sub-tree, e.g. $\Delta_j^{re}{\equiv}\exists\bar{w}{\cdot}\Delta^b{*}\texttt{P}_1(\bar{t}_1){*}...{*}\texttt{P}_\texttt{n}(\bar{t}_n)$ where $\bar{t_i}^S{\subseteq}\bar{w}\ \forall i \in \{1...n\}$, as inputs. It generates a regular formula representing the set of formulas which can



(a) input (b) flattening $\mathcal{C}(\Delta_2 \rightarrow \Delta_4)$ (c) flattened tree.

Fig. 6: Flattening a complex cyclic tree.

be obtained by unfolding according to the path which iterates the cycle (from Δ_c to Δ_b and back to Δ_c) an arbitrary number of times and finally follows the path from Δ_c to Δ_j^{re}. As the formulas obtained by unfolding according to the paths of the cycle are existentially heap-quantified, following lemma 3, they are equi-satisfiable with their numeric part. As so, `flat` constructs a new arithmetical inductive predicate, called $\mathtt{P_{cyc}}$, to extrapolate the arithmetic constraints over the path from Δ_c to Δ_b. The generation of $\mathtt{P_{cyc}}$ only succeeds if the arithmetical constraints of Δ_c, Δ_j^{re} and Δ_b are of the form ϕ_c, $\phi_c \wedge \phi_{base}$ and $\phi_c \wedge \phi_{rec}$, respectively. Let $t_1,..,t_i$ be a sequence of integer-typed parameters of the matched inductive predicates in Δ_c and $t'_1,..,t'_i$ be the corresponding sequence of integer-typed parameters of the matched inductive predicates in Δ_b. Then, `flat` generates the predicate $\mathtt{P_{cyc}}(t_1,...,t_i)$ defined as follows.

$$\mathtt{pred}\ \mathtt{P_{cyc}}(t_1,...,t_i) \equiv \exists \bar{w}_b{\cdot}\phi_{base} \vee \exists \bar{w}_c{\cdot}\phi_{rec} \wedge \mathtt{P_{cyc}}(t'_1,...,t'_i)$$

where $\bar{w}_b{=}FV(\phi_{base}) \setminus \{t_1,...,t_i\}$ and $\bar{w}_c{=}FV(\phi_{rec}) \setminus \{t_1,...,t_i\} \setminus \{t'_1,...,t'_i\}$. Afterward, `flat` produces the output as: $\exists \bar{w}{\cdot}\Delta^b \wedge \mathtt{P^N_1}(\bar{t}^N_1) \wedge ... \wedge \mathtt{P^N_n}(\bar{t}^N_n) \wedge \mathtt{P_{cyc}}(t_1,..,t_i)$.

Finally, we highlight the flattening procedure with a fairly complex example in Fig. 6. The input tree, presented in Fig. 6a), has three cycles. First, `flat_tree` flattens the lower tree $\mathcal{C}(\Delta_2 \rightarrow \Delta_4)$ and produces the tree in the middle, Fig. 6b), where $\Delta'_5 \equiv \mathtt{flat}(\mathcal{C}(\Delta_2 \rightarrow \Delta_4), \Delta_5)$. After that, it flattens the intermediate tree and produces a cyclic-free tree in Fig. 6c). In the final tree, $\Delta'_3 \equiv \mathtt{flat}(\mathcal{C}(\Delta \rightarrow \{\Delta_1, \Delta_6\}), \Delta_3)$ and $\Delta''_5 \equiv \mathtt{flat}(\mathcal{C}(\Delta \rightarrow \{\Delta_1, \Delta_6\}), \Delta'_5)$. As the latter tree has two back-links, the corresponding arithmetic predicate generated for it has two recursive branches.

5.3 Correctness

Procedure `utree` First, it is easy to verify that the cyclic unfolding tree derived by the procedure `utree` preserves satisfiability and unsatisfiability of the given predicate.

Lemma 4. *Let $\mathcal{T}_i$ be the cyclic unfolding tree derived by procedure* `utree` *for predicate* $\mathtt{P}_i(\bar{t}_i)$. *$\mathcal{T}_i$ contains at least one satisfiable leaf node iff* $\mathtt{P}_i(\bar{t}_i)$ *is satisfiable.*

Next, we provide a complexity analysis for procedure `utree`. Intuitively, the procedure terminates if there is no more leaf node for unfolding. This happens when all leaf nodes are either base formulas, or formulas with unobservable occurrences of inductive predicates or linked back. The last case occurs if two nodes involved in a back link have similar arrangement over free predicate arguments. As the number of these free arguments are finite, so is the number of arrangements. In particular, suppose we have N

inductive predicates (e.g., $\mathtt{P_1}$,...,$\mathtt{P_N}$), and m is the maximal length of predicate parameters (including one more for `null`). The maximal free *pointer-typed* variables of an inductive predicate is also m. We compute the complexity based on N and m.

Lemma 5. *Every path of the cyclic unfolding tree generated by procedure* `utree` *(algorithm 1) has at most* $\mathcal{O}(2^m \times (2^m)^N \times 2^{2m^2})$ *nodes.*

Procedure `extract_regular` For simplicity, we only discuss the minimal cyclic subtrees including one cycle. Let $\mathcal{C}(\Delta_c{\rightarrow}\Delta_b)$ be a minimal cyclic sub-tree and ${\Delta^b}_j$ be a satisfiable leaf node in the tree. Let $\mathtt{lassos}(\Delta_c,\Delta_b,{\Delta^b}_j,k)$ be a formula which is obtained by unfolding the tree following the cycle (from Δ_c to Δ_b and back to Δ_c) k times and finally following the path from Δ_c to ${\Delta^b}_j$.

Lemma 6. $s,h \models \bigvee_{k\geq 0} \mathtt{lassos}(\Delta_c,\Delta_b,{\Delta^b}_j,k)$ *iff there exist* $s' \subseteq s$ *and* $h' \subseteq h$ *such that* $s',h' \models \mathtt{flat}(\mathcal{C}(\Delta_c{\rightarrow}\Delta_b), {\Delta^b}_j)$.
Proof By structural induction on k and lemma 3. □
The correctness of function `flat_tree` immediately follows Lemma 6.

Lemma 7. *Let* $\mathcal{C}(\Delta_c{\rightarrow}\Delta_b)$ *be a minimal cyclic sub-tree.* $\mathcal{C}(\Delta_c{\rightarrow}\Delta_b)$ *is satisfiable iff there exist* $\Delta^{re} \in \mathtt{flat_tree}(\mathcal{C}(\Delta_c{\rightarrow}\Delta_b))$ *and* s,h *such that* $s,h \models \Delta^{re}$.

6 Decision Procedure

Satisfiability of inductive predicates is solvable if all cycles of their unfolding trees can be flattened into regular formulas and all these regular formulas are flattable.

6.1 Decidable Fragment

Our decidable fragment is based on classes of regular formulas where each formula is flattable. We focus on the special class of regular formulas generated from the procedure `pred2reg` in the previous section (i.e., based on inductive predicates) and show how to compute bases for this class. In particular, each regular formula in this class is a set of base formulas unfolded from inductive predicates, e.g. $\Delta^{re} \equiv \Delta^b{*}(\exists\bar{w}{\cdot}\mathtt{P_1}(\bar{v}_1){*}...{*}\mathtt{P_n}(\bar{v}_n))$, where $\bar{v}_i^S{\subseteq}\bar{w}$ for all $i{\in}\{1...n\}$. Following Lemma 3, we have: Δ^{re} is equi-satisfiable with $\Delta^b{\wedge}(\exists\bar{w}{\cdot}{\mathtt{P_1}}^N(\bar{t}_1^N){*}...{\wedge}{\mathtt{P_n}}^N(\bar{t}_n^N))$. Hence, Δ^{re} is flattable, i.e. can be represented by a base formula, if every ${\mathtt{P_i}}^N(\bar{t}_i^N)$ is equivalent with a Presburger formula ϕ_i for all $i{\in}\{1..n\}$. As so, Δ^{re} is equi-satisfiable to the base formula: $\Delta^b{\wedge}(\exists\bar{w}{\cdot}\phi_1{\wedge}...{\wedge}\phi_n)$. In consequence, we define a class of flattable formulas, called flat DPI formula, based on DPI predicates where each predicate is equivalent to a Presburger formula [31].

An arithmetic inductive predicate is DPI if it is not inductive or is defined as follows.

$$\mathtt{pred}\ P^N(\bar{x}) \equiv \bigwedge_{1\leq i\leq m} \phi_{0,i} \vee \exists\bar{z}\cdot\bigwedge_{1\leq i\leq m}\phi_i \wedge \bigwedge_{1\leq l\leq L} P^N(\bar{z}^l)$$

where m is the arity of P^N, $FV(\phi_{0,i}){\subseteq}\{x_i\}$, $\bar{z}{\supseteq}\bar{z}^l$, and there exists j such that ϕ_i is either of $x_i{=}f(\bar{z}_i)$, $x_i{\geq}f(\bar{z}_i)$, or $x_i{\leq}f(\bar{z}_i)$ for all $i{\neq}j$, and ϕ_j is either of the following:

(1) $x_j = f(\bar{z}_j) + c \wedge \phi'$ (2) $x_j \geq f(\bar{z}_j) + c \wedge \phi'$ (3) $x_j \leq f(\bar{z}_j) + c \wedge \phi'$

(4) a conjunction of the following forms with some integer constant $n > 0$:
$\phi', nx_j = f(\bar{z}_j), nx_j \geq f(\bar{z}_j), or\ nx_j \leq f(\bar{z}_j)$

Algorithm 2: Deriving Bases.

```
    input  : 𝒫
    output: base^𝒫
 1  base^𝒫←∅; 𝒫^cyc←∅; Pres←∅;
 2  foreach Pi(t̄i) ∈ 𝒫 do
 3  |  (reg(Pi(t̄i)), 𝒫^cyc_Pi)←pred2reg(Pi(t̄i));     /* to regular formulas */
 4  |  𝒫^cyc←𝒫^cyc∪𝒫^cyc_Pi;      /* and numeric predicates for cycles */
 5  end
 6  foreach P^N_j(t̄j) ∈ 𝒫^N∪𝒫^cyc do
 7  |  Pres(P^N_j(t̄j))←pred2pres(pred P^N_j(t̄j));    /* compute fixed points */
 8  end
 9  foreach Pi(t̄i) ∈ 𝒫 do
10  |  base^𝒫(Pi(t̄i))←subst(Pres, reg(Pi(t̄i)));
11  end
12  return base^𝒫;
```

where c is some integer constant, $\bar{z}_j$ is $z_j^1, \ldots, z_j^L$, ϕ' is an arithmetical formula such that $FV(\phi') \subseteq \bar{z}_j$ and $\phi'[z/\bar{z}_j]$ is true for any z, $f(\bar{z}_j)$ is a combination of $z_j^1, \ldots, z_j^L$ with *max*, *min*, defined by $f(\bar{z}_j) ::= z_j^l \mid max(f(\bar{z}_j), f(\bar{z}_j)) \mid min(f(\bar{z}_j), f(\bar{z}_j))$, and f's may be different from each other in the conjunction of (4).

The authors in [31] showed that each inductive predicate DPI exactly represents some eventually periodic sets which are equivalent to some sets characterized by some Presburger arithmetical formulas.

Lemma 8 ([31]). *For every DPI inductive predicate $P(\bar{x})$, there is a formula ϕ equivalent to $P(\bar{x})$ such that ϕ does not contain any inductive predicates.*

Finally, we define flat DPI formulas based on the DPI predicates as follows.

Definition 4 (Flat DPI Formula) *Let $\Delta \equiv \Delta^b * (\exists \bar{w} \cdot \mathtt{P_1}(\bar{v}_1) * \ldots * \mathtt{P_n}(\bar{v}_n))$ where $\bar{v}_i^S \subseteq \bar{w}$ for all $i \in \{1 \ldots n\}$. Δ is flattable if, for all $i \in \{1 \ldots n\}$, the arithmetic predicate $\mathtt{P_i}^N(\bar{v}_i^N)$ is a DPI predicate.*

We remark that flat formulas can be extended to any class of inductive predicates whose numeric projections can be defined in Presburger arithmetic. For example, in App. **??**, we show how to transform an inductive predicate into periodic relations [5]. As every periodic relation is equivalent to a Presburger formula [3,13,22,5], an instantiation of the flat formulas based on periodic relations is straightforward.

Now, we define a decidable fragment based on the flattable formulas.

Definition 5 (Decidable Fragment) *Let $\mathcal{P} = \{\mathtt{P_1}, \ldots, \mathtt{P_n}\}$ and $\mathcal{P}^{cyc} = \{\mathtt{P_1^{cyc}}, \ldots, \mathtt{P_m^{cyc}}\}$ be arithmetic predicates generated by function `flat_tree` while transforming the predicates in $\mathcal{P}$ using `pred2reg`. Solving satisfiability for every inductive predicate $\mathtt{P_i}$ in $\mathcal{P}$ is decidable iff every arithmetic predicate in $\mathcal{P}^N \cup \mathcal{P}^{cyc}$ is DPI where $\mathcal{P}^N = \{\mathtt{P_1}^N, \ldots, \mathtt{P_n}^N\}$.*

We remark that the decidable fragment is parameterized by the classes of flattable formulas. It is extensible to any decidable fragment of arithmetic inductive predicates.

6.2 Decision Algorithm

Computing Bases for Inductive Predicates We present a procedure, called `pred2base`, to compute for each inductive predicate in $\mathcal{P}$ a set of base formulas. `pred2base` is described in Algorithm 2. It takes a set of predicates $\mathcal{P}$ as input and produces a mapping $\texttt{base}^{\mathcal{P}}$ which maps each inductive predicate to a set of base formulas. `pred2base` first uses procedure `pred2reg` (lines 2-5) to transform the predicates into regular formulas (which are stored in `reg`) together with a set of arithmetic inductive predicates (which are stored in $\mathcal{P}^{\text{cyc}}$) while flattening cycles. We recap that for each inductive predicate function `pred2reg` first uses procedure `utree` in Sec. 5.1 to construct a cyclic unfolding tree and then uses procedure `extract_regular` in Sec. 5.2 to flatten the tree into a set of regular formulas. After that, it uses function `pred2pres` (lines 6-8) to compute for each inductive predicate in $\mathcal{P}^{\text{N}}{\cup}\mathcal{P}^{\text{cyc}}$ an equivalent Presburger formula. These relations is stored in the mapping `Pres`. Finally, at lines 9-11 it obtains a set of base formulas from substituting all arithmetic inductive predicates in the corresponding regular formulas by their equivalent Presburger formulas.

Satisfiability Solving Let Δ be a formula over a set of user-defined predicates $\mathcal{P}$ where $\mathcal{P}{=}\{\texttt{P}_1, ..., \texttt{P}_\texttt{m}\}$. The satisfiability of Δ is reduced to the satisfiability of the predicate: $\texttt{pred}\ \texttt{P}_0(\bar{t}_0) \equiv \Delta$; where $\texttt{P}_0$ is a fresh symbol and $\bar{t}_0$ is the set of free variables in Δ: $\bar{t}_0{\equiv}FV(\Delta)$.

6.3 Correctness

We now show the correctness of our procedure in the decidable fragment.

Theorem 1. *Procedure* `pred2base` *terminates for the decidable fragment.*

Proposition 3. *Let* $\texttt{P}_\texttt{i}(\bar{t}_i)$ *be an inductive predicate in the decidable fragment. If* $\texttt{P}_\texttt{i}(\bar{t}_i)$ *is satisfiable,* $\texttt{reg}(\texttt{P}_\texttt{i}(\bar{v}_i))$ *produced by procedure* `pred2reg` *contains at least one satisfiable formula.*

Proposition 4. *Let* $\texttt{P}_\texttt{i}(\bar{t}_i)$ *be an inductive predicate in the decidable fragment. If procedure* `pred2reg` *can derive for it a non-empty set of satisfiable regular formulas, then there exists an unfolding tree of* $\texttt{P}_\texttt{i}(\bar{t}_i)$ *containing at least one satisfiable leaf node.*

The proof is trivial.

Theorem 2. *Suppose that* $\mathcal{P}$ *is a system of inductive predicates in the proposed decidable fragment. Assume that procedure* `pred2base` *can derive for every* $\texttt{P}_\texttt{i}(\bar{t}_i)$ *a base* $\texttt{base}^{\mathcal{P}}\texttt{P}_\texttt{i}(\bar{t}_i)$. *For all* s, h *and* $\texttt{P}_\texttt{i} \in \mathcal{P}$, $s{,}h \models \texttt{P}_\texttt{i}(\bar{t}_i)$ *iff there exist* $s'{\subseteq}s$, $h'{\subseteq}h$, *and* $\Delta^b \in \texttt{base}^{\mathcal{P}}\texttt{P}_\texttt{i}(\bar{t}_i)$ *such that* $s'{,}h' \models \Delta^b$.

Proof The “if” direction follows immediately from lemma 3, lemma 8 and Prop. 3. The “only if” direction follows immediately from lemma 3, lemma 8 and Prop. 4. □

The above theorem implies that base generation for a system of heap-only inductive predicates is decidable with the complexity $\mathcal{O}(2^m \times 2^{2m^2} \times (2^m)^N)$ time in the worst case. This finding is consistent with the one in [6].

7 Implementation and Evaluation

The proposed solver has been implemented based on the S2SAT framework [19]. We use Fixcalc [27] to compute closure for arithmetic relations. The SMT solver Z3 [11] is used for satisfiability problems over arithmetic. In the following, we first describe how to infer over-/under-approximated bases for those predicates beyond the decidable fragment. While over-approximated bases are important for unsatisfiability in verifying safety [9,28,17], under-approximated bases are critical for satisfiability in finding bugs [18]. After that, we show experimental results on the base computation and the satsisfiability problem.

We sometimes over-approximate a base formula in order to show unsatisfiability, which helps to prune infeasible disjunctive program states and discharge entailment problems with empty heap in RHS [9]. In particular, the validity of the entailment checking $\Delta \vdash \mathtt{emp}{\wedge}\pi_c$ is equivalent to the unsatisfiability of the satisfiability problem $\Delta{\wedge}\neg\pi_c$. Similarly, we sometimes under-approximate a base formula in order to show satisfiability, which helps to generate counter-examples that highlight scenarios for real errors. For the latter, our approach is coupled with an error calculus [18] to affirm a real bug in HIP/S2 system [9,17]. When an error (which may be a false positive) is detected, we perform an additional satisfiability check on its pre-condition to check its feasibility. If it is satisfied, we invoke an error explanation procedure to identify a sequence of reachable code statements leading to the error [18]. With our new satisfiability procedure, we can confirm true bugs (which were not previously possible) so as to provide support towards fixing program errors.

It can be implied from section 6 that generating approximated base for a formula relies on the approximation of the arithmetic part of inductive predicates, and then of regular formulas. To compute an under-approximation, we adopt the k-index bound approach from [5]. In particular, to compute a closure for a predicate $\mathtt{P}^{\mathtt{N}}$, we only consider all unfolded formulas which have at most k occurrences of inductive predicates. As the disjunction of the bounded formulas is an under-approximation, the closure computed is an under-approximated base. To compute an over-approximation, we adopt the approach in [32]. In particular, first we transform the system of arithmetic inductive predicates into a system of constrained Horn clauses. After that, we use Fixcalc [27] to solve the constraints and compute an over-approximated base.

In the rest, we show the capability of our base inference and its application in program verification. We remark that, in [19] we show how a satifiability solver in separation logic is applied into the verification system $\mathtt{S2}_{\mathtt{td}}$. The experiments were performed on a machine with the Intel i7-960 (3.2GHz) processor and 16 GB of RAM.

Base Inference Using our proposed procedure, we have inferred bases for a broad range of data structures. The results are shown in Table 1. The first column shows the names of inductive predicates including cyclic linked-list, list segment, linked-list with even size, binary trees. TLL is binary trees whose nodes point to their parent and all leave nodes are linked as a singly-linked list. In all these predicates, n is the length. The second column shows the inferred bases. Note that we use _ for existentially quantified variables for simplicity. The third column presents type of the base (exact base or over-approximated base). The last column captures time (in seconds) of the computation.

Table 1: Bases Inference for Data Structures

Data Structure	Base Inferred	Type	Sec.
Singly llist (size)	$\{\texttt{emp}\wedge\texttt{root}{=}\texttt{null}\wedge n{=}0; \texttt{root}{\mapsto}c_1(_,_)\wedge n{>}0\}$	exact	0.15
Even llist (size)	$\{\texttt{emp}\wedge\texttt{root}{=}\texttt{null}\wedge n{=}0; \exists i\cdot \texttt{root}{\mapsto}c_1(_,_)\wedge i{>}0 \wedge n{=}2{*}i\}$	exact	0.28
Sorted llist (size, sorted)	$\{\texttt{emp}\wedge\texttt{root}{=}\texttt{null}\wedge n{=}0 \wedge sm{\leq}lg;$ $\texttt{root}{\mapsto}c_1(_,_)\wedge n{>}0 \wedge sm{\leq}lg\}$	exact	0.14
Doubly llist (size)	$\{\texttt{emp}\wedge\texttt{root}{=}\texttt{null}\wedge n{=}0; \texttt{root}{\mapsto}c_1(_,_)\wedge n{>}0\}$	exact	0.16
CompleteT (size, minheight)	$\{\texttt{emp}\wedge\texttt{root}{=}\texttt{null}\wedge n{=}0\wedge minh{=}0;$ $\texttt{root}{\mapsto}c_2(_,_)\wedge n{\leq}2{*}minh{-}1\wedge nmin{\leq}n; \}$	over	2.3
Heap trees (size, maxelem)	$\{\texttt{emp}\wedge\texttt{root}{=}\texttt{null}\wedge n{=}0\wedge mx{=}0;$ $\texttt{root}{\mapsto}c_2(_,_)\wedge n{>}0 \wedge mx{\geq}0\}$	over	0.3
AVL (height, size)	$\{\texttt{emp}\wedge\texttt{root}{=}\texttt{null}\wedge n{=}0\wedge bal{=}1;$ $\texttt{root}{\mapsto}c_2(_,_)\wedge h{>}0\wedge n{\geq}h\wedge n{\geq}2{*}h{-}2\}$	over	0.67
RBT(size, color, blackheight)	$\{\texttt{emp}\wedge\texttt{root}{=}\texttt{null}\wedge n{=}0 \wedge bh{=}1 \wedge cl{=}0;$ $\texttt{root}{\mapsto}c_2(_,_)\wedge n{>}0\wedge cl{=}1;\ \texttt{root}{\mapsto}c_2(_,_)\wedge n{>}0\wedge cl{=}0\}$	over	0.61
TLL	$\{\texttt{root}{\mapsto}c_4(_,_,_,_)\wedge\texttt{root}{=}\texttt{ll}; \texttt{root}{\mapsto}c_4(_,_,_,_){*}ll{\mapsto}c_4(_,_,_,_)\}$	exact	0.14

Table 2: Experimental Results on Satisfiability Problems

Data Structure (pure properties)	#query	#unsat	#sat	Time (seconds)
Singly llist (size)	666	75	591	0.85
Even llist (size)	139	125	14	1.04
Sorted llist (size, sorted)	217	21	196	0.46
Doubly llist (size)	452	50	402	1.08
CompleteT (size, minheight)	387	33	354	55.41
Heap trees (size, maxelem)	487	67	400	7.22
AVL (height, size)	881	64	817	52.15
RBT (size, blackheight, color)	1741	217	1524	40.85
TLL	128	13	115	0.39

While our proposal can infer bases for most predicates, there are also predicates where we have inferred approximated bases (AVL tree, heap tree, complete tree and red-black tree). These typically occur when they are outside of the decidable fragments. In all these cases, we had to infer over-approximation and under-approximations by k-index (under-approximated bases are not shown for brevity).

Satisfiability Solving We have implemented a new satisfiability solver based on the base inference. Our solver supports input as presented in Sec. 2 as well as in SMT2 format based on the description in [30]. We have integrated our proposed satisfiability procedure into `HIP/S2` [9,17], a verification system based on separation logic. Table 2 shows the experimental results on a set of satisfiability problems generated from the verification of heap-manipulating programs. The first column lists the data structures and their pure properties. The second column lists the total number of satisfiability queries sent to the decision procedure. The next two columns show the amount of unsat and sat

queries, respectively. We use $k{=}10$ for the inference of under-approximation. The last column captures the processed time (in seconds) for queries of each data structure. The experimental results show that our satisfiability solver could exactly decide all sat and unsat problems from our suite of verification tasks for complex data structures. This is despite the use of approximated bases for four examples, namely `Heap` trees, `Complete` trees, `AVL` and `RBT`, that are outside of the decidable fragment.

8 Related Work

Solving satisfiability in fragments of separation logic with inductive predicates has been studied extensively. Several decidable fragments were proposed with some restrictions over either shape of inductive predicates, or arithmetic, or satisfiability queries. Proposals in [2,23,10,15,6,12,19,31][5] presented decision procedures for fragments including inductive predicates with heap properties, pure equalities but without arithmetic. Initial attempts like [2,23,10] focus only on linked lists. Smallfoot [2] exploits the small model property of linked lists. SPEN [12] enhances the decidable fragment above with nested lists and skip lists. [15] extends the decidable fragment with tree structures. The satisfiability problem is reduced to decidability of Monadic Second Order Logic on graphs with bounded tree width. Finally, SLSAT [6] proposes a decision procedure for arbitrary inductive definitions. The essence of SLSAT is an algorithm to derive for each predicate an equi-satisfibale base. Our work is an extension of SLSAT to support a combination of inductive predicates and arithmetic. To support arithmetical properties, instead of computing a least fixed point for heap property, our procedure first constructs a cyclic unfolding tree and then flattens the tree to derives the base. The decidable fragment in [31] has the following restrictions: for each inductive definition, (1) it has only a single induction case, (2) its inductive case has only a single occurrence of the inductive predicate unless the satisfiability of the spatial part becomes trivial, and (3) mutual inductive definitions are not allowed. Our decidable fragment removes these restrictions. Finally, [19] supports satisfiability checking of the universal fragment restricted in both shape and arithmetic. In comparison, our procedure supports arbitrary inductive definitions with relations based on semilinear sets over arithmetical parameters.

In terms of decision procedures supporting inductive predicates and arithmetic, GRASShoper [25] and Asterix [24] are among the first decision procedures where shape definitions are restricted to linked lists. The decidable fragments have been recently widened in extended GRASShoper [26], CompSPEN [14], $\texttt{S2SAT}_{\texttt{SL}}$ [19], [20] and [31]. While CompSPEN extends the graph-based algorithm [10] to doubly-linked list, $\texttt{S2SAT}_{\texttt{SL}}$ is an instantiation of `S2SAT` [19]. For back-link construction, the instantiations [19,20] are based on both heap and arithmetic constraints. Our algorithm in this work is more compositional i.e., it first forms back-links based only on the heap domain and then reduces the satisfiability problems into the satisfiability problems over arithmetic. By doing so, we can exploit well-developed results for the arithmetic domain. In this work, we reuse the result based on semilinear sets [31] for the arithmetic. In App **??**, we show how to adapt results based on periodic relations [5]. We are currently

[5] We remark that works in [2,23,10,15,12] also discussed decision procedures for the entailment problem which is beyond the scope of this paper.

investigating how to use regular model checking [4] to enhance our decision procedure. The procedure $\texttt{S2SAT}_{\texttt{SL}}$ presented in [19] constructs back-links based on a combination of heap and arithmetic domains. In this work, back-links are constructed based on heap domain only. The satisfiability of the arithmetic part is processed in a separate phase. By doing so, the decidable fragment proposed in this paper is much more expressive when compared with the decidable fragment in [19]. For instance, while the decidable fragment in [19] includes a restricted fragment of heap-only predicates, the decidable fragment presented in this work includes arbitrary heap-only predicates. Our proposal may be viewed as an extension of the work [31] with the construction of cyclic unfolding trees to support arbitrary spatial predicates. To the best of our knowledge, our proposal is the most powerful decision procedure for satisfiability in separation logic.

9 Conclusion

We have presented a novel decision procedure for an expressive fragment of separation logic including shape and arithmetic properties. Our procedure is based on computing an equi-satisfiable base formula for each inductive predicate. This base computation, in turn, relies on the computation of the base for a set of flat formulas. We provide a complexity analysis to show that the decision problem for heap-based fragment is, in the worst case, in exponential time. We have implemented our proposal in a prototype tool and integrated it into an existing verification system. Experimental results shows that our procedure works effectively over the set of satisfiability benchmarks.

Acknowledgements. Quang Loc and Jun Sun are partially supported by NRF grant RGNRF1501 and Wei-Ngan by MoE Tier-2 grant MOE2013-T2-2-146.

References

1. J. Berdine, C. Calcagno, and P. W. O'Hearn. A Decidable Fragment of Separation Logic. In *FSTTCS*. Springer-Verlag, December 2004.
2. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic Execution with Separation Logic. In *APLAS*, volume 3780, pages 52–68, November 2005.
3. Bernard Boigelot. Symbolic methods for exploring infinite state spaces, 1998.
4. Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. *Regular Model Checking*, pages 403–418. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.
5. Marius Bozga, Radu Iosif, and Filip Konecný. Fast acceleration of ultimately periodic relations. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 227–242, 2010.
6. James Brotherston, Carsten Fuhs, Juan A. Navarro Pérez, and Nikos Gorogiannis. A decision procedure for satisfiability in separation logic with inductive predicates. In *CSL-LICS '14*, pages 25:1–25:10, New York, NY, USA, 2014. ACM.
7. James Brotherston and Nikos Gorogiannis. Cyclic abduction of inductively defined safety and termination preconditions. In *Proceedings of SAS-21*, number 8723 in LNCS, pages 68–84. Springer, 2014.
8. Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FSTTCS*, pages 108–119, 2001.

9. W.N. Chin, C. David, H.H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *SCP*, 77(9):1006–1036, 2012.
10. B. Cook, C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*, volume 6901, pages 235–249. 2011.
11. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
12. Constantin Enea, Ondrej Lengal, Mihaela Sighireanu, and Tomas Vojnar. *Compositional Entailment Checking for a Fragment of Separation Logic*, pages 314–333. Springer International Publishing, Cham, 2014.
13. Alain Finkel and Jerome Leroux. *How to Compose Presburger-Accelerations: Applications to Broadcast Protocols*, pages 145–156. 2002.
14. Xincai Gu, Taolue Chen, and Zhilin Wu. *A Complete Decision Procedure for Linearly Compositional Separation Logic with Data Constraints*, pages 532–549. IJCAR 2016, 2016.
15. R. Iosif, A. Rogalewicz, and J. Simácek. The tree width of separation logic with recursive definitions. In *CADE*, pages 21–38, 2013.
16. S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26, London, January 2001.
17. Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pages 52–68, 2014.
18. Quang Loc Le, Asankhaya Sharma, Florin Craciun, and Wei-Ngan Chin. Towards complete specifications with an error calculus. In *NASA Formal Methods.*, pages 291–306, 2013.
19. Quang Loc Le, Jun Sun, and Wei-Ngan Chin. Satisfiability modulo heap-based programs. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, Proceedings, Part I*, pages 382–404, Cham, 2016.
20. Quang Loc Le, Jun Sun, and Shengchao Qin. Verifying heap-manipulating programs using constrained horn clauses (technical report). 2017.
21. Quang Loc Le, Jun Sun, Shengchao Qin, and Wei-Ngan Chin. Frame inference for inductive entailment proofs in separation logic (technical report). 2017.
22. Antoine Mine. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
23. Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI '11*, pages 556–566. ACM, 2011.
24. Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic modulo theories. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems - Volume 8301*, pages 90–106, New York, NY, USA, 2013. Springer-Verlag New York, Inc.
25. R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using smt. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044, pages 773–789. 2013.
26. Ruzica Piskac, Thomas Wies, and Damien Zufferey. Grasshopper:complete heap verification with mixed specifications. In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference. Proceedings*, pages 124–139. 2014.
27. C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. In *ASIAN*, pages 331–345, 2006.
28. X. Qiu, P. Garg, A. Ştefănescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242, New York, NY, USA, 2013. ACM.
29. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, pages 55–74, 2002.
30. Cristina Serban. A formalization of separation logic in smt-lib v2.5 syntax, types and semantics. Technical report, Verimag, 2015. [Online; accessed Jan-2017].
31. Makoto Tatsuta, Quang Loc Le, and Wei-Ngan Chin. *Decision Procedure for Separation Logic with Inductive Definitions and Presburger Arithmetic*, pages 423–443. APLAS, 2016.

32. Minh-Thai Trinh, Quang Loc Le, Cristina David, and Wei-Ngan Chin. Bi-abduction with pure properties for specification inference. In *Programming Languages and Systems: 11th Asian Symposium, APLAS 2013, Australia. Proceedings*, pages 107–123, 2013.

A Proof of Proposition 1

Proof We prove lemma 1 by structural induction on Δ.

- $\Delta{\equiv}\texttt{emp}$. Trivial.
- $\Delta{\equiv}x_1{\mapsto}c_1(\bar{v}_1)$. We construct additional stack s_1 and heap h_1 such that $s(x_1){=}l_1$, $dom(h_1){=}\{l_1 \rightarrow r_1\}$ and $r_1(c_1, f_{1i}){=}s_1(\bar{v}_{1_i})$.
  $\iff |\text{dom}(h_1)|{=}1 \wedge s \cup s_1, h_1 \models x_1{\mapsto}c_1(\bar{v_1}_i)$ (semantics of points-to predicate in Fig. 2).
- $\Delta{\equiv}\kappa_1{*}\kappa_2$ where $\kappa_1{\equiv}x_1{\mapsto}c_1(\bar{v}_1){*}...{*}x_i{\mapsto}c_i(\bar{v}_i)$, $\kappa_2{\equiv}x_{i+1}{\mapsto}c_{i+1}(\bar{v}_{i+1}){*}...{*}x_n{\mapsto}c_n(\bar{v}_n)$

$$
\begin{array}{ll}
& s \models \mathbf{eXPure}(\Delta) \\
\iff & s \models \mathbf{eXPure}(\kappa_1{*}\kappa_2) \\
\iff & s \models \bigwedge_{j=1}^{i} x_j{\neq}\texttt{null}\wedge \bigwedge_{k=i+1}^{n} x_k{\neq}\texttt{null}\wedge \\
& \quad \bigwedge\{x_j{\neq}x_k \mid j \in \{1,..,i,.,n\}{\wedge}k \in \{1,..,i,..n\}{\wedge}j{\neq}k\} \text{ (Definition of } \mathbf{eXPure}) \\
\iff & s \models \mathbf{eXPure}(\kappa_1){\wedge}\mathbf{eXPure}(\kappa_2){\wedge} \\
& \quad \bigwedge\{x_j{\neq}x_k \mid j \in \{1,..,i\}{\wedge}k \in \{i+1,..n\}\} \text{ (Definition of } \mathbf{eXPure}) \\
\implies & \exists s_1, h_1{\cdot}|\text{dom}(h_1)|{=}i \wedge (s_1(x_j) \rightarrow _) \in dom(h_1)\ \forall j \in 1..i \wedge s{\subseteq}s_1{\wedge}s_1, h_1 \models \kappa_1 \text{ and} \\
& \exists s_2, h_2{\cdot}|\text{dom}(h_2)|{=}n{-}i \wedge (s_2(x_j) \rightarrow _) \in dom(h_2)\ \forall j \in (i+1)..n \wedge s{\subseteq}s_2 \wedge \\
& \quad s_2, h_2 \models \kappa_2 \text{ and} \\
& s \models \bigwedge\{x_j{\neq}x_k \mid j \in \{1,..,i\}{\wedge}k \in \{i+1,..n\}\} \qquad \text{(induction hypothesis)} \\
\iff & \exists s_1, h_1{\cdot}|\text{dom}(h_1)|{=}i \wedge (s_1(x_j) \rightarrow _) \in dom(h_1)\ \forall j \in 1..i \wedge s{\subseteq}s_1{\wedge}s_1, h_1 \models \kappa_1 \text{ and} \\
& \exists s_2, h_2{\cdot}|\text{dom}(h_2)|{=}n{-}i \wedge (s_2(x_j) \rightarrow _) \in dom(h_2)\ \forall j \in (i+1)..n \wedge \\
& \quad s{\subseteq}s_2{\wedge}s_2, h_2 \models \kappa_2 \text{ and} \\
& \bigwedge\{s(x_j){\neq}s(x_k) \mid j \in \{1,..,i\}{\wedge}k \in \{i+1,..n\}\} \qquad \text{(semantics of } \neq) \\
\iff & \exists s_1, h_1{\cdot}|\text{dom}(h_1)|{=}i \wedge (s_1(x_j) \rightarrow _) \in dom(h_1)\ \forall j \in 1..i \wedge s{\subseteq}s_1{\wedge}s_1, h_1 \models \kappa_1 \text{ and} \\
& \exists s_2, h_2{\cdot}|\text{dom}(h_2)|{=}n{-}i \wedge (s_2(x_j) \rightarrow _) \in dom(h_2)\ \forall j \in (i+1)..n \wedge s{\subseteq}s_2 \wedge \\
& \quad s_2, h_2 \models \kappa_2 \wedge \text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset \\
\iff & \exists s_1, h_1, s_2, h_2{\cdot}|\text{dom}(h_1{\cdot}h_2)|{=}n \wedge ((s_1 \cup s_2)(x_i) \rightarrow _) \in dom(h_1 \cdot h_2)\ \forall i \in 1..n \wedge \\
& s{\subseteq}s_1 \cup s_2 \wedge s_1{\cup}s_2, h_1{\cdot}h_2 \models \kappa_1{*}\kappa_2 \quad \text{(semantics of } *) \\
\iff & \exists s_1, h_1, s_2, h_2{\cdot}|\text{dom}(h_1{\cdot}h_2)|{=}n \wedge ((s_1 \cup s_2)(x_i) \rightarrow _) \in dom(h_1 \cdot h_2)\ \forall i \in 1..n \wedge \\
& s{\subseteq}s_1 \cup s_2 \wedge s_1{\cup}s_2, h_1{\cdot}h_2 \models \Delta
\end{array}
$$

- $\Delta{\equiv}\exists\bar{w}{\cdot}x_1{\mapsto}c_1(\bar{v}_1){*}...{*}x_n{\mapsto}c_n(\bar{v}_n){\wedge}\pi$, assume $s \models \mathbf{eXPure}(\Delta)$.
  We construct additional stack s_1 and heap h_1 such that $s(x_i){=}l_i$ where $x_i \in x_1...x_n$, $\text{dom}(h_1){=}\bigcup\{l_i \rightarrow r_i\}$ and $r_i(c_i, f_{i_j}){=}s_1(v_{ij})$.
  We construct another heap h_2 such that for each $w_j \in \bar{w}$, $s(w_j){=}l_j$ and $\text{dom}(h_2){=}\bigcup\{l_j \rightarrow r_j\}$ where $r_j(c_j, f_{j_i}) = s_1(v_{j_i}) = k_{j_i}$ for some value k_{j_i}.
  We have, $|\text{dom}(h)|{=}n{+}|\bar{w}|$. Furthermore, following the semantics of a symbolic heap, we have $s \cup s_1, h_1 \cdot h_2 \models \Delta$

□

B Proof of Proposition 2

By structural induction on Δ

– $\Delta{\equiv}\texttt{emp}$. Straightforward.
– $\Delta{\equiv}x{\mapsto}c(v_i)$.

$$\begin{array}{l} s,h \models \Delta \\ \iff s,h \models x{\mapsto}c(v_i) \\ \implies \exists \nu \cdot \nu{=}s(x) \wedge \nu \neq s(\texttt{null}) \quad \text{(semantics for points-to predicate - Fig 2)} \\ \iff s \models x{\neq}\texttt{null} \\ \iff s \models \mathbf{eXPure}(\Delta) \quad (\mathbf{eXPure}\text{ definition}) \end{array}$$

– $\Delta = \exists v_1..v_n \cdot \kappa{\wedge}\pi$.

$$\begin{array}{l} s,h \models \Delta \\ \iff s,h \models \exists v_{1...n} \cdot \kappa \wedge \pi \\ \iff \exists \nu_1..\nu_n \cdot s[(v_i{\mapsto}\nu_i)_{i=1}^n], h \models \kappa \textit{ and } s[(v_i{\mapsto}\nu_i)_{i=1}^n] \models \pi \quad \text{(model for sep. constraint - Fig 2)} \\ \implies \exists \nu_1..\nu_n \cdot s[(v_i{\mapsto}\nu_i)_{i=1}^n] \models \mathbf{eXPure}(\kappa) \textit{ and } s[(v_i{\mapsto}\nu_i)_{i=1}^n] \models \pi \\ \quad \text{(induction hypothesis and } \mathbf{eXPure}\text{ definition)} \\ \iff \exists \nu_1..\nu_n \cdot s[(v_i{\mapsto}\nu_i)_{i=1}^n] \models \mathbf{eXPure}(\kappa) \wedge \pi \quad \text{(semantics for sep. constraint - Fig 2)} \\ \iff \exists \nu_1..\nu_n \cdot s[(v_i{\mapsto}\nu_i)_{i=1}^n] \models \mathbf{eXPure}(\kappa \wedge \pi) \quad (\mathbf{eXPure}\text{ definition}) \\ \iff \exists \nu_1..\nu_n \cdot s \models \exists\, v_1..v_n \cdot \mathbf{eXPure}(\kappa \wedge \pi) \quad \text{(model for separation constraint - Fig 2)} \\ \iff \exists \nu_1..\nu_n \cdot s \models \mathbf{eXPure}(\exists\, v_1..v_n \cdot \kappa \wedge \pi) \quad (\mathbf{eXPure}\text{ definition}) \\ \iff s \models \mathbf{eXPure}(\Delta) \end{array}$$

– $\Delta = \kappa_1{*}\kappa_2$.

$$\begin{array}{l} s,h \models \Delta \\ \iff s,h \models \kappa_1{*}\kappa_2 \\ \iff s,h_1 \models \kappa_1 \wedge s,h_2 \models \kappa_2 \wedge h = h_1{*}h_2 \\ \quad \text{Let } \mathbf{eXPure}(\kappa_1) = \alpha_1,\ \mathbf{eXPure}(\kappa_2) = \alpha_2 \text{ where } \texttt{PTO}(\kappa_1) \cap \texttt{PTO}(\alpha_2) = \emptyset \\ \implies s \models \alpha_1 \wedge s \models \alpha_2 \text{ and } \texttt{PTO}(\kappa_1) \cap \texttt{PTO}(\alpha_2) = \emptyset \quad \text{(induction hypothesis)} \\ \iff s \models \alpha_1{\wedge}\alpha_2{\wedge} \bigwedge\{\ v_i{\neq}v_j \mid v_i{\in}\texttt{PTO}(\kappa_1), v_i{\in}\texttt{PTO}(\kappa_2)\} \\ \iff s \models \mathbf{eXPure}(\kappa_1{*}\kappa_2) \quad (\mathbf{eXPure}\text{ definition of } *) \\ \iff s \models \mathbf{eXPure}(\Delta) \end{array}$$

$\texttt{PTO}(\kappa)$ returns x_i variables of points-to predicates $x_i{\mapsto}c_i(\bar{v}_i)$ in heap κ. □

C Proof of Lemma 2

While Lemma 9 states the “if” direction, Lemma 10 states the “only if” direction.

Lemma 9. *Let $\exists\bar{w}{\cdot}\alpha$ be a (dis)equality conjunction. If $s \models \alpha$ then $s \models \Pi(\alpha, \bar{w})$*

Proof We prove lemma 9 by induction on size of α. The proof for this lemma is based on semantics of $\wedge$.

– Case $\alpha{\equiv}\texttt{true}$. Trivially.
– Case $\alpha{\equiv}v_1{=}v_2{\wedge}\alpha_1$ and $v_1 \in \bar{w}$.

$$\begin{array}{l} \quad s \models \alpha \\ \iff s \models v_1{=}v_2 \text{ and } s \models \alpha_1 \ \text{(semantics of } \wedge) \\ \implies s \models \alpha_1[v_2/v_1] \end{array}$$

– Case $\alpha{\equiv}v_1{=}v_2{\wedge}\alpha_1$ and $v_1 \notin \bar{w}$, $v_2 \in \bar{w}$.

$$\begin{array}{rl} & s \models \alpha \\ \Longleftrightarrow & s \models v_1{=}v_2 \text{ and } s \models \alpha_1 \ \ \text{(semantics of } \wedge) \\ \Longrightarrow & s \models \alpha_1[v_1/v_2] \end{array}$$

– Other cases are similar

Lemma 10. *Let $\exists \bar{w}{\cdot}\alpha$ be a* satisfiable *(dis)equality conjunction. If $s \models \mathit{\Pi}(\alpha, \bar{w})$ $\exists s'{\cdot}s \subseteq s'$ and $s' \models \alpha$*

Proof By induction on size of α.

1. Case $\alpha{\equiv}\texttt{true}$. Trivially.
2. Case $\alpha{\equiv}v_1{=}v_2{\wedge}\alpha_1$ and $v_1 \in \bar{w}$.
   By assumption, $s \models \mathit{\Pi}(\alpha_1, \bar{w})$. By induction, there exists a stack s_1 such that $s \subseteq s_1$ and $s_1 \models \alpha_1$. We examines four subcases:
   (a) Case $s_1(v_1)$ and $s_2(v_2)$ are both defined. Never.
   (b) Case $s_1(v_1)$ is defined and $s_1(v_2)$ is undefined. We define $s_2(v_2)$ such that $s_2(v_2){=}s_1(v_1)$. This implies $s_1 \cup s_2 \models v_1{=}v_2$. Thus, $(s_1 \cup s_2) \models \alpha$.
   (c) Case $s_1(v_2)$ is defined, $s_1(v_1)$ is undefined and both are undefined are similar.
3. Case $\alpha{\equiv}v_1{=}v_2{\wedge}\alpha_1$ and $v_1 \notin \bar{w}$, $v_2 \in \bar{w}$.
   By assumption, $s \models \mathit{\Pi}(\alpha_1, \bar{w})$. By induction, there exists a stack s_1 such that $s \subseteq s_1$ and $s_1 \models \alpha_1$. We examines two subcases:
   (a) Case $s_1(v_1)$ is defined. As $v_2 \in \bar{w}$, v_2 has been substituted already. Thus, $s_1(v_2)$ is undefined. We define $s_2(v_2)$ such that $s_2(v_2){=}s_1(v_1)$. This implies $s_1 \cup s_2 \models v_1{=}v_2$. Thus, $(s_1 \cup s_2) \models \alpha$.
   (b) Case $s_1(v_1)$ is undefined. Never.
4. Case $\alpha{\equiv}v_1{=}v_2{\wedge}\alpha_1$ and $v_1 \notin \bar{w}$, $v_2 \notin \bar{w}$. By assumption, $s \models \mathit{\Pi}(\alpha_1, \bar{w})$. By induction, there exists a stack s_1 such that $s \subseteq s_1$ and $s_1 \models \alpha_1$. As $v_1 \notin \bar{w}$ and $v_2 \notin \bar{w}$, $s_1(v_1)$ and $s_1(v_2)$ are both defined. As α is satisfiable, $s_1(v_1){=}s_1(v_2)$. This implies $s_1 \models v_1{=}v_2$. Thus, $s_1 \models \alpha$.
5. Case $\alpha{\equiv}v_1{\neq}v_2{\wedge}\alpha_1$ and $v_1 \in \bar{w}$, $v_2 \in \bar{w}$.
   By assumption, $s \models \mathit{\Pi}(\alpha_1, \bar{w})$. By induction, there exists a stack s_1 such that $s \subseteq s_1$ and $s_1 \models \alpha_1$. As $v_1 \in \bar{w}$ and $v_2 \in \bar{w}$, $s_1(v_1)$ and $s_2(v_2)$ are both undefined. We define $s_2(v_1)$ and $s_2(v_2)$ such that $s_2(v_2){\neq}s(v_2)$ (As domain of integer is infinite, we always find a such value for v_1 and v_2) This implies $s_1 \cup s_1 v_1{\neq}v_2$. Thus, $(s_1 \cup s_2) \models \alpha$.
6. Case $\alpha{\equiv}v_1{\neq}v_2{\wedge}\alpha_1$ and $v_1 \in \bar{w}$, $v_2 \notin \bar{w}$. Similar to Case 5.
7. Case $\alpha{\equiv}v_1{=}v_2{\wedge}\alpha_1$ and $v_1 \notin \bar{w}$, $v_2 \in \bar{w}$. Similar to Case 5.
8. Case $\alpha{\equiv}v_1{=}v_2{\wedge}\alpha_1$ and $v_1 \notin \bar{w}$, $v_2 \notin \bar{w}$. By assumption, $s \models \mathit{\Pi}(\alpha_1, \bar{w})$. By induction, there exists a stack s_1 such that $s \subseteq s_1$ and $s_1 \models \alpha_1$. As $v_1 \notin \bar{w}$ and $v_2 \notin \bar{w}$, $s_1(v_1)$ and $s_1(v_2)$ are both defined. As α is satisfiable, $s_1(v_1){\neq}s_1(v_2)$. This implies $s_1 \models v_1{\neq}v_2$. Thus, $s_1 \models \alpha$.

D Proof of Lemma 3

Let n be the number of base formulas of $\Upsilon^{\exists}$. we prove this lemma by structural induction on n.

Base Case

- n=0. trivial
- n=1. Let $\Delta^{\exists}{\equiv}\exists\bar{w}{\cdot}x_1{\mapsto}c_1(\bar{v}_1){*}...{*}x_n{\mapsto}c_n(\bar{v}_n){\wedge}\alpha{\wedge}\phi$ where $x_i{\in}\bar{w}$ for all $i{\in}\{1,..,n\}$ and $FV(\alpha)\subseteq\bar{w}$. We prove that for any Δ_b, for all s, h such that $s{,}h\models\Delta^b{*}\Delta^{\exists}$ iff there exist $s'{\subseteq}s$, $h'{\subseteq}h$ and $s'{,}h'\models\Delta^b{\wedge}(\exists\bar{w}{\cdot}\phi)$. The following shows the "if" direction.

$$\begin{array}{lll} & s{,}h\models\Delta^b{*}\Delta^{\exists} & \\ \Longrightarrow & s{,}h\models\Delta^b{\wedge}\mathbf{eXPure}(\Delta^{\exists}) & \text{(Prop. 2)} \\ \Longleftrightarrow & s{,}h\models\Delta^b{\wedge}(\exists\bar{w}{\cdot}\alpha_1{\wedge}\alpha{\wedge}\phi), & FV(\alpha_1)\subseteq\bar{w} \\ \Longleftrightarrow & s'{,}h\models\Delta^b{\wedge}(\exists\bar{w}{\cdot}\Pi(\alpha_1,\bar{w}){\wedge}\Pi(\alpha,\bar{w}){\wedge}\phi) & \text{(Lemma 2)} \\ \Longleftrightarrow & s'{,}h\models\Delta^b{\wedge}(\exists\bar{w}{\cdot}\mathtt{true}\,{\wedge}\mathtt{true}\,{\wedge}\phi) & \text{(as } FV(\alpha_1)\subseteq\bar{w}, FV(\alpha)\subseteq\bar{w}) \\ \Longleftrightarrow & s'{,}h\models\Delta^b{\wedge}(\exists\bar{w}{\cdot}\phi) & \end{array}$$

Proof for the "only if" direction is similar.

Induction Case Assume the lemma is true for all $n{\leq}k$. We prove that the lemma is also true for $n{=}k{+}1$. We use $\mathtt{ptos}(\Delta)$ to return all points-to variables in Δ.

$$\begin{array}{lll} & s{,}h\models\Delta^b{*}(\exists\bar{w}{\cdot}{\Delta^b}_1{*}...{*}{\Delta^b}_k{*}{\Delta^b}_{k+1}) & \mathtt{ptos}({\Delta^b}_i)\in\bar{w}\forall i\in\{1..k{+}1\} \\ \Longleftrightarrow & s'{,}h'\models\Delta^b{*}(\exists\bar{w}{\cdot}{\Delta^b}_{k+1}){\wedge}{\Delta^b}^N_1{\wedge}...{\wedge}{\Delta^b}^N_k & \text{(by induction on the first } k \text{ conjuncts)} \\ \Longleftrightarrow & s'{,}h'\models\Delta^b{*}\exists\bar{w}{\cdot}({\Delta^b}_{k+1}{\wedge}{\Delta^b}^N_1{\wedge}...{\wedge}{\Delta^b}^N_k)^N & \text{(by induction on the} (k{+}1)^{th} \text{ conjunct)} \\ \Longleftrightarrow & s'{,}h'\models\Delta^b{*}\exists\bar{w}{\cdot}{\Delta^b}^N_{k+1}{\wedge}{\Delta^b}^N_1{\wedge}...{\wedge}{\Delta^b}^N_k & \text{(by definition of the numeric projection)} \\ \Longleftrightarrow & s'{,}h'\models\Delta^b{*}\exists\bar{w}{\cdot}{\Delta^b}^N_1{\wedge}...{\wedge}{\Delta^b}^N_k{\wedge}{\Delta^b}^N_{k+1} & \end{array}$$

E Proof of Lemma 5

Proof The complexity is based on the longest path from an interior node Δ_{comp} of an unfolding tree to a leaf Δ_{bud} such that Δ_{bud} can be linked back to Δ_{comp}. In the worst case, this back-link is established if (i) all observable points-to predicates are linked (i.e. $\mathcal{O}(2^m)$ possibilities), (ii) observable arguments of inductive predicates are linked (i.e. $\mathcal{O}(2^m)^N$ possibilities) and (iii) (dis)equalities constraints between free pointer-typed variables are identical (i.e. $\mathcal{O}(2^{2m^2})$ possibilities). □

F Proof of Theorem 1

Proof `pred2base` relies on the termination of procedure `pred2reg`. Lemma 5 shows that it terminates for constructing a cyclic unfolding tree. Moreover, it is trivial to show that the number of cycles in a cyclic unfolding tree is finite. It implies that function `extract_regular` always terminates. Hence, `pred2reg` terminates. □

G Proof of Proposition 3

Proof As $\mathtt{P_i}(\bar{t}_i)$ is satisfiable, lemma 4 implies that `utree` can always derive for it a cyclic unfolding tree, called $\mathcal{T}_i$, which has at least one satisfiable leaf node. The proof

is based on structural induction on the nested level of $\mathcal{T}_i$ (i.e., the maximal number of companion nodes of a path in the tree) using lemma 7 for the base case. Note that as arithmetical constraints in recursive branches of DPI predicates are existentially quantified, the conditions for successfully generating $\mathtt{P}_\mathtt{i}^\mathtt{cyc}$ $(\forall i{\in}\{1...m\})$ to extrapolate the cycles are immediately satisfied. $\square$