7-2016

Satisfiability modulo heap-based programs

Quang Loc LE

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Wei-Ngan CHIN

Citation

LE, Quang Loc; SUN, Jun; and CHIN, Wei-Ngan. Satisfiability modulo heap-based programs. (2016). *Proceedings of the 28th International Conference, CAV 2016, Toronto, Canada, July 17–23*. 9779, 382-404.

Available at: https://ink.library.smu.edu.sg/sis_research/4937

Satisfiability Modulo Heap-Based Programs

Quang Loc Le[1(✉)], Jun Sun[1], and Wei-Ngan Chin[2]

[1] Singapore University of Technology and Design, Singapore, Singapore
lequangloc@gmail.com
[2] National University of Singapore, Singapore, Singapore

Abstract. In this work, we present a semi-decision procedure for a fragment of separation logic with user-defined predicates and Presburger arithmetic. To check the satisfiability of a formula, our procedure iteratively unfolds the formula and examines the derived disjuncts. In each iteration, it searches for a proof of either satisfiability or unsatisfiability. Our procedure is further enhanced with automatically inferred invariants as well as detection of cyclic proof. We also identify a syntactically restricted fragment of the logic for which our procedure is terminating and thus complete. This decidable fragment is relatively expressive as it can capture a range of sophisticated data structures with non-trivial pure properties, such as size, sortedness and near-balanced. We have implemented the proposed solver and a new system for verifying heap-based programs. We have evaluated our system on benchmark programs from a software verification competition.

Keywords: Decision procedures · Satisfiability · Separation logic · Inductive predicates · Cyclic proofs

1 Introduction

Satisfiability solvers, particularly those based on Satisfiability Modulo Theory (SMT) technology [3,19], have made tremendous practical advances in the last decade to the point where they are now widely used in tools for applications as diverse as bug finding [27], program analyses [4] to automated verification [2]. However, current SMT solvers are based primarily on first-order logic, and do not yet cater to the needs of resource-oriented logics, such as separation logic [26,40]. Separation logic has recently established a solid reputation for reasoning about programs that manipulate heap-based data structures. One of its strengths is the ability to concisely and locally describe program states that hold in separate regions of heap memory. In particular, a spatial conjunction (i.e., $\kappa_1 * \kappa_2$) asserts that a given heap can be decomposed into two disjoint regions and the formulas, κ_1 and κ_2, hold respectively and separately in the two memory regions. In this work, we investigate the problem of verifying heap-manipulating programs in the framework of SMT. We reduce this problem to solving verification conditions representing precise program semantics [9,10,22,44].

Developing an SMT solver supporting separation logic with inductive predicates and Presburger arithmetic is challenging as the satisfiability problem for this fragment is undecidable [30,31]. We focus on an expressive fragment which consists of spatial predicates expressing empty heap assertion (`emp`), points-to assertion ($x{\mapsto}c(\bar{v})$), and inductive predicate assertions ($\mathtt{P}(\bar{v})$). Moreover, it may include pure constraints on data values and capture desired properties of structural heaps (such as size, height, sortedness and even near-balanced tree properties). We thus face the challenge of handling recursive predicates with pure properties, that are inherently infinite. Furthermore, we would like to support both satisfiability (`SAT`) and unsatisfiability (`UNSAT`) checks.

There have been a number of preliminary attempts in this direction. For instance, early proposals *fixed* the set of shape predicates that may be used, for example, to linked lists (in SeLoger [17,23], and SLL$\mathbb{B}$ [34]) or trees (GRIT [36]). There are few approaches supporting user-defined predicates [14,25,39]. Brotherston *et al.* recently made an important contribution by introducing `SLSAT`, a decision procedure for a fragment of separation logic with arbitrary *shape-only* inductive predicates [12]. However, `SLSAT` is limited to the shape domain, whereas shape predicates extended with pure properties are often required for automated verification of functional correctness.

In this paper, we start by proposing a new procedure, called `S2SAT`, which combines under-approximation and over-approximation for simultaneously checking `SAT` and `UNSAT` properties for a sound and complete theory augmented with inductive predicates. `S2SAT` takes a set of user-defined predicates and a logic formula as inputs. It iteratively constructs *an unfolding tree* by unfolding the formula in a breadth-first, flow- and context-sensitive manner until either a symbolic model, or a proof of unsatisfiability or a fixpoint (e.g., a cyclic proof) is identified. In each iteration, it searches over the leaves of the tree (the disjunction of which is equivalent to the input formula) to check whether there is a satisfiable leaf (which proves satisfiability) or whether all leaves are unsatisfiable. In particular, to prove `SAT`, it considers *base disjuncts* which are derived from base-case branches of the predicates. These disjuncts are under-approximations of the input formula and critical for satisfiability. Disjuncts which have no inductive predicates are precisely decided. To prove `UNSAT`, `S2SAT` over-approximates the leaves prior to prove `UNSAT`. Our procedure systematically enumerates all disjuncts derived from a given inductive formula, so it is terminating for `SAT`. However, it may not be terminating for `UNSAT` with those undecidable augmented logic. To facilitate termination, we propose an approach for fixpoint computation. This fixpoint computation is useful for domains with finite model semantics i.e., collecting semantics for a given formula of such domains is finite. In other words, the input formula is unsatisfiable when the unfolding goes on forever without uncovering any models. We have implemented one instantiation of the fixpoint detection for inductive proving based on cyclic proof [13] s.t. the soundness of the cyclic proof guarantees the well-foundedness of all reasoning.

```
  struct node {int val; node next;}    8 node ll(int i){
1 int main(int n){                     9   if(i==0) return null;
2   if(n<0) return 0;                  10  else return new node(i,ll(i-1)); }
3   node x=ll(n);
4   int r=test(x);                     11 int test(node p){
5   if(!r) ERROR();                    12   if(p==null) return 1;
6   return 1;                          13   else {if(p->val<0) return 0;
7 }                                    14          else return test(p->next); }}
```

Fig. 1. Motivating example.

To explicitly handle heap-manipulating programs, we propose a separation logic instantiation of S2SAT, called S2SAT$_{\mathtt{SL}}$. Our base theory is a combination of the aforementioned separation logic predicates except inductive predicates. We show that our decision procedure for this base theory is sound and complete. S2SAT$_{\mathtt{SL}}$ over-approximates formulas with soundly inferred predicate invariants. In addition, we describe some syntax restrictions such that S2SAT$_{\mathtt{SL}}$ is always able to construct a cyclic proof for a restricted formula so that our procedure is terminating and complete.

To summarize, we make the following technical contributions in this work.

- We introduce cyclic proof into a satisfiability procedure for a base theory augmented with inductive predicates (refer to Sect. 3).
- We propose a satisfiability procedure for separation logic with user-defined predicates and Presburger arithmetic (Sect. 4).
- We prove that S2SAT$_{\mathtt{SL}}$ is: (i) sound for SAT and UNSAT; (ii) and terminating (i.e., proposing a new decision procedure) for restricted fragments (Sect. 5).
- We present a mechanism to automatically derive sound (over-approximated) invariants for user-defined predicates (Sect. 6).
- We have implemented the satisfiability solver S2SAT$_{\mathtt{SL}}$ and the new verification system, called S2$_{\mathtt{td}}$. We evaluated S2SAT$_{\mathtt{SL}}$ and S2$_{\mathtt{td}}$ with benchmarks from recent competitions. The experimental results show that our system is expressive, robust and efficient (Sect. 7).

Proofs of Lemmas and Theorems presented in this paper are available in the companion technical report [30].

2 Illustrative Example

We illustrate how our approach works with the example shown in Fig. 1. Our verification system proves that this program is memory safe and function ERROR() (line 5) is never called. Our system uses symbolic execution in [6,14] and large-block encoding [8] to provide a semantic encoding of verification conditions. For safety, one of the generated verification conditions is: $\Delta_0 \equiv ll(n,x)_0^0 {*} test(x,r_1)_0^1 {\wedge} n{\geq}0 {\wedge} r_1{=}0$. If Δ_0 is unsatisfiable, function ERROR() is never called. In Δ_0, ll and test are Interprocedural Control Flow Graph (ICFG) of the functions `ll` and `test`. Our system eludes these ICFGs as inductive predicates. For each predicate, a parameter `res` is appended at the end to model the return value of the function; for instance, the variables x (in `ll`) and r_1 (in `test`) of Δ_0 are the actual parameters corresponding to `res`. Each inductive predicate instance is also labeled with a subscript for the unfolding number and a superscript for the sequence number, which are used to control the unfolding in a breadth-first and flow-sensitive manner.

To discharge Δ_0, $\texttt{S2SAT}_{\texttt{SL}}$ iteratively derives a series of unfolding trees $\mathcal{T}_i$. An unfolding tree is a tree such that each node is labeled with an unfolded disjunct, corresponding to a path condition in the program. We say that a leaf of $\mathcal{T}_i$ is closed if it is unsatisfiable; otherwise it is open. During each iteration, $\texttt{S2SAT}_{\texttt{SL}}$ either proves `SAT` by identifying a satisfiable leaf of $\mathcal{T}_i$ which contains no user-defined predicate instances or proves `UNSAT` by showing that an over-approximation of all leaves is unsatisfiable. Initially, $\mathcal{T}_0$ contains only one node Δ_0. As Δ_0 contains inductive predicates, it is not considered for proving `SAT`. $\texttt{S2SAT}_{\texttt{SL}}$ then over-approximates Δ_0 to a first-order logic formula by substituting each predicate instance with its corresponding sound invariants in order to prove `UNSAT`. We assume that `ll` (resp. `test`) is annotated with invariant $i{\geq}0$ (resp. $0{\leq}\texttt{res}{\leq}1$). Hence, the over-approximation of Δ_0 is computed as: $\pi_0{\equiv}n{\geq}0{\wedge}0{\leq}r_1{\leq}1{\wedge}n{\geq}0{\wedge}r_1{=}0$. Formula π_0 is then passed to an SMT solver, such as Z3 [19], for unsatisfiable checking. As expected, π_0 is not unsatisfiable.

Next, $\texttt{S2SAT}_{\texttt{SL}}$ selects an open leaf for unfolding to derive $\mathcal{T}_1$. A leaf is selected in a breadth-first manner; furthermore a predicate instance of the selected leaf is selected for unfolding if its sequence number is the smallest. With Δ_0, the `ll` instance is selected. As so, $\mathcal{T}_1$ has two open leaves corresponding to two derived disjuncts:

$$\Delta_{11}{\equiv}test(x,r_1)^1_0{\wedge}n{\geq}0{\wedge}r_1{=}0{\wedge}n{=}0{\wedge}x{=}\texttt{null}$$
$$\Delta_{12}{\equiv}x{\mapsto}node(n,r_2){*}ll(n_1,r_2)^0_1{*}test(x,r_1)^1_0{\wedge}n{\geq}0{\wedge}r_1{=}0{\wedge}n{\neq}0{\wedge}n_1{=}n{-}1$$

Since Δ_{11} and Δ_{12} include predicate instances, they are not considered for `SAT`. To prove `UNSAT`, $\texttt{S2SAT}_{\texttt{SL}}$ computes their over-approximated invariants:

$$\pi_{11}{\equiv}0{\leq}r_1{\leq}1{\wedge}n{\geq}0{\wedge}r_1{=}0{\wedge}n{=}0{\wedge}x{=}\texttt{null}$$
$$\pi_{12}{\equiv}x{\neq}\texttt{null}{\wedge}n_1{\geq}0{\wedge}0{\leq}r_1{\leq}1{\wedge}n{\geq}0{\wedge}r_1{=}0{\wedge}n{\neq}0{\wedge}n_1{=}n{-}1$$

As neither π_{11} nor π_{12} is unsatisfiable, $\texttt{S2SAT}_{\texttt{SL}}$ selects `test` of Δ_{11} for unfolding to construct $\mathcal{T}_2$. For efficiency, unfolding is performed in a context-sensitive manner. A branch is infeasible (and pruned in advance) if its invariant is inconsistent with the (over-approximated) context. For instance, the invariant of the `then` branch at line 12 of `test` is $\texttt{inv}_{test_o}{\equiv}p{=}\texttt{null}{\wedge}\texttt{res}{=}1$. As $\texttt{inv}_{test_o}$ (after proper renaming) is inconsistent with π_{11}, this branch is infeasible. Similarly, both `else` branches of `test` are infeasible. For $\mathcal{T}_3$,

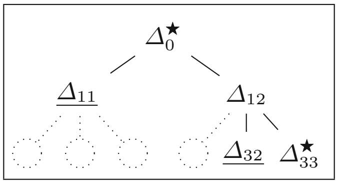

Fig. 2. Unfolding tree $\mathcal{T}_3$.

the remaining leaf Δ_{12} is selected for unfolding. As the `test`'s unfolding number is smaller than `ll`'s, `test` is selected. After the `then` branch is identified as infeasible and pruned, $\mathcal{T}_3$ is left with two open leaves as shown in Fig. 2, where infeasible leaves are dotted-lined. Δ_{32} and Δ_{33} are as below.

$$\Delta_{32}{\equiv}x{\mapsto}node(n,r_2){*}ll(n_1,r_2)^0_1{\wedge}\underline{n{\geq}0}{\wedge}r_1{=}0{\wedge}\underline{n{\neq}0}{\wedge}n_1{=}n{-}1{\wedge}x{\neq}\mathtt{null}{\wedge}\underline{n{<}0}{\wedge}r_1{=}0$$
$$\Delta_{33}{\equiv}x{\mapsto}node(n,r_2){*}ll(n_1,r_2)^0_1{*}test(r_2,r_1)^1_1{\wedge}n{\geq}0 \wedge r_1{=}0{\wedge}n{\neq}0{\wedge}n_1{=}n{-}1$$
$$\quad{\wedge}x{\neq}\mathtt{null}{\wedge}n{\geq}0$$

As Δ_{32} and Δ_{33} include inductive predicate instances, `SAT` checking is not applicable. For `UNSAT` checking, $\mathtt{S2SAT}_{\mathtt{SL}}$ proves that Δ_{32} is unsatisfiable (its unsatisfiable cores are underlined as above); and shows that Δ_{33} can be linked back to Δ_0 (i.e., subsumed by Δ_0). The latter is shown based on some weakening and substitution principles (see Sect. 4.2). In particular: (i) Substituting Δ_{33} with $\theta{=}[n_2/n,x_1/x,n/n_1,x/r_2]$ such that predicate instances in the substituted formula, i.e., Δ_{33_a}, and Δ_0 are identical; as such, Δ_{33_a} is computed as below.

$$\Delta_{33_a}{\equiv}x_1{\mapsto}node(n_2,x){*}ll(n,x)^0_1{*}test(x,r_1)^1_1{\wedge}n_2{\geq}0{\wedge}r_1{=}0{\wedge}n_2{\neq}0{\wedge}n{=}n_2{-}1$$
$$\quad{\wedge}x_1{\neq}\mathtt{null}{\wedge}n_2{\geq}0$$

(ii) subtracting identical inductive predicates between Δ_{33_a} and Δ_0; (iii) weakening the remainder of Δ_{33_a} (i.e., $x_1{\mapsto}node(n_2,x)$ is eliminated); (iv) checking validity of the implication between pure of the remainder of Δ_{33_a} with the pure part of the remainder of Δ_0, i.e., $n_2{\geq}0{\wedge}r_1{=}0{\wedge}n_2{\neq}0{\wedge}n{=}n_2{-}1{\wedge}$ $x_1{\neq}\mathtt{null}{\wedge}n_2{\geq}0 \implies n{\geq}0{\wedge}r_1{=}0$. The back-link between Δ_{33} and Δ_0 establishes a cyclic proof which then proves Δ_0 is unsatisfiable.

Algorithm 1. `S2SAT` Procedure.

```
input  : $\lambda^{ind}$
output: SAT or UNSAT
1  $i{\leftarrow}0$; $\mathcal{T}_0{\leftarrow}\{\lambda^{ind}\}$ ;                          /* initialize */
2  while true do
3      (is_sat,$\mathcal{T}_i$) ← UA_test($\mathcal{T}_i$) ;                      /* check SAT */
4      if is_sat then return SAT ;                                   /* SAT */
5      else
6          $\mathcal{T}_i$←OA_test($\mathcal{T}_i$) ;                             /* prune UNSAT */
7          $\mathcal{T}_i$←link_back($\mathcal{T}_i$) ;                       /* detect fixpoint */
8          if is_closed($\mathcal{T}_i$) then return UNSAT;                     /* UNSAT */
9          else
10             $\lambda^{ind}{}_i$←choose_bfs($\mathcal{T}_i$) ;                /* choose an open leaf */
11             $i{\leftarrow}i{+}1$ ;
12             $\mathcal{T}_i$←unfold($\lambda^{ind}{}_i$);
13         end
14     end
15 end
```

3 S2SAT Algorithm

In this section, we present S2SAT, a procedure for checking satisfiability of formula with inductive predicates. We start by defining our target formulas. Let $\mathcal{L}$ be a *base theory* (logic) with the following properties: (i) $\mathcal{L}$ is closed under propositional combination and supports boolean variables; (ii) there exists a complete decision procedure for $\mathcal{L}$. Let $\mathcal{L}^{ind}$ be the extension of $\mathcal{L}$ with inductive predicate instances defined in a system with a set of predicates $\mathcal{P}{=}\{P_1, ..., P_k\}$. Each predicate may be annotated with a *sound* invariant. We use λ to denote a formula in $\mathcal{L}$ and λ^{ind} to denote a formula in the extended theory. Semantically, $\lambda^{ind}{\equiv}\bigvee_{i=0}^{n}\lambda_i,\ n{\geq}0$.

S2SAT is presented in Algorithm 1. S2SAT takes a formula λ^{ind} as input, systematically enumerates disjuncts λ_i and can produce two possible outcomes if it terminates: SAT with a satisfiable formula λ_i or UNSAT with a proof. We remark that non-termination is classified as UNKNOWN.

S2SAT maintains a set of open leaves of the unfolding tree $\mathcal{T}_i$ that is derived from λ^{ind}. In each iteration, S2SAT selects and unfolds an open leaf so as either to include more reachable base formulas (with the hope to produce a SAT answer), or to refine inductive formulas (with the hope to produce an UNSAT answer). Specially, in each iteration, S2SAT checks whether the formula is SAT at line 3; whether it is UNSAT at line 6; whether a fixpoint can be established at line 7. Function UA_test searches for a satisfiable *base* disjunct (i.e., is_sat is set to true). Simultaneously, it marks all unsatisfiable base disjuncts *closed*. Next, function OA_test uses predicate invariants to over-approximate open leaves of $\mathcal{T}_i$, and marks those with an *unsatisfiable* over-approximation closed. After that, function link_back attempts to link remaining open leaves back to interior nodes so as to form a fixpoint (i.e., a (partial) pre-proof for induction proving). The leaves which have been linked back are also marked as closed. Whenever all leaves are closed, S2SAT decides λ^{ind} as UNSAT (line 8). Otherwise, the choose_bfs (line 10) chooses an *open* leave in breadth-first manner for unfolding.

Procedure link_back takes the unfolding tree $\mathcal{T}_i$ as input and checks whether each open leaf $\lambda^{ind^{bud}}{\in}\mathcal{T}_i$ matches with one interior node $\lambda^{ind^{comp}}$ in $\mathcal{T}_i$ via a *matching function* $\mathtt{f_{fix}}$. $\mathtt{f_{fix}}$ is based on weakening and substitution principles [13]. Intuitively, $\mathtt{f_{fix}}$ detects the case of (i) the unfolding goes forever if we keep unfolding $\lambda^{ind^{bud}}$; and (ii) $\lambda^{ind^{bud}}$ has no model when $\lambda^{ind^{comp}}$ has no model. If $\mathtt{f_{fix}}(\lambda^{ind^{bud}}, \Delta^{comp}){=}\mathtt{true}$, Δ^{bud} is marked closed.

Our procedure systematically enumerates all disjuncts derived from a given inductive formula, so it is terminating for SAT. However, it may not be terminating for UNSAT with those undecidable augmented logic. In the next paragraph, we discuss the soundness of the algorithm.

Soundness. When S2SAT terminates, there are the following three cases.

– (case A) S2SAT produces SAT with a base satisfiable ${\lambda^{ind}}_i$;

– (case B) `S2SAT` produces `UNSAT` with a proof that all leaves of $\mathcal{T}_i$ are unsatisfiable;
– (case C) `S2SAT` produces `UNSAT` with a fixpoint: a proof that some leaves of $\mathcal{T}_i$ are unsatisfiable and the remaining leaves are linked back.

Under the assumption that $\mathcal{L}$ is both sound and complete, case A can be shown to be sound straightforwardly. Soundness of case B immediately follows the soundness of `OA_test`. In the following, we describe the cyclic proof instantiation of `link_back` for fixpoint detection and prove the soundness of case C.

We use CYCLIC to denote the cyclic proof for entailment procedure adapted from [13]. The following definitions are adapted from their analogues of CYCLIC.

Definition 1 (Pre-Proof). *A* pre-proof *derived for a formula λ^{ind} is a pair ($\mathcal{T}_i$, $\mathcal{L}$) where $\mathcal{T}_i$ is an unfolding tree whose root labelled by λ^{ind} and $\mathcal{L}$ is a back-link function assigning every open leaf ${\lambda^{ind}}_l$ of $\mathcal{T}_i$ to an interior node ${\lambda^{ind}}_c = \mathcal{L}({\lambda^{ind}}_l)$ such that there exists some substitution θ i.e., ${\lambda^{ind}}_c = {\lambda^{ind}}_l[\theta]$. ${\lambda^{ind}}_l$ is referred as a bud and ${\lambda^{ind}}_c$ is referred as its companion.*

A *path* in a pre-proof is a sequence of nodes $({\lambda^{ind}}_i)_{i\geq 0}$.

Definition 2 (Trace). *Let $({\lambda^{ind}}_i)_{i\geq 0}$ be a path in a pre-proof $\mathcal{PP}$. A trace following $({\lambda^{ind}}_i)_{i\geq 0}$ is a sequence $(\alpha_i)_{i\geq 0}$ such that, for all i≥0, α_i is a predicate instance $\mathtt{P}(\bar{t})$ in the formula ${\lambda^{ind}}_i$, and either:*

1. *α_{i+1} is the subformula according to $\mathtt{P}(\bar{t})$ occurrence in ${\lambda^{ind}}_{i+1}$, or*
2. *${\lambda^{ind}}_i[\bar{t}/\bar{v}]$ where ${\lambda^{ind}}_i$ is branches of inductive predicate $\mathtt{P}(\bar{v})$. i is a progressing point of the trace.*

To ensure that pre-proofs correspond to sound proofs, a global *soundness condition* must be imposed on such pre-proofs as follows.

Definition 3 (Cyclic Proof). *A pre-proof is a cyclic proof if, for every infinite path $({\lambda^{ind}}_i)_{i\geq 0}$, there is a tail of the path p=$({\lambda^{ind}}_i)_{i\geq n}$ such that there is an infinitely progressing trace following p.*

Theorem 1 (Soundness). *If there is a cyclic proof of ${\lambda^{ind}}_0$, ${\lambda^{ind}}_0$ is* `UNSAT`*.*

Proof. We reduce our cyclic proof problem for satisfiability to the cyclic proof problem for entailment check, i.e., ${\lambda^{ind}}_0 \vdash$ `false` of CYCLIC. Assume there is a cyclic proof $\mathcal{PP}$ of ${\lambda^{ind}}_0$. From $\mathcal{PP}$ we construct the pre-proof $\mathcal{PP}_{\vdash}$ for the sequent ${\lambda^{ind}}_0 \vdash$ `false` as follows. For each node $({\lambda^{ind}}_i)_{i\geq 0}$ in $\mathcal{PP}$, we replace the formula ${\lambda^{ind}}_i$ by the sequent ${\lambda^{ind}}_i \vdash$ `false`. Since $\mathcal{PP}$ is a cyclic proof, it follows that for every infinite path $({\lambda^{ind}}_i)_{i\geq 0}$, there is a tail of the path, p=$({\lambda^{ind}}_i)_{i\geq n}$, such that there is an infinitely progressing trace following p (Definition 3). Since formulas in [13] are only traced through the LHS of the sequent and not its RHS, it is implied that for every infinite path $({\lambda^{ind}}_i\vdash$`false`$)_{i\geq 0}$, there is a tail of the path, p=$({\lambda^{ind}}_i \vdash$ `false`$)_{i\geq n}$, such that there is an infinitely progressing trace following p. Thus, $\mathcal{PP}_{\vdash}$ is a cyclic proof (Definition 3 of [13]). As such ${\lambda^{ind}}_0 \models$`false` (Theorem 6 of [13]). In other words, ${\lambda^{ind}}_0$ is `UNSAT`. □

To sum up, to implement a sound cyclic proof system besides the matching function, a global *soundness condition* must be established on pre-proofs to guarantee well-foundedness of all reasoning.

4 Separation Logic Instantiation of S2SAT

In this section, to explicitly handle heap-manipulating programs, we propose a separation logic instantiation of `S2SAT`, called $\texttt{S2SAT}_{\texttt{SL}}$. We start by presenting `SLPA`, a fragment of separation logic with inductive predicates and arithmetic.

4.1 A Fragment of Separation Logic

Syntax. The syntax of `SLPA` formulas is presented in Fig. 3. We use $\bar{x}$ to denote a sequence (e.g., $\bar{v}$ for sequence of variables), and x_i to denote the i^{th} element. Whenever possible, we discard f_i of the points-to predicate and use its short form as $x{\mapsto}c(v_i)$. 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. All free variables are implicitly universally quantified at the outermost level. To express different scenarios for shape predicates, the fragment supports disjunction Φ over formulas. Each predicate instance is of the form $\texttt{P}(\bar{v})^o_u$ where o and u are labels used for context- and flow- sensitive unfolding. In particular, o captures the sequence number and u is the number of unfolding. For simplicity, we occasionally omit these two numbers if there is no ambiguity. A formula Δ is a *base formula* if it does not have any user-defined predicate instances. Otherwise, Δ is an *inductive formula.*

User-Defined Predicate. A user-defined predicate `P` is of the following general form

$$\texttt{pred}\ \texttt{P}(\bar{t}) \equiv \bigvee_{i=1}^{n} (\exists \bar{w}_i \cdot\ \Delta_i \ \mid\ \pi_i^b) \quad \overline{inv}\colon \pi;$$

Formula	$\Phi ::= \Delta \mid \Phi_1 \vee \Phi_2$	$\Delta ::= \exists \bar{v} \cdot\ (\kappa{\wedge}\pi)$
Spatial formula	$\kappa ::= \texttt{emp} \mid x{\mapsto}c(f_i{:}v_i) \mid \texttt{P}(\bar{v})^o_u \mid \kappa_1{*}\kappa_2$	
Pure formula	$\pi ::= \pi_1{\wedge}\pi_2 \mid b \mid \alpha \mid \phi$	
Ptr (Dis)Equality	$\alpha ::= v_1{=}v_2 \mid v{=}\texttt{null} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{null} \mid \alpha_1{\wedge}\alpha_2$	
Presburger arith.	$\phi ::= i \mid \exists v \cdot\ \phi \mid \neg\phi \mid \phi_1{\wedge}\phi_2 \mid \phi_1{\vee}\phi_2$	
Boolean formula	$b ::= \texttt{true} \mid \texttt{false} \mid v \mid b_1{=}b_2$	
Linear arithmetic	$i ::= a_1{=}a_2 \mid a_1{\leq}a_2$	
	$a ::= k^{\texttt{int}} \mid v \mid k^{\texttt{int}}{\times}a \mid a_1{+}a_2 \mid -a \mid max(a_1{,}a_2) \mid min(a_1{,}a_2)$	

$\texttt{P} \in \mathcal{P} \quad c \in \mathit{Node} \quad f_i \in \mathit{Fields} \quad v{,}v_i{,}x{,}y{,}\texttt{res}{,}\texttt{res}' \in \mathit{Var} \quad \bar{v} \equiv v_1, \ldots, v_n$

Fig. 3. Syntax.

where $\mathtt{P}$ is predicate name; $\bar{t}$ is a set of formal parameters; and $\exists \bar{w}_i \cdot\ \Delta_i$ (i $\in$ *1*...n) is a branch. Each branch is optionally annotated with a sound invariant π_i^b which is a pure formula that over-approximates the branch. π is an optionally sound *predicate invariant.* It must be a superset of all possible models of the predicate $\mathtt{P}$ via a pure constraint on stack. The default invariant of each inductive predicate is $\mathtt{true}$. For efficiency, we infer more precise invariants automatically (see Sect. 6). Inductive branches may be recursive. We assume that the recursion is direct, i.e., a recursive branch of predicate $\mathtt{P}$ includes at least one predicate instance $\mathtt{P}$. In each branch, we require that variables which are not formal parameters must be existentially quantified i.e., $\forall i \in 1...n{\cdot}FV(\Delta_i){=}\bar{t}$ and $\bar{w}_i{\cap}\bar{t}{=}\emptyset$ where $FV(\Delta)$ are all free variables in the formula Δ.

In the following, we apply `SLPA` to model two data structures: sorted lists (`sortll`) without an annotated invariant and AVL trees (`avl`) with annotated-invariant.

$$\begin{array}{l}\mathtt{pred\ sortll}(\mathtt{root},n,m) \equiv \mathtt{root}{\mapsto}node(m,\mathtt{null}) \wedge \mathtt{n{=}1}\\ \quad\vee\ \exists\, q,n_1,m_1{\cdot}\mathtt{root}{\mapsto}node(m,q) * \mathtt{sortll}(q,n_1,m_1){\wedge}n{=}n_1{+}1{\wedge}m{\leq}m_1\end{array}$$

$$\begin{array}{l}\mathtt{struct\ c_2\ \{\ c_2\ left;\ c_2\ right;\}}\ \textit{// data structure declaration}\\ \mathtt{pred\ avl(root,n,h)} \equiv \mathtt{emp} \wedge \mathtt{root{=}null} \wedge \mathtt{n{=}0} \wedge \mathtt{h{=}0}\ \mid \mathtt{root{=}null} \wedge \mathtt{n{=}0} \wedge \mathtt{h{=}0}\\ \quad\vee\ \exists\, \mathtt{l,r,n_1,n_2,h_1,h_2{\cdot}root{\mapsto}c_2(l,r)} * \mathtt{avl(l,n_1,h_1)} * \mathtt{avl(r,n_2,h_2)}\wedge\\ \qquad \mathtt{n{=}n_1{+}n_2{+}1} \wedge \mathtt{h{=}max(h_1,h_2){+}1} \wedge -\mathtt{1{\leq}h_1{-}h_2{\leq}1}\quad \mid \mathtt{root{\neq}null} \wedge \mathtt{n{>}0} \wedge \mathtt{h{>}0}\\ \quad \overline{inv}\text{: } n{\geq}0 \wedge h{\geq}0\end{array}$$

Semantics. In the following, we discuss the semantics of `SLPA`. 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:

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

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. 4. $dom(f)$ is the domain of function f; $h_1\#h_2$ denotes that heaps h_1 and h_2 are disjoint, i.e., $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$; and $h_1{\cdot}h_2$ denotes the union of two disjoint heaps. Inductive predicates are interpreted using the least model semantics [42]. Semantics of pure formulas depend on stack valuations; it is straightforward and omitted in Fig. 4, for simplicity.

4.2 Implementation of Separation Logic Instantiation

In the following, we describe how $\mathtt{S2SAT_{SL}}$ is realized. In particular, we show how the functions `UA_test`, `OA_test`, `unfold`, and `link_back` are implemented.

$$
\begin{array}{lll}
s,h \models \texttt{emp} & \texttt{iff} & h{=}\emptyset \\
s,h \models v{\mapsto}c(f_i : v_i) & \texttt{iff} & l{=}s(v), \mathrm{dom}(h){=}\{l \rightarrow r\} \text{ and } r(c,f_i){=}s(v_i) \\
s,h \models \mathtt{p}(\bar{v}) & \texttt{iff} & (s(\bar{v}),h) \in [\![\mathtt{p}(\bar{v})]\!] \\
s,h \models \kappa_1 * \kappa_2 & \texttt{iff} & \exists h_1,h_2{\cdot}\ h_1\#h_2 \text{ and } h{=}h_1{\cdot}h_2 \text{ and} \\
 & & s,h_1 \models \kappa_1 \text{ and } s,h_2 \models \kappa_2 \\
s,h \models \texttt{true} & \texttt{iff} & \text{always} \\
s,h \models \texttt{false} & \texttt{iff} & \text{never} \\
s,h \models \exists v_1,...,v_n{\cdot}(\kappa{\wedge}\pi) & \texttt{iff} & \exists \alpha_1...\alpha_n \cdot s(v_1{\mapsto}\alpha_1{*}...{*}v_n{\mapsto}\alpha_n), h \models \kappa \\
 & & \text{and } s(v_1{\mapsto}\alpha_1{*}...{*}v_n{\mapsto}\alpha_n) \models \pi \\
s,h \models \Phi_1 \vee \Phi_2 & \texttt{iff} & s,h \models \Phi_1 \text{ or } s,h \models \Phi_2
\end{array}
$$

Fig. 4. Semantics.

Deciding Separation Logic Formula. Given an SLPA formula, the functions UA_test and OA_test in $\mathtt{S2SAT_{SL}}$ work similarly, by reducing the formula to a first-order formula systematically and deciding the first-order formula. In the following, we define a function called **eXPure**, which transforms a separation logic formula into a first-order formula. **eXPure** is defined over the symbolic heap as follows:

$$
\begin{array}{l}
\mathbf{eXPure}(\exists\bar{w}{\cdot}\ x_1{\mapsto}c_1(\bar{v}_1){*}...{*}x_n{\mapsto}c_n(\bar{v}_n)\ *\ \mathtt{P_1}(\bar{t}_1){*}...{*}\mathtt{P_m}(\bar{t}_m)\ \wedge\ \pi) \equiv \\
\quad \exists\ \bar{w}{\cdot}\ \bigwedge\{x_i{\neq}\mathtt{null} \mid i{\in}1...n\} \wedge \bigwedge\{x_i{\neq}x_j \mid i,j{\in}1...n \text{ and } i{\neq}j\} \wedge \\
\quad \bigwedge\{\mathtt{inv}(\mathcal{P},\mathtt{P_j},\bar{\mathtt{t_j}}) \mid j \in 1...m\} \wedge \\
\quad \pi
\end{array}
$$

where the reduction at the first line (after $\equiv$) is for points-to predicates, and the second line is for user-defined predicates. The auxiliary function $\mathtt{inv}(\mathcal{P},\mathtt{P},\bar{\mathtt{v}})$ returns the invariant of the predicate P with a proper renaming.

Next, the auxiliary procedure $\mathtt{sat_P}(\exists\bar{w}{\cdot}\ \pi)$ takes a quantified first-order formula as input. It preprocesses the formula and then invokes an SMT solver to solve it. The preprocessing consists of two steps. First, the existential quantifiers $\bar{w}$ are eliminated through a projection $\Pi(\pi,\bar{w})$. Second, remaining existential quantifiers are skolemized and null is substituted by special number (i.e., zero). The preprocessed formulas are of the form of linear arithmetic with free function symbols. These formulas may contain existential ($\exists$) and universal ($\forall$) quantifiers but no $\exists\forall$ alternation. Hence, they are naively supported by SMT solvers.

Deriving Unfolding Tree. Next, we describe how function unfold works in $\mathtt{S2SAT_{SL}}$. Given a formula, unfold selects one predicate instance for unfolding as follows.

$$
\frac{\pi_c{\equiv}\mathbf{eXPure}(\kappa * \mathtt{P}(\bar{v}) \wedge \pi) \qquad \Gamma_i{=}\mathtt{unfoldP}(\mathtt{P}(\bar{v})^o_u, \pi_c)}{\mathtt{unfold}(\exists\bar{w}_0{\cdot}\ \kappa{*}\mathtt{P}(\bar{v})^o_u{\wedge}\pi) \rightsquigarrow \{\exists\bar{w}_0{\cdot}\ \kappa{*}\Delta_i{\wedge}\pi \mid \Delta_i{\in}\Gamma_i\}}
$$

Predicate instances in κ are sorted by a pair of unfolding number and ordering number where the former has higher priority. The instance $\mathtt{P}(\bar{v})^o_u$ is selected if u is the smallest number of unfoldings and `o` is the smallest number among instances which have the same unfolding number `u`. The procedure `unfold` outputs a set of disjuncts which are combined from branches of the predicate `P` with the remainder $\kappa \wedge \pi$. At the middle, the predicate instance is unfolded by the procedure `unfoldP`. This auxiliary procedure $\texttt{unfoldP}(\texttt{P}(\bar{t})^o_u, \pi_c)$ unfolds the user-defined predicate `P` with actual parameter $\bar{t}$ under the context π_c. It outputs branches of the predicate `P` that are not inconsistent with the context. It is formalized as follows.

$$\frac{\pi_c^P \equiv \Pi(\pi_c, \bar{v}) \qquad (\bigvee_{i=1}^m (\exists \bar{w}_i \cdot\ \kappa_i \wedge \pi_i \mid \pi_i^b), \bar{t}) = \texttt{lookup}(\mathcal{P}, \texttt{P}) \qquad \bar{w}_i' = \mathit{fresh}(\bar{w}_i) \\ (\bar{v}', \pi_{eq}) = \texttt{freshEQ}(\bar{v}) \qquad \rho_p = [\bar{v}'/\bar{t}] \qquad \rho_i^\exists = [\bar{w}_i'/\bar{w}_i] \qquad \rho_i = \rho_p \circ \rho_i^\exists}{\texttt{unfoldP}(\texttt{P}(\bar{v})^o_u, \pi_c) \leadsto \{\exists \bar{w}_i' \cdot\ [\rho_i]\kappa_i \wedge [\rho_i]\pi_i \wedge \pi_{eq} \mid \mathbf{sat_p}(\pi_c^P \wedge [\rho_i]\pi_i^b \wedge \pi_{eq}) \neq \texttt{unsat}, i \in 1...m\}}$$

In the first line, the procedure looks up the definition of `P` and refreshes the existential quantifiers (using the function *fresh*(...)). In the second line, formal parameters are substituted by the corresponding actual arguments. Finally, the substituted definition is combined and pruned as shown in the RHS of $\leadsto$. Function $\texttt{freshEQ}(\bar{v})$ above refreshes the sequence of variables $\bar{v}$ and produces the equality constraints π_{eq} between the old and new ones, i.e. $\pi_{eq} \equiv \bigwedge v_i = v_i'$. Let $\texttt{Q}(\bar{t})^{o_l}_{\_}$ denote a predicate instance of the derived κ_i, its unfolding number is set to `u+1` if its corresponding branch Δ_i is recursive. Otherwise, it is `u`. Its sequence number is set to $o_l + o$.

The branch invariant is used as a *necessary condition* to unfold a branch. The formalism underlying the pruning process is as follows: given a context Δ_c with its over-approximation π_c and a branch Δ_i with its over-approximation π_i^b, if $\pi_c \wedge \pi_i^b$ is unsatisfiable, so is $\Delta_c * \Delta_i$. Similar to the specialization calculus [15], our unfolding mechanism also prunes infeasible disjuncts while unfolding user-defined predicates. However, the specialization calculus performs exhaustive pruning with multiple unfolding that may be highly costly and redundant compared with our one-step unfolding.

Detecting Cyclic Proof. In the following, we implement the *matching function* $\texttt{f}_{\texttt{cyclic}}$, an instantiation of $\texttt{f}_{\texttt{fix}}$, to form a cyclic proof for fixpoint detection. $\texttt{f}_{\texttt{cyclic}}$ checks whether there exists a *well-founded* ordering relation R between Δ^{comp} and Δ^{bud} so as to form an *infinite* path following the path between these two nodes. If Δ^{bud} matches with Δ^{comp}, Δ^{bud} is marked as closed. For global infinitary soundness, $\texttt{f}_{\texttt{cyclic}}$ only considers those Δ^{bud} and Δ^{comp} of the restricted form as: $\Delta^{comp} \equiv \Delta_{b_1} * \texttt{P}_1(\bar{t}_1)^0_m * ... * \texttt{P}_\texttt{i}(\bar{t}_i)^i_m$, and $\Delta^{bud} \equiv \Delta_{b_2} * \texttt{P}_1(\bar{t}_1')^0_n * ... * \texttt{P}_\texttt{k}(\bar{t}_k')^k_n$, where $\texttt{k} \geq \texttt{i}$, $\texttt{n} > \texttt{m}$, Δ_{b_1} and Δ_{b_2} are base formulas.

Like [13], $\texttt{f}_{\texttt{cyclic}}$ is implemented using the weakening and substitution principle. In particular, it looks for a substitution θ s.t. $\Delta^{bud}\theta \implies \Delta^{comp}$. $\texttt{f}_{\texttt{cyclic}}(\Delta^{bud}, \Delta^{comp})$ is formalized as the procedure $\Delta^{bud} \vdash_{lb} \Delta^{comp}$ whose rules are presented in Fig. 5. These rules are applied as follows.

– First, existential variables are refreshed ($\underline{[\mathbf{EX{-}L}]}$, $\underline{[\mathbf{EX{-}R}]}$ rules).

$$\overset{[\underline{\textbf{EX−L}}]}{\dfrac{\bar{w}'{=}\mathit{fresh}\ \bar{w} \quad \Delta_1[\bar{w}'/\bar{w}] \vdash_{lb} \Delta_2}{\exists \bar{w}{\cdot}\Delta_1 \vdash_{lb} \Delta_2}} \qquad \overset{[\underline{\textbf{EX−R}}]}{\dfrac{\bar{w}'{=}\mathit{fresh}\ \bar{w} \quad \Delta_1 \vdash_{lb} \Delta_2[\bar{w}'/\bar{w}]}{\Delta_1 \vdash_{lb} \exists \bar{w}{\cdot}\Delta_2}} \qquad \overset{[\underline{\textbf{PURE}}]}{\dfrac{\pi_1 \implies \pi_2}{\pi_1 \vdash_{lb} \pi_2}}$$

$$\overset{[\underline{\textbf{SUBST}}]}{\dfrac{s{\in}\bar{v} \quad t{\in}\bar{w} \quad \exists R{\cdots}R(s,t) \quad t'{=}\mathit{fresh}\ t \quad (\kappa_1{*}\mathtt{P}(\bar{v}){\wedge}\pi_1)[t'/t;t/s] \vdash_{lb} \kappa_2{*}\mathtt{P}(\bar{w}){\wedge}\pi_2}{\kappa_1{*}\mathtt{P}(\bar{v}){\wedge}\pi_1 \vdash_{lb} \kappa_2{*}\mathtt{P}(\bar{w}){\wedge}\pi_2}}$$

$$\overset{[\underline{\textbf{PRED−MATCH}}]}{\dfrac{(\kappa_1{\wedge}\pi_1)[\bar{v}/\bar{w}] \vdash_{lb} \kappa_2{\wedge}\pi_2}{\kappa_1{*}\mathtt{P}(\bar{v}){\wedge}\pi_1 \vdash_{lb} \kappa_2{*}\mathtt{P}(\bar{w}){\wedge}\pi_2}} \qquad \overset{[\underline{\textbf{PRED−WEAKEN}}]}{\dfrac{\mathtt{P}(\bar{w}){\notin}\kappa_2 \quad \bar{v}{\cap}FV(\kappa_1{\wedge}\pi_1){=}\emptyset \quad \kappa_1{\wedge}\pi_1 \vdash_{lb} \kappa_2{\wedge}\pi_2}{\kappa_1{*}\mathtt{P}(\bar{v}){\wedge}\pi_1 \vdash_{lb} \kappa_2{\wedge}\pi_2}}$$

$$\overset{[\underline{\textbf{PTO−MATCH}}]}{\dfrac{(\kappa_1{\wedge}\pi_1){\wedge}[\bar{v}_1/\bar{v}_2] \vdash_{lb} \kappa_2{\wedge}\pi_2}{\kappa_1{*}v{\mapsto}c(\bar{v_1}){\wedge}\pi_1 \vdash_{lb} \kappa_2{*}v{\mapsto}c(\bar{v_2}){\wedge}\pi_2}} \qquad \overset{[\underline{\textbf{PTO−WEAKEN}}]}{\dfrac{v{\mapsto}c(\bar{w}){\notin}\kappa_1 \quad (\kappa_1{\wedge}\pi_1)[\bar{v}_1/\bar{v}_2] \vdash_{lb} \kappa_2{\wedge}\pi_2}{\kappa_1{*}v{\mapsto}c(v_1){\wedge}\pi_1 \vdash_{lb} \kappa_2{*}v{\mapsto}c(v_2){\wedge}\pi_2}}$$

Fig. 5. Rules for back-link.

– Second, *inductive* variables in Δ^{bud} are substituted ([**SUBST**] rule). This substitution is based on well-ordering relations R. Let $\mathtt{P}(t)^k_m$ be a predicate instance in Δ^{comp} and its corresponding subformula in Δ^{bud} be $R(s,t)$, then s, t are inductive variables. Two examples of well-founded relations R are structural induction for pointer types where $R(s,t)$ iff s is a subterm of t and natural number induction on integers where $R(s,t)$ iff $0{<}s{<}t$.
– Third, heaps are exhaustively matched ([**PRED−MATCH**] and [**PTO−MATCH**] rules) and weakened ([**PRED−WEAKEN**] and [**PTO−WEAKEN**] rules). Soundness of these rules directly follows from the frame rule [26,40].
– Last, back-link is decided via the implication between pure formulas ([**PURE**] rule).

5 Soundness and Termination of S2SAT$_{\text{SL}}$

In the following, we establish the correctness of S2SAT$_{\text{SL}}$.

5.1 Soundness

We show that (i) S2SAT$_{\text{SL}}$ is sound and complete for base formulas; and (ii) the functions UA_test, OA_test and link_back in S2SAT$_{\text{SL}}$ are sound. These two tasks rely on soundness and completeness of the function **eXPure** over base formulas, soundness of **eXPure** over inductive formulas, and soundness of the function $\mathtt{f}_{\mathtt{cyclic}}$.

Lemma 1 (Equiv-Satisfiable Reduction). *Let $\Delta{\equiv}\exists\bar{w}{\cdot}x_1{\mapsto}c_1(\bar{v}_1){*}...{*}x_n{\mapsto}c_n(\bar{v}_n){\wedge}\alpha{\wedge}\phi$ be a base formula. Δ is satisfiable iff* **eXPure**(Δ) *is satisfiable.*

The proof is based on structural induction on Δ.

Lemma 2 (Over-Approximated Reduction). *Given a formula Δ such that the invariants of user-defined predicates appearing in Δ are sound, then*

$$\forall s, h \cdot s, h {\models} \Delta \implies s {\models} \mathbf{eXPure}(\Delta)$$

In the following lemma, we consider the case $\Gamma{=}\{\}$ at line 8 of Algorithm 1.

Lemma 3. *Given a formula Δ_0 and the* matching function $\mathtt{f_{cyclic}}$ *as presented in the previous section, Δ_0 is* `UNSAT` *if $\Gamma{=}\{\}$ (line 8).*

To prove this Lemma, in [30] we show that there is a "trace manifold" which implies the global infinitary soundness (see [11], ch. 7) when a bud is linked back.

Theorem 2 (Soundness). *Given a formula Δ and a set of user-defined predicates $\mathcal{P}$,*

– *Δ is satisfiable if* $\mathtt{S2SAT_{SL}}$ *returns* `SAT`.
– *if* $\mathtt{S2SAT_{SL}}$ *terminates and returns* `UNSAT`*, Δ is unsatisfiable.*

While the soundness of `SAT` queries follows Lemma 1, the soundness of `UNSAT` queries follows Lemmas 2 and 3. As satisfiability for `SLPA` is undecidable [30, 31], there is no guarantee that $\mathtt{S2SAT_{SL}}$ terminates on all inputs. In the next subsection, we show that $\mathtt{S2SAT_{SL}}$ terminates for satisfiable formulas in `SLPA` and with certain restrictions on the fragment, $\mathtt{S2SAT_{SL}}$ always terminates.

5.2 Termination

Termination for SAT. In this paragraph, we show that $\mathtt{S2SAT_{SL}}$ always terminates when it decides a satisfiable formula. Given a satisfiable formula

$$\Delta{\equiv}\exists\bar{w}{\cdot}\ x_1{\mapsto}c_1(\bar{v}_1){*}...{*}x_n{\mapsto}c_n(\bar{v}_n)\ {*}\mathtt{P_0}(\bar{t}_0)^0_0{*}...{*}\mathtt{P_n}(\bar{t}_n)^n_0 \wedge \pi$$

There exists a satisfiable base formula Δ_k such as:

$$\Delta_k{\equiv}x_1{\mapsto}c_1(\bar{v}_1){*}...{*}x_n{\mapsto}c_n(\bar{v}_n)\ {*}\ \Delta^{\mathtt{P_0}}_{k_0}{*}...{*}\Delta^{\mathtt{P_n}}_{k_n} \wedge \pi$$

where Δ^P_k ($k{\geq}0$) denotes a base formula derived by unfolding the predicate `P` k times and then substituting all predicate instances `P` by `P`'s base branch. Let k_m be the maximal number among k_0,..,k_n. The breadth-first unfolding manner in the algorithm `S2SAT` ensures that $\mathtt{S2SAT_{SL}}$ identifies Δ_k before it encounters the following leaf:

$$y_1{\mapsto}c_1(\bar{t}_1){*}...{*}y_i{\mapsto}c_i(\bar{t}_i)\ {*}\ \mathtt{P_0}(\bar{t}_0)_{\bar{k}_m+1}{*}...{*}\mathtt{P_j}(\bar{t}_j)_{\bar{k}_m+1} \wedge \pi$$

We remark that the soundness of cyclic proof ensures that our `link_back` function only considers *infinitely* many unfolding traces. Thus, it never links *finite* many unfolding traces, i.e., traces connecting the root to satisfiable base leaves, like Δ_k.

Decidable Fragment. In the following, we describe universal $\mathtt{SLPA_{ind}}$, a fragment of SLPA, for which we prove that $\mathtt{S2SAT_{SL}}$ always terminates. Compared to SLPA, universal $\mathtt{SLPA_{ind}}$ restricts the set of inductive predicates $\mathcal{P}$ as well as the inputs of $\mathtt{S2SAT_{SL}}$.

Definition 4 ($\mathtt{SLPA_{ind}}$). *An inductive predicate* pred $\mathtt{P}(\bar{t}){\equiv}\Phi$ *is well-founded* $\mathtt{SLPA_{ind}}$ *if it has one induction case with N occurrences of* P*, and it has the shape as follows.*

$$\Phi \equiv \Phi_0 \vee \exists \bar{w}{\cdot}x_1{\mapsto}c_1(\bar{v}_1){*}...{*}x_n{\mapsto}c_n(\bar{v}_n){*}\mathtt{P}(\bar{w}_1){*}...{*}\mathtt{P}(\bar{w}_N){\wedge}\pi$$

where Φ_0 is disjunction of base formulas and the two following restrictions.

1. $\forall n{\in}1...N$ $\bar{w}_n{\subseteq}\bar{w}{\cup}\{\mathtt{null}\}$ *and $\bar{w}_n$ do not appear in the equalities of π,*
2. *if t_i is a numerical parameter and there exists a well-ordering relation R such that $R(s,t_i,w_{1_i},..,w_{m_i})$ ($1{\leq}m{\leq}N$) is a subformula of π, the following conditions hold.*
   - *t_i is constrained separately (i.e., there does not exist $j{\neq}i$ and a subformula ϕ of π such that $\{t_i,t_j\}{\subseteq}FV(\phi)$ or $\{t_i,w_{n_j}\}{\subseteq}FV(\phi)$ or $\{w_{m_i},w_{n_j}\}{\subseteq}FV(\phi)$ $\forall m,n{\in}1...N$, and*
   - *$\forall n{\in}1...N$, $\pi \implies t_i{>}w_{n_i}$ or $\pi \implies t_i{<}w_{n_i}$.*
   - *if $t_i{\in}FV(\Phi_0)$ then $\Phi_0 \implies t_i{=}k$, for some integer k*
   t_i is denoted as inductive *parameters.*

Restriction 1 guarantees that $\mathtt{f_{cyclic}}$ can soundly weaken the heap by discarding irrelevant points-to predicates and $N{-}1$ occurrences of P (when $N{\geq}2$) while it links back. Restriction 2 implies that $t_i{>}w_i{\geq}k_1$ or $t_i{<}w_i{\leq}k_2$ for some integer k_1, k_2. This ensures that leaf nodes of unfolding trees of an unsatisfiable input must be UNSAT or linked back.

The above $\mathtt{SLPA_{ind}}$ fragment is expressive enough to describe a range of data structures, e.g. sorted lists sortll, lists/trees with size properties, or even AVL trees avl.

Definition 5 (Universal $\mathtt{SLPA_{ind}}$). *Given a separation logic formula*

$$\Delta_0{\equiv}x_1{\mapsto}c_1(\bar{v}_1){*}...{*}x_n{\mapsto}c_n(\bar{v}_n){*}\mathtt{P_1}(\bar{t}_1){*}...{*}\mathtt{P_n}(\bar{t}_n){\wedge}\phi_0$$

Δ_0 is universal $\mathtt{SLPA_{ind}}$ *if all predicates* $\mathtt{P_1},..,\mathtt{P_n}$ *are well-founded* $\mathtt{SLPA_{ind}}$*, and if all $\bar{x}$ of free, arithmetical, inductive variables, with $\bar{x}{\subseteq}(\bar{t}_1{\cup}...{\cup}\bar{t}_n)$, ϕ_0 is a conjunction of $\phi_{0,i}$ where $\phi_{0,i}$ is either of the following form: (i)* true *; or (ii) $x_i{\geq}k_1$ for some integer k_1; or (iii) $x_i{\leq}k_2$ for some integer k_2.*

Theorem 3 (Termination). $\mathtt{S2SAT_{SL}}$ *terminates for universal* $\mathtt{SLPA_{ind}}$ *formulas.*

6 Sound Invariant Inference

In order to perform fully automatic verification without user-provided invariants, $\texttt{S2SAT}_{\texttt{SL}}$ supports automatic invariant inference. In this section, we describe invariant inference from user-defined predicates and predicate branches. While the former is used for over-approximation, the latter is used for context-sensitive predicate unfolding. To infer invariants for a set of user-defined predicates, we first build a dependency graph among the predicates. After that, we process each group of mutual dependent predicates following a bottom-up manner. For simplicity, we present the inference for one directly recursive predicate. The inference for a group of mutual inductive predicates is similar.

Inferring Predicate Invariant. Our invariant inference is based on the principle of second-order abduction [28,45]. Given the predicate P defined by m branches as $\texttt{P}(\bar{t}) \equiv \bigvee_{i=1}^{m} \Delta_i$, we assume a sound invariant of P as an unknown (second-order) variable $\texttt{I}(\bar{t})$. After that we prove the lemma $\texttt{P}(\bar{v}){\vdash}\texttt{I}(\bar{v})$ via induction; and simultaneously generate a set of pure relational assumptions using second-order abduction. The steps to prove the above lemma and generate a set of m relational assumptions over I are as follows.

1. Unfold LHS of the lemma to generate a set of m subgoals i.e. $\Delta_i[\bar{v}/\bar{t}]{\vdash}\texttt{I}(\bar{v})$ where $i \in 1...m$. The original lemma is taken as the induction hypothesis.
2. For each subgoal i, over-approximate its LHS to a pure formula π_i and form an assumption relation $\pi_i \implies \texttt{I}(\bar{v})$. There are two cases to compute π_i.
   – if Δ_i is a base formula, then $\pi_i{\equiv}\textbf{eXPure}(\Delta_i)$.
   – if Δ_i includes k instances P such that $\Delta_i{\equiv}\Delta_{rest_i}{*}\texttt{P}(\bar{v_1}){*}...{*}\texttt{P}(\bar{v_k})$, then we compute $\pi_{i_0}{\equiv}\textbf{eXPure}(\Delta_{rest_i})$, $\pi_{i_j}{\equiv}\texttt{I}(\bar{v_j})$, for all $j \in 1...k$, and $\pi_i{\equiv} \bigwedge_{j=1}^{k} \pi_{i_k}$.
3. Our system applies a least fixed point analysis to the set of gathered relational assumptions. We use the analyzer LFP presented in [45] to compute these invariants.

We illustrate this procedure to infer an invariant for sortll. First, our system introduces an unknown relation $\texttt{I}(\texttt{root},n,m)$. Second, it generates the below relational constraints.

$$\begin{array}{ll}\texttt{root}{\neq}\texttt{null}{\wedge}n{=}1 & \implies \texttt{I}(\texttt{root},n,m)\\ \texttt{root}{\neq}\texttt{null}{\wedge}\texttt{I}(Q,N_1,M_1){\wedge}n{=}N_1{+}1{\wedge}m{\leq}M_1 & \implies \texttt{I}(\texttt{root},n,m)\end{array}$$

Finally, it analyzes these two constraints and produces the following result:

$$\texttt{I}(\texttt{root},n,m){\equiv}\texttt{root}{\neq}\texttt{null}{\wedge}n{\geq}1$$

Lemma 4 (Sound Invariant Inference). *Given a predicate* $\texttt{P}(\bar{t}) \equiv \Phi$, *and* $\mathcal{R}$ *be a set of relational assumptions generated by the steps above. If* $\mathcal{R}$ *has a solution, i.e.,* $\texttt{I}(\bar{v}){\equiv}\pi$, *then we have* $\forall s, h \cdot s, h \models \texttt{P}(\bar{v})$, $s \models \pi$.

Proof Sketch: Soundness of Lemma 2 implies that for all $i \in 1...m$, π_i is an over-approximated abstraction of Δ_i. As such, the soundness of this lemma immediately follows from the soundness of second-order abduction [28,45]. □

Table 1. Exponential time and space satisfiability checks.

Succ-circuit (1–20)						Succ-rec (1–20)					
n	SLSAT	S2SAT$_{SL}$	n	SLSAT	S2SAT$_{SL}$	n	SLSAT	S2SAT$_{SL}$	n	SLSAT	S2SAT$_{SL}$
1	1 ms	21 ms	11	SO	37.46 s	1	0 ms	25 ms	11	1796.4 s	410.92 s
2	2 ms	23 ms	12	SO	170.53 s	2	1 ms	30 ms	12	TO	TO
3	27 ms	30 ms	13	SO	988.29 s	3	4 ms	33 ms	13	TO	TO
4	867 ms	34 ms	14	SO	TO	4	21 ms	39 ms	14	X	TO
5	30 s	0.05 s	15	SO	TO	5	134 ms	52 ms	15	X	TO
6	30 s	0.09 s	16	SO	TO	6	830 ms	76 ms	16	X	TO
7	SO	0.20 s	17	SO	TO	7	5.0 s	0.21 s	17	X	TO
8	SO	0.61 s	18	SO	TO	8	29.5 s	0.87 s	18	X	TO
9	SO	2.21 s	19	SO	TO	9	167.8 s	4.83 s	19	X	TO
10	SO	8.49 s	20	SO	TO	10	1065 s	45.28 s	20	X	TO

Inferring Branch Invariant. Given a predicate P defined by m branches as $\texttt{P}(\bar{t}) \equiv \bigvee_{i=1}^{m}(\exists \bar{w}_i \cdot \Delta_i)$ $\overline{inv}$: π, we compute invariants for each branch of P as $\Pi(\textbf{eXPure}(\Delta_i), \bar{w}_i)$ $\forall$ i=1...m. For example, with the invariant inferred for the predicate *sortll* as above, our system computes its branch invariants π_1^b for the base branch and π_2^b for the inductive branch as below.

$$\pi_1^b \equiv \Pi(\textbf{eXPure}(\texttt{root}{\mapsto}node(m,\texttt{null}) \wedge \texttt{n=1}), \{\}) \equiv \texttt{root}{\neq}\texttt{null} \wedge \texttt{n=1}$$
$$\pi_2^b \equiv \Pi(\textbf{eXPure}(\texttt{root}{\mapsto}node(m,q) * \texttt{sortll}(q,n_1,m_1) \wedge n{=}n_1{+}1 \wedge m{\leq}m_1), \{q,n_1,m_1\}) \equiv \texttt{root}{\neq}\texttt{null} \wedge n{\geq}1$$

Soundness of **eXPure** implies that the branch invariant over-approximates its branch.

7 Implementation and Evaluation

We have implemented the proposed solver S2SAT$_{SL}$ and a new interprocedural (top-down) program verification tool, called S2$_{td}$, which uses S2SAT$_{SL}$. We make use of Omega Calculator [38] to eliminate existential quantifiers, Z3 [19] as a back-end SMT solver, and FixCalc [37] to find closure form in inferring invariants for user-defined predicates.

In the following, we evaluate S2SAT$_{SL}$ and S2$_{td}$'s robustness and efficiency on a set of benchmarks from the software verification competition SV-COMP [7]. We also present an evaluation of S2SAT$_{SL}$ in compositional (modular) program verification with the HIP/S2 system [14,28] for a range of data structures.

7.1 Robustness and Efficiency

In [12], Brotherston *et al.* introduced a new and challenging set of satisfiability benchmarks discussed in Proposition 5.13 of [12]. In this Proposition, Brotherston *et al.* stated that there exists a family of predicates of size O(n) and that SLSAT runs in $\Omega(2^n)$ time and space regardless of search strategies. Since SLSAT relies on bottom-up and *context-insensitive* fixed point computation, it has to explore all possible models before answering a query. Their approach is designed for computing invariants of shape predicates rather than satisfiability checks. In contrast, $\texttt{S2SAT}_{\texttt{SL}}$ performs top-down and *context-sensitive* searches, as it is dedicated for satisfiability solving. Moreover, it prunes infeasible disjuncts, significantly reduces the search space, and provides better support for model discovery.

We conducted an experiment on comparing SLSAT's and $\texttt{S2SAT}_{\texttt{SL}}$'s performance on this set of benchmarks. The results are shown in Table 1. The size n of succ−circuit∗ (succ−rec∗) benchmarks expresses the breadth (depth, resp.) of dependency. This set of benchmarks is a part of the User-Defined Predicate Satisfiability (UDB_sat) suite of SL-COMP 2014 [41]. The output is either a definite answer (sat, unsat) with running time (in milliseconds (ms), or seconds (s)), or an error. In particular, SO denotes stack overflow; TO denotes timeout (i.e., tools run longer than 1800 s); and X denotes a fatal error. The experimental results show that $\texttt{S2SAT}_{\texttt{SL}}$ is much more robust and also more efficient than SLSAT. While $\texttt{S2SAT}_{\texttt{SL}}$ successfully solved 24 (out of 40) benchmarks, SLSAT was capable of handling 17 benchmarks. Furthermore, on 17 benchmarks that SLSAT discharged successfully, $\texttt{S2SAT}_{\texttt{SL}}$ outperforms SLSAT, i.e., about 6.75 (3126 s/462 s) times faster. As shown in the table, $\texttt{S2SAT}_{\texttt{SL}}$ ran with neither stack overflow nor fatal errors over all these challenging benchmarks.

Table 2. Experimental results on complex data structures.

Data structure (pure props)	#Query	#UNSAT	#SAT	Time
Singly llist (size)	666	75	591	1.25
Even llist (size)	139	125	14	2.40
Sorted llist (size, sorted)	217	21	196	0.91
Doubly llist (size)	452	50	402	2.07
Complete tree (size, minheight)	387	33	354	143.98
Heap trees (size, maxelem)	467	67	400	13.87
AVL (height, size, near-balanced)	881	64	817	84.82
BST (height, size, sorted)	341	34	307	2.28
RBT (size, height, color)	1741	217	1524	65.54
rose-tree	55	6	49	0.34
TLL	128	13	115	0.24
Bubble (size, sorted)	300	20	280	1.09
Quick sort (size, sorted)	225	29	196	2.33

7.2 Modular Verification with S2SAT$_{\text{SL}}$

In this subsection, we evaluate S2SAT$_{\text{SL}}$ in the context of modular program verification. S2SAT$_{\text{SL}}$ solver is integrated into the HIP/S2 [14,28,29] system to prune infeasible program paths in symbolic execution. Furthermore, S2SAT$_{\text{SL}}$ is also used by the entailment procedure SLEEK to discharge verification conditions (VC) generated. In particular, when SLEEK deduces a VC to the following form: $\Delta \vdash \texttt{emp}\wedge\pi_r$, the error calculus in SLEEK [29] invokes S2SAT$_{\text{SL}}$ to discharge the following queries: Δ and $\Delta\wedge\neg\pi_r$ for safety and $\Delta\wedge\pi_r$ for *must* errors. In experiments, we have extracted those VCs generated while HIP/S2 verified heap-manipulating programs.

We have evaluated S2SAT$_{\text{SL}}$ deciding the VCs discussed above. The experimental results are described in Table 2. Each line shows a test on one program. The first column lists data structures and their pure properties. rose-trees are trees with nodes that are allowed to have a variable number of children, stored as doubly-linked lists. TLL is a binary tree whose nodes point to their parents and all leaf nodes are linked as a singly-linked list. #Query is the number of satisfiability queries sent to S2SAT$_{\text{SL}}$ for each data structure. The next two columns report the outputs from S2SAT$_{\text{SL}}$. The last column shows the time (in seconds) taken by the S2SAT$_{\text{SL}}$ solver. In this experiment, S2SAT$_{\text{SL}}$ terminated on all queries. Furthermore it exactly decided all SAT and UNSAT queries. These experimental results affirm the correctness of our algorithm S2SAT$_{\text{SL}}$. They also show that S2SAT$_{\text{SL}}$ is expressive, effective, and can be integrated into program verification systems for discharging satisfiability problems of separation logic formulas.

7.3 Recursive Program Verification with S2SAT$_{\text{SL}}$

We have evaluated and compared our verification system S2$_{\text{td}}$ with state-of-the-art verification tools on a set of SV-COMP benchmarks[1]. The results are presented in Table 3. There are 102 recursive/loop programs taken from *Recursive* and *HeapReach* sub-categories in the benchmark; timeout is set to 180 s. In each program, there is at least one user-supplied assertion to model safety properties. The first column identities the subset of verification systems which competed in both the above sub-categories. The next three columns count the instances of correct safe (s$\surd$), correct error (e$\surd$) and unknown (e.g., timeout). The next two columns capture the number of false positives (s✗) and false negatives (e✗). We rank these tools based on their points. Following the SV-COMP competition, we gave +2 for one s$\surd$, +1 for one

Table 3. Experimental results on recursive programs.

Tool	#s$\surd$	#e$\surd$	#unk	#s✗	#e✗	Points	Mins
ESBMC [18]	38	40	21	0	3	20	53
UAutomizer [24]	17	23	62	0	0	57	23
SeaHorn [22]	48	45	5	4	0	77	26
CBMC [16]	33	39	29	1	0	89	90
Smack-corral [1]	33	37	28	0	0	103	105
S2$_{\text{td}}$	**41**	**45**	**16**	**0**	**0**	**127**	**25**

1 http://sv-comp.sosy-lab.org/2016/.

e$\surd$, 0 for unk, -16 for one s✗, and -32 for one e✗. The last column expresses the total time in minutes. The results show that the proposed verification approach is promising; indeed, our system is effective and efficient: it produces the best correctness with *zero* false answers within the nearly-shortest time.

8 Related Work

Close to our work is the SeaHorn verification system [22]. While SeaHorn relies on Z3-PDR to handle inductive predicates on non-heap domains, it is unclear (to us) how SeaHorn supports induction reasoning for heap-based programs (which is one contribution of our present work).

Our S2SAT satisfiability procedure is based on unfolding which is similar to the algorithm in the Leon system [43,44]. Leon, a verifier for functional programs, adds an unfolding mechanism for inductive predicates into complete theories. However, Leon only supports classic logic and not structural logic (i.e., separation logic). Neither does Leon support inductive reasoning. Furthermore, our system infers sound invariants for inductive predicates to facilitate over-approximation.

Our work is related to work on developing satisfiability solvers in separation logic. In the following, we summarize the development in this area. Smallfoot [5] has the first implemented decision procedure for a fragment of separation logic. This solver was originally customized to work with spatial formulas over list segments. Based on a fixed equality (disequality) constraint branches of the list segment, the proposals presented by [17,32] further enhanced decision procedure for this fragment with equality reasoning. They provided normalization rules with a graph technique [17] and a superposition calculus [32] to infer (dis)equality constraints on pointers and used these constraints to prune infeasible branches of predicate instances during unfolding. Although these proposals can decide the formula of that fragment in polynomial time, it is not easy to extend them to a fragment with general inductive predicates (i.e., the fragment SLPA). Decision procedures in [33–36] support decidable fragments of separation logic with inter-reachable data structures using SMT. Our proposal extends these procedures to those fragments with general inductively-defined predicates. Indeed, our decidable fragment can include more complex data structures, such as AVL trees.

$\texttt{S2SAT}_{\texttt{SL}}$ is closely related to the satisfiability solvers [12,25] which are capable of handling separation logic formulas with general user-defined predicates. Decision procedures [12,25] are able to handle predicates without pure properties. The former described a decidable fragment of user-defined predicates with bounded tree width. The problem of deciding separation logic formulas is then reduced to monadic second-order logic over graphs. The latter, SLSAT, decides formulas with user-defined predicates via a equi-satisfiable fixed point calculation. The main disadvantage of SLSAT is that it is currently restricted to the domain of pointer equality and disequality, so that it cannot be used to support predicates with pure properties from infinite abstract domains.

Using over-approximation in decision procedures is not new. For example, D'Silva *et al.* have recently made use of abstract domains inside satisfiability solvers [20,21]. In separation logic, satisfiability procedures in HIP/SLEEK [14] and Dryad [39] decide formulas via a sound reduction that over-approximates predicate instances. HIP/SLEEK and Dryad are capable of proving the validity of a wide range of expressive formulas with arbitrary predicates. However, expressivity comes with cost; as these procedures are incomplete, and they do not address the satisfiability problem. We believe that S2SAT can be integrated into these systems to improve upon these two shortcomings.

9 Conclusion and Future Work

We have presented a satisfiability procedure for an expressive fragment of separation logic. Given a formula, our procedure examines both under-approximation (so as to prove SAT) and over-approximation (so as to prove UNSAT). Our procedure was strengthened with invariant generation and cyclic proof detection. We have also implemented a solver and a new verification system for heap-manipulating programs. We have evaluated them on a range of competition problems with either complex heap usage patterns or exponential complexity of time and space.

For future work, we might investigate S2SAT-based decision procedures for other complete theories (i.e., Presburger, string, bag/set) augmented with inductive predicates. We would also study a more general decidable fragment of separation logic by relaxing the restrictions for termination. Finally, we would like to improve $\mathtt{S2}_{\mathtt{td}}$ for array, string and pointer arithmetic reasoning as well as witness generation for erroneous programs.

Acknowledgements. We wish to thank Christopher M. Poskitt for his helpful comments on the manuscript. Quang Loc and Jun Sun are partially supported by NRF grant RGNRF1501 and Wei-Ngan by NRF grant NRF2014NCR-NCR001-040.

References

1. Haran, A., Carter, M., Emmi, M., Lal, A., Qadeer, S., Rakamarić, Z.: SMACK+corral: a modular verifier. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 451–454. Springer, Heidelberg (2015)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., M. Leino, K.R.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011)
4. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: ISSTA, pp. 3–14. ACM, New York (2008)
5. Berdine, J., Calcagno, C., W.O'Hearn, P.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 97–109. Springer, Heidelberg (2004)

6. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
7. Beyer, D.: Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 887–904. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49674-9_55
8. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, pp. 25–32 (2009)
9. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W., Beklemishev, L.D., Beklemishev, L.D. (eds.) Gurevich Festschrift II 2015. LNCS, vol. 9300, pp. 24–51. Springer, Heidelberg (2015). doi:10.1007/978-3-319-23534-9_2
10. Bjørner, N., McMillan, K.L., Rybalchenko, A.: Program verification as satisfiability modulo theories. In: SMT, pp. 3–11 (2012)
11. Brotherston. J.: Sequent calculus proof systems for inductive definitions. Ph.D. thesis, University of Edinburgh, November 2006
12. Brotherston, J., Fuhs, C., Pérez, J.A.N., Gorogiannis, N.: A decision procedure for satisfiability in separation logic with inductive predicates. In: CSL-LICS 2014, pp. 25:1–25:10. ACM, New York (2014)
13. Brotherston, J., Gorogiannis, N., Petersen, R.L.: A generic cyclic theorem prover. In: Jhala, R., Igarashi, A. (eds.) APLAS 2012. LNCS, vol. 7705, pp. 350–367. Springer, Heidelberg (2012)
14. Chin, W., David, C., Nguyen, H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. SCP **77**(9), 1006–1036 (2012)
15. Chin, W.-N., Gherghina, C., Voicu, R., Le, Q.L., Craciun, F., Qin, S.: A specialization calculus for pruning disjunctive predicates to support verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 293–309. Springer, Heidelberg (2011)
16. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
17. Cook, B., Haase, C., Ouaknine, J., Parkinson, M., Worrell, J.: Tractable reasoning in a fragment of separation logic. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 235–249. Springer, Heidelberg (2011)
18. Cordeiro, L., Fischer, B.: Verifying multi-threaded software using smt-based context-bounded model checking. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 331–340. ACM, New York (2011)
19. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
20. D'Silva, V., Haller, L., Kroening, D.: Satisfiability solvers are static analysers. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 317–333. Springer, Heidelberg (2012)
21. D'Silva, V., Haller, L., Kroening, D.: Abstract satisfaction. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, pp. 139–150. ACM, New York (2014)

22. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Heidelberg (2015)
23. Haase, C., Ishtiaq, S., Ouaknine, J., Parkinson, M.J.: SeLoger: a tool for graph-based reasoning in separation logic. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 790–795. Springer, Heidelberg (2013)
24. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 36–52. Springer, Heidelberg (2013)
25. Iosif, R., Rogalewicz, A., Simacek, J.: The tree width of separation logic with recursive definitions. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 21–38. Springer, Heidelberg (2013)
26. Ishtiaq, S., O'Hearn, P.: BI as an assertion language for mutable data structures. In: ACM POPL, pp. 14–26, London, January 2001
27. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: PLDI, pp. 437–446. ACM, New York (2011)
28. Le, Q.L., Gherghina, C., Qin, S., Chin, W.-N.: Shape analysis via second-order bi-abduction. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 52–68. Springer, Heidelberg (2014)
29. Le, Q.L., Sharma, A., Craciun, F., Chin, W.-N.: Towards complete specifications with an error calculus. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 291–306. Springer, Heidelberg (2013)
30. Le, Q.L., Sun, J., Chin, W.-N.: Satisfiability modula heap-based programs. Technical report (2016). http://loc.bitbucket.org/papers/satsl-cav16.pdf
31. Makoto, T., Le, Q.L., Chin, W.-N.: Presburger arithmetic and separation logic with inductive definitions. Technical report, May 2016
32. Navarro Pérez, J.A., Rybalchenko, A.: Separation logic + superposition calculus = heap theorem prover. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 556–566 (2011)
33. Navarro Pérez, J.A., Rybalchenko, A.: Separation logic modulo theories. In: Shan, C. (ed.) APLAS 2013. LNCS, vol. 8301, pp. 90–106. Springer, Heidelberg (2013)
34. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 773–789. Springer, Heidelberg (2013)
35. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 711–728. Springer, Heidelberg (2014)
36. Piskac, R., Wies, T., Zufferey, D., Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 711–728. Springer, Heidelberg (2014)
37. Popeea, C., Chin, W.-N.: Inferring disjunctive postconditions. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 331–345. Springer, Heidelberg (2008)
38. Pugh, W.: The omega test: a fast practical integer programming algorithm for dependence analysis. Commun. ACM **8**, 102–114 (1992)
39. Qiu, X., Garg, P., Ştefănescu, A., Madhusudan, P.: Natural proofs for structure, data, and separation. In: PLDI, pp. 231–242. ACM, New York (2013)
40. Reynolds, J., Logic, S.: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, pp. 55–74 (2002)

41. Sighireanu, M., Cok, D.R.: Report on SL-COMP 2014. In: JSAT (2016)
42. Sims, É.-J.: Extending separation logic with fixpoints and postponed substitution. Theor. Comput. Sci. **351**(2), 258–275 (2006)
43. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, pp. 199–210. ACM, New York (2010)
44. Suter, P., Köksal, A.S., Kuncak, V.: Satisfiability modulo recursive programs. In: Yahav, E. (ed.) Static Analysis. LNCS, vol. 6887, pp. 298–315. Springer, Heidelberg (2011)
45. Trinh, M.-T., Le, Q.L., David, C., Chin, W.-N.: Bi-abduction with pure properties for specification inference. In: Shan, C. (ed.) APLAS 2013. LNCS, vol. 8301, pp. 107–123. Springer, Heidelberg (2013)