Automating Separation Logic with Trees and Data

Ruzica Piskac[1], Thomas Wies[2⋆], and Damien Zufferey[3⋆⋆]

[1]Yale University [2]New York University [3]MIT CSAIL

Abstract. Separation logic (SL) is a widely used formalism for verifying heap manipulating programs. Existing SL solvers focus on decidable fragments for list-like structures. More complex data structures such as trees are typically unsupported in implementations, or handled by incomplete heuristics. While complete decision procedures for reasoning about trees have been proposed, these procedures suffer from high complexity, or make global assumptions about the heap that contradict the separation logic philosophy of local reasoning. In this paper, we present a fragment of classical first-order logic for local reasoning about tree-like data structures. The logic is decidable in NP and the decision procedure allows for combinations with other decidable first-order theories for reasoning about data. Such extensions are essential for proving functional correctness properties. We have implemented our decision procedure and, building on earlier work on translating SL proof obligations into classical logic, integrated it into an SL-based verification tool. We successfully used the tool to verify functional correctness of tree-based data structure implementations.

1 Introduction

Separation logic (SL) [30] has proved useful for building scalable verification tools for heap-manipulating programs that put no or very little annotation burden on the user [2,5,6,12,15,40]. The high degree of automation of these tools relies on solvers for checking entailments between SL assertions. Typically, the focus is on decidable fragments such as separation logic of linked lists [4] for which entailment can be checked efficiently [10, 31]. Although there exist expressive decidable SL fragments that support complex data structures such as trees [18], these fragments have very high complexity. Therefore, reasoning about tree data structures is mostly unsupported in actual implementations, or handled by incomplete heuristics [29, 35]. This raises the question whether a practical and complete entailment procedure for SL of trees can be realized.

Contributions. In this paper, we give a positive answer to this question. Our solution builds on our earlier work on reducing entailment checking in separation logic to satisfiability checking in classical first-order logic [32]. Our main technical contribution therefore lies in the identification of a fragment of first-order logic that (1) supports reasoning about mutable tree data structures; (2) is sufficiently expressive to serve as a target of our SL reduction; and (3) is decidable in NP. We call this logic GRIT (for

⋆ Supported in part by NSF grant CCS-1320583.
⋆⋆ Supported by DARPA (Grant FA8650-11-C-7192)

Graph Reachability and Inverted Trees). The decision procedure for GRIT exploits locality of an axiomatic encoding of the logic's underlying theory and reduces satisfiability of GRIT formulas to satisfiability in effectively propositional logic (EPR). The latter is automated using an SMT solver. One advantage of this approach is that it allows for combinations with other decidable first-order theories. We therefore study several decidable extensions of our basic logic that utilize such combinations to support reasoning about data values stored in trees (e.g., sortedness constraints).

We have implemented our decision procedure on top of the SMT solver Z3 [11] and integrated it into our SL-based verification tool GRASShopper [33]. We successfully used the tool to automatically verify memory safety and consistency properties of tree-based data structures such as skew heaps and binary search trees. We have further used the tool to verify functional correctness of a tree-based set data structure and a union-find data structure. Proving such strong functional correctness properties often requires user-provided hints in the form of intermediate lemmas. However, GRASShopper can verify them completely automatically.

Related Work. The decision procedure for the target logic of our SL reduction draws ideas from the efficient SMT-based techniques for reasoning about reachability in function graphs [19, 23, 36, 38]. These techniques can be generalized to logics of trees [39] by viewing trees as inverted lists [3]. We make three important improvements over [39]. First, our logic does not make the global assumption that the entire heap forms a forest. This is important because such global assumptions contradict the philosophy of separation logic, where assertions express properties of heap regions rather than the entire heap. In particular, such assumptions preclude the encoding of the frame rule, which is crucial for enabling compositional program verification using separation logic. Second, we greatly simplify the decision procedure presented in [39]. This simplification turns a decision procedure that is mostly of theoretical interest into a procedure that is efficiently implementable. Finally, we consider extensions for reasoning about data.

Most other known decidable logics for reasoning about trees rely on monadic second-order logic (MSOL) [22, 37]. However, the high complexity of MSOL over trees limits its usefulness in verification. There exist some other expressive logics that support reachability in trees with lower complexity [8,13,17,41]. All these logics are still at least in EXPTIME, and their decision procedures are given in terms of automata-theoretic techniques, tableaux procedures, or small model properties. These can be difficult to combine efficiently with SMT solvers. One exception is the STRAND logic [26], which combines MSOL over trees with a data logic. There exists an implementation of a decision procedure for a decidable fragment of STRAND, which integrates MONA and an SMT solver. While the complexity of this procedure is at least double exponential, it has shown to be practically efficient [27]. However, similar to the logic in [39], STRAND makes global assumptions about the structure of the heap and is therefore inappropriate for an encoding of separation logic. Another orthogonal logic for reasoning about heap structures and data is described in [7]. This logic is incomparable to GRIT because it supports nested list structures but not trees, while GRIT supports trees but no nested structures.

Other tools that have been used for proving functional correctness of linked data structure implementations include Bedrock [9], Dafny [24], Jahob [42], HIP/SLEEK [29],

```
struct Node { var d: int; var l, r: Node; ghost var p: Node; }

procedure extract_max(rt, ghost pr: Node, implicit ghost C: set[int]) returns (nrt, max: Node)
  requires bst(rt, pr, C) ∗ rt ≠ null;
  ensures bst(nrt, pr, C \ {max.d}) ∗ acc(max);
  ensures max.r = null ∧ max.p = null ∧ max.d ∈ C ∧ (∀z ∈ (C \ {max.d})). z < max.d);
{
  var c, m: Node;
  if (rt.r != null) {
    c, m := extract_max(rt.r, rt);
    rt.r := c;
    return rt, m;
  } else {
    c := rt.l; rt.p := null;
    if (c != null) c.p := pr;
    return c, rt;
} }
```

Fig. 1. Extracting the node with the maximal value from a sorted binary search tree

and VeriFast [21]. While these tools can handle more programs and properties than GRASShopper supports, they also require more user guidance, either in the form of annotated ghost state or lemmas for discharging intermediate proof obligations.

Static shape analysis tools such as Forester [2] can automatically infer data structure invariants, e.g., that a specific reference points to a sorted tree. However, they only infer restricted properties about data stored in the heap and can usually not verify full functional correctness of data structure implementations.

2 Motivating Example and Overview

We motivate our approach through an example of a procedure that extracts the node storing the maximal value from a sorted binary search tree. The procedure and its specification are shown in Figure 1.

Specification. The extract_max procedure takes as argument the root of a binary search tree. The tree represents a set of integer values C, which is declared as an additional ghost parameter of the procedure. The precondition of the procedure, denoted by the requires clause, is an SL assertion that relates the two parameters using the inductive predicate bst(rt,pr,C). This predicate describes a heap region that forms a sorted binary search tree with root rt, parent node pr, and that stores the set of values C. We call the heap nodes in the region that are described by an SL assertion the *footprint* of the assertion. Note that the contract of extract_max provides the implicit guarantee that the procedure does not modify any allocated heap nodes that are outside of the footprint of its precondition.

The predicate bst is defined as follows:

$$\begin{aligned}\mathsf{bst}(x, y, C) \equiv\ & x = \mathsf{null} \land C = \varnothing \lor (\exists D E.\ \mathsf{acc}(x) * \mathsf{bst}(x.\mathsf{l}, x, D) * \mathsf{bst}(x.\mathsf{r}, x, E) * \\ & x.\mathsf{p} = y * C = \{x.\mathsf{d}\} \cup D \cup E * \forall u \in D.\ u < x.\mathsf{d} * \forall u \in E.\ u > x.\mathsf{d})\end{aligned}$$

The atomic predicate acc(x) in the definition of bst represents a heap region that consists of the single heap node x. That is, acc(x) means that x is in the footprint of the predicate. Such SL assertions are combined to assertions describing larger heap regions using *spatial conjunction*, denoted by '$*$'. Spatial conjunction asserts that the composed heap regions are disjoint in memory. Hence, bst describes an actual tree and not a DAG. Note that atomic assertions such as $\mathsf{x} = \mathsf{null}$ only express constraints on values but describe empty heap regions.

The procedure extract_max returns a pair of nodes (nrt, max) where nrt is the new root of the remaining tree, and max the node that has been removed. The postcondition, denoted by the ensures clauses, states that the procedure indeed yields the modified tree with the maximal node max properly removed.

One important detail in the contract of extract_max is the keyword implicit in the declaration of the ghost variable C. This annotation means that C is existentially quantified across the procedure contract. That is, we do not need to explicitly provide the actual value of C at call sites to extract_max, such as the recursive call on line 10. Instead, the verifier will automatically infer the existence of the actual value and use it when assuming the postcondition. This is in contrast to most other automated verification systems, which do not support implicit ghost parameters. However, to make our approach for reasoning about trees work, we do require the program to be annotated with ghost parent pointers. These must be updated along with the forward pointers that span the trees. We argue in the companion report [34] that in many cases these annotations with ghost parent pointers can be inferred automatically using simple heuristics.

Verification. The actual verification of extract_max involves a sequence of transformations that progressively make the semantics of separation logic explicit until we obtain a program in which all contracts are expressed in GRIT. The logic is closed under verification condition (VC) generation, and the generated VCs are then discharged using the decision procedure that we present in Sec. 5. The transformation includes the translation of SL assertions into first-order logic, the encoding of the semantics of SL Hoare triples by making the footprints of procedure contracts explicit, the insertion of checks for memory safety and absence of memory leaks, etc. The details of these transformations are described in our previous work [32, 33]. In the following, we only provide an abridged summary.

GRIT. The GRIT logic can express properties of sets of heap nodes using set operations and certain forms of set comprehensions. The logic further provides predicates that describe the structure of the heap. For example, the GRIT predicate $\mathsf{Tree}(S, x, y, l, r, p)$ expresses that the heap region described by the set S forms a tree with root x, parent node y, left pointer field l, right pointer field r, and parent pointer field p. Another important predicate is the *reachability predicate* $\mathsf{R}(f, x, y)$, which expresses that x can reach y by following the pointer field f in the heap. The logic also provides special constructs for expressing updates of pointer fields and frame conditions of procedure calls. Specifically, the *frame predicate* $\mathsf{Frame}(S, F, f, f')$ expresses that the values of the pointer fields f and f' agree on the heap nodes in the set $S \backslash F$.

Reduction to GRIT. We next explain how we reduce the problem of checking verification conditions with SL assertions to checking satisfiability of GRIT formulas. To this end, consider the path of extract_max that goes through the "then" branch of the condi-

$S = S_1 \cup S_2 \wedge S_1 = \{x.\, \mathsf{R}(\mathsf{p}, x, \mathsf{rt})\} \wedge S_2 = \varnothing \wedge$	footprint of precondition
$\mathsf{Tree}(S_1, \mathsf{rt}, \mathsf{pr}, \mathsf{l}, \mathsf{r}, \mathsf{p}) \wedge \mathsf{rt} \neq \mathsf{null} \wedge S_1 \cap S_2 = \varnothing \wedge$	precondition
$\mathsf{rt.r} \neq \mathsf{null} \wedge$	line 9
$F = F_1 \cup F_2 \wedge F_1 = \{x.\, \mathsf{R}(\mathsf{p}, x, \mathsf{rt.r})\} \wedge F_2 = \varnothing \wedge$	initial footprint of rec. call
$F = F'_1 \cup F'_2 \wedge F'_1 = \{x.\, \mathsf{R}(\mathsf{p}_1, x, \mathsf{c})\} \wedge F'_2 = \{\mathsf{m}\} \wedge$	final footprint of rec. call
$\mathsf{Tree}(F_1, \mathsf{c}, \mathsf{rt}, \mathsf{l}_1, \mathsf{r}_1, \mathsf{p}_1) \wedge \mathsf{m.r}_1 = \mathsf{m.p}_1 = \mathsf{null} \wedge F'_1 \cap F'_2 = \varnothing \wedge$	postcondition of rec. call
$\mathsf{Frame}(S, F, \mathsf{l}, \mathsf{l}_1) \wedge \mathsf{Frame}(S, F, \mathsf{r}, \mathsf{r}_1) \wedge \mathsf{Frame}(S, F, \mathsf{p}, \mathsf{p}_1) \wedge$	frame condition of rec. call
$\mathsf{r}_2 = \mathsf{write}(\mathsf{r}_1, \mathsf{rt}, \mathsf{c}) \wedge$	line 11
$\mathsf{nrt} = \mathsf{rt} \wedge \mathsf{max} = \mathsf{m} \wedge$	line 12
$S' = S'_1 \cup S'_2 \wedge S'_1 = \{x.\, \mathsf{R}(\mathsf{p}_2, x, \mathsf{nrt})\} \wedge S'_2 = \{\mathsf{max}\} \wedge$	footprint of postcondition
$\neg(\mathsf{Tree}(S'_1, \mathsf{nrt}, \mathsf{pr}, \mathsf{l}_1, \mathsf{r}_2, \mathsf{p}_2) \wedge \mathsf{max.p}_2 = \mathsf{null} \wedge \mathsf{max.r}_2 = \mathsf{null} \wedge$	negated postcondition
$S'_1 \cap S'_2 = \varnothing \wedge S' = S)$	

Fig. 2. Verification condition for a path of extract_max with simplified pre and postconditions

tional on line 9 to the return point on line 12. Our goal is to check that the postcondition of extract_max holds after this path has been executed, assuming the precondition holds initially. For exposition purposes, we consider the simplified precondition tree(rt,pr) $*$ rt $\neq$ null and the simplified postcondition

$$\mathsf{tree(nrt,pr)} * \mathsf{acc(max)} * (\mathsf{max.r} = \mathsf{null} \wedge \mathsf{max.p} = \mathsf{null})$$

That is, we abstract from the data values by defining the predicate tree as follows:

$$\mathsf{tree}(x, y) \equiv x = \mathsf{null} \vee \mathsf{acc}(x) * \mathsf{tree}(x.\mathsf{l}, x) * \mathsf{tree}(x.\mathsf{r}, x) * x.\mathsf{p} = y$$

The VC that is obtained from the simplified contract of extract_max and the considered path reduces to the GRIT formula shown in Fig. 2. This formula is unsatisfiable and thus the obtained VC is valid. We next explain this GRIT formula in more detail.

Translation of SL Assertions. The reduction to GRIT translates each SL assertion into a conjunction of two GRIT formulas: one formula that describes the footprint of the SL assertion, and another formula that describes the structure of the heap region captured by the assertion. The generation of the footprint formula proceeds recursively on the structure of the SL assertion, introducing auxiliary set variables to represent the footprints of all spatial conjuncts. These auxiliary set variables are implicitly existentially quantified, capturing the semantics of spatial conjunction. For example, in Fig. 2, the footprint of the precondition is described by the set S, which is itself the disjoint union of the sets S_1 and S_2. Here, S_1 represents the actual footprint of the tree rooted in rt. The variable S_1 is defined as the set of all nodes that can reach rt via the parent field p. S_2 is the footprint of the SL assertion rt $\neq$ null. Note that the defining formula for the footprint S' of the negated postcondition is pulled over the negation. Yet, we do not introduce universal quantifiers for the set variables S'_1 and S'_2 in the negated postcondition. The dualization of the universal quantifiers for the auxiliary set variables is possible because these variables are uniquely defined by the translated SL assertions. We refer the reader to the companion tech report [34] for the details of how to translate SL assertions with tree predicates to GRIT.

Implicit Frame Inference. The recursive call to extract_max on line 10 is handled by assuming the translated postcondition of the call and the defining formula of the footprint F of the call's precondition. The latter is used to express the call's frame condition using the predicate Frame. Note that the actual frame $S \backslash F$, i.e., the set of heap nodes that are not touched by the recursive call, is automatically inferred by the decision procedure of GRIT from the defining formulas of the footprint sets S and F.

3 Graph Reachability and Stratified Sets

Our reduction of separation logic to first-order logic decomposes SL assertions into constraints on the shape of the heap and constraints on the footprint sets. The crux in this translation is the handling of inductive predicates such as bst and tree. To avoid the need for reasoning about induction, both the shape constraints and the footprint sets are expressed in terms of reachability over pointer fields in the heap. To support such an encoding, we define a first-order logic of *graph reachability and stratified sets* (GRASS). This logic can express structural properties of mutable finite graphs as well as sets of nodes in these graphs. The general GRASS logic is undecidable. The logic GRIT, which we formally introduce in the next section, then imposes syntactic restrictions on GRASS that will ensure decidability while being sufficiently expressive to serve as a target for our reduction of separation logic over trees.

We follow standard notation and conventions for syntax and semantics of many-sorted first-order logic with equality. The signature of the GRASS logic is $\Sigma_{GS} = (S_{GS}, \Omega_{GS}, \Pi_{GS})$ where $S_{GS} = \{\mathsf{node}, \mathsf{field}, \mathsf{set}\}$ is the set of sorts. The set of function symbols Ω_{GS} consists of the symbols $\mathsf{null} : \mathsf{node}$, $\mathsf{read} : \mathsf{field} \times \mathsf{node} \to \mathsf{node}$, $\mathsf{write} : \mathsf{field} \times \mathsf{node} \times \mathsf{node} \to \mathsf{field}$, and a countable infinite set of constant symbols for each sort in S_{GS}. The constant symbol null is a dedicated constant symbol of sort node that we use to represent null pointers. The set of predicate symbols Π_{GS} consists of the symbols $\mathsf{B} : \mathsf{field} \times \mathsf{node} \times \mathsf{node} \times \mathsf{node}$ and $\in : \mathsf{node} \times \mathsf{set}$. The GRASS logic then comprises all first-order formulas over the signature Σ_{GS}.

We define the semantics of GRASS formulas with respect to a theory $\mathcal{T}_{GS}$ of first-order structures over Σ_{GS}. A structure $\mathcal{A}$ is in $\mathcal{T}_{GS}$ iff the following conditions are satisfied. First, $\mathcal{A}$ interprets the sort node by a finite set $\mathsf{node}^{\mathcal{A}}$. The interpretation of the remaining sorts and symbols, with the exception of constant symbols, is then uniquely determined by the interpretation of $\mathsf{node}^{\mathcal{A}}$ as follows. First, the sort field is interpreted by the set of all functions in $\mathsf{node}^{\mathcal{A}} \to \mathsf{node}^{\mathcal{A}}$, and the sort set by the set of all subsets of $\mathsf{node}^{\mathcal{A}}$. We consider the elements of $\mathsf{node}^{\mathcal{A}}$ to represent nodes in a heap graph and the elements of $\mathsf{field}^{\mathcal{A}}$ pointer fields. The function symbols read and write represent field look-up and field update. They must satisfy the following properties

$$\forall u \in \mathsf{node}^{\mathcal{A}}, f \in \mathsf{field}^{\mathcal{A}}.\ \mathsf{read}^{\mathcal{A}}(f, u) = (\,\mathrm{if}\ u = \mathsf{null}^{\mathcal{A}}\ \mathrm{then}\ u\ \mathrm{else}\ f(u)) \quad \text{and}$$

$$\forall u, v \in \mathsf{node}^{\mathcal{A}}, f \in \mathsf{field}^{\mathcal{A}}.\ \mathsf{write}^{\mathcal{A}}(f, u, v) = \lambda w \in \mathsf{node}^{\mathcal{A}}.\ \mathrm{if}\ w = u\ \mathrm{then}\ v\ \mathrm{else}\ f_{\mathcal{A}}(w)$$

The *between predicate* $\mathsf{B}(f, x, y, z)$ denotes that x reaches z via an f-path that must go though y. To formally define the semantics of B, we note that for a binary relation r over a set X (respectively, a unary function $r : X \to X$), we denote by r^* the reflexive transitive closure of r. Furthermore, for $f \in \mathsf{field}^{\mathcal{A}}$ we define $f_{\mathcal{A}} = \lambda u \in \mathsf{node}^{\mathcal{A}}.\ \mathsf{read}^{\mathcal{A}}(f, u)$. Then for all $u, v, w \in \mathsf{node}^{\mathcal{A}}$ and $f \in \mathsf{field}^{\mathcal{A}}$ we require

$$\mathsf{B}^{\mathcal{A}}(f, u, v, w) \Leftrightarrow (u, w) \in f_{\mathcal{A}}^* \land\ (u, v) \in \{(u_1, f_{\mathcal{A}}(u_1)) \mid u_1 \in \mathsf{node}^{\mathcal{A}} \land u_1 \neq w\}^*$$

$$
\begin{array}{ll}
x : \text{node variable}, \quad t : \text{node constant}, \quad S : \text{set constant}, & \mathsf{Fld} \in \{P, L, R\}, \quad f \in \mathsf{Fld} \\
T ::= x \mid t \mid \mathsf{null} \mid \mathsf{read}(f, T) & T_{\mathsf{Fld}} ::= f \mid \mathsf{write}(T_{\mathsf{Fld}}, T, T) \\
A ::= T = T \mid T_{\mathsf{Fld}} = T_{\mathsf{Fld}} \mid \mathsf{B}(T_P, T, T, T) \mid T \in S & R ::= A \mid \neg R \mid R \wedge R \\
F_{\forall} ::= \forall \boldsymbol{x}.R \quad \text{where the variables } \boldsymbol{x} \text{ do not occur below read or write in } R & \\
F ::= A \mid F_{\forall} \mid \mathsf{Tree}(S, T, T_L, T_R, T_P) \mid \mathsf{Frame}(S, S, T_{\mathsf{Fld}}, T_{\mathsf{Fld}}) \mid \neg F \mid F \wedge F &
\end{array}
$$

Fig. 3. Logic of graph reachability and inverted trees (GRIT)

The second conjunct states that u reaches v without going through w. Finally, the interpretation of the set membership relation $\in$ in $\mathcal{A}$ is as expected. We define the reachability predicate $\mathsf{R}(f, x, y)$ as a short-hand for $\mathsf{B}(f, x, y, y)$.

4 The GRIT Logic

We now introduce the logic of graph reachability and inverted trees (GRIT). In our formal treatment, we restrict ourselves to the case of binary trees. However, the logic and decision procedure can be easily generalized to trees of arbitrary bounded rank. We do not discuss the case of unranked trees. Surprisingly, the treatment of unranked trees is much simpler than the bounded case.

Syntax. Figure 3 defines the syntax of GRIT formulas. A GRIT formula F is a Boolean combination of atomic formulas A, restricted quantified formulas $F_{\forall}$, tree predicates $\mathsf{Tree}(S, t, l, r, p)$, and frame predicates $\mathsf{Frame}(A, S, f, f')$. The atomic formulas A form a subset of the atomic formulas of GRASS. Namely, we partition the constant symbols of sort field into three disjoint sets: a set of parent fields P, a set of left successor fields L, and a set of right successor fields R. Equalities between terms of sort field are then restricted to terms that are built from field constants in the same partition. To ensure decidability of the logic, we do not allow quantification over formulas that contain terms in which node variables appear below the function symbols read and write, as in $\mathsf{read}(p, x)$. Also, we restrict the reachability predicate B to parent fields. This restriction yields a much simpler and more practical decision procedure compared to the logic proposed in [39]. We assume that all GRIT formulas are closed.

Syntactic Short-hands. Throughout the remainder of the paper, we will use syntactic short-hands for disjunction, implication, bi-implication, and existential quantification in GRIT formulas. Further note that we can express standard set operations such as union and intersection using restricted quantified GRIT formulas and fresh auxiliary set constants. For example, the formula $t \notin S \cup T$ stands for the GRIT formula

$$\neg(t \in S_1) \wedge \forall x.\, x \in S_1 \Leftrightarrow x \in S \vee x \in T$$

where S_1 is a fresh set constant. Set equality, subset inclusion, and set comprehensions can be expressed similarly. To ease the notation, we will use the expected syntactic short-hands for such encodings. Finally, we write $t.f$ to mean $\mathsf{read}(f, t)$.

Semantics. GRIT formulas are interpreted in the models of the theory $\mathcal{T}_{GS}$, which we have defined in the previous section. Thus, we only need to provide the semantics of the predicates Tree and Frame. We define the semantics of these predicates in terms of formulas in our general graph reachability logic with stratified sets. The defining formulas are chosen in such a way that we obtain a simple and efficient decision procedure by first expanding the predicates with their defining formulas and then applying quantifier instantiation techniques. In the following, let $\mathcal{A}$ be a structure in $\mathcal{T}_{GS}$.

As we have seen in Sec. 2, the tree predicate is crucial for the translation of SL tree predicates such as tree. A tree predicate $\mathsf{Tree}(S, t, l, r, p)$ holds true in $\mathcal{A}$ if $\mathcal{A}$ contains a tree with footprint S, root t, spanned by the given fields l, r, and p. Our formal definition of Tree, which we provide below, uses the reachability predicates to give a noninductive definition of trees. Formally, $\mathcal{A}$ satisfies $\mathsf{Tree}(S, t, l, r, p)$ iff the following formula holds in $\mathcal{A}$:

$$t = \mathsf{null} \wedge S = \emptyset \;\vee \tag{1}$$
$$\forall x, y.\, x \in S \wedge y \in S \wedge \mathsf{R}(p, x, y) \wedge \mathsf{R}(p, y, x) \Rightarrow x = y \;\wedge \tag{2}$$
$$\forall x.\, x \in S \Rightarrow x.l = \mathsf{null} \vee \mathsf{R}(p, x.l, x) \;\wedge \tag{3}$$
$$\forall x, y.\, x \in S \wedge \mathsf{B}(p, x.l, y, x) \Rightarrow y = x.l \vee y = x \;\wedge \tag{4}$$
$$\forall x.\, x \in S \Rightarrow x.r = \mathsf{null} \vee \mathsf{R}(p, x.r, x) \;\wedge \tag{5}$$
$$\forall x, y.\, x \in S \wedge \mathsf{B}(p, x.r, y, x) \Rightarrow y = x.r \vee y = x \;\wedge \tag{6}$$
$$\forall x, y.\, y \in S \wedge \mathsf{R}(p, x, y) \Rightarrow x = y \vee \mathsf{B}(p, x, y.l, y) \vee \mathsf{B}(p, x, y.r, y) \;\wedge \tag{7}$$
$$\forall x.\, x \in S \wedge x.l = x.r \Rightarrow x.l = \mathsf{null} \;\wedge \tag{8}$$
$$\forall x.\, x \in S \Rightarrow x.l \neq x \wedge x.r \neq x \;\wedge \tag{9}$$
$$\forall x.\, x \in S \Leftrightarrow \mathsf{R}(p, x, t) \tag{10}$$

The first disjunct (1) defines the structure of an empty tree and the second disjunct the structure of nonempty trees. We explain the conjuncts (2)-(10) in the second disjunct in more detail. The conjunct (2) ensures that the set S does not contain nontrivial p cycles. The conjuncts (3)-(7) express that on S the field p is the inverse of l and r. Specifically, (3) and (4) together are equivalent to the formula $\forall x. x \in S \Rightarrow x.l = \mathsf{null} \vee x.l.p = x$. Using reachability constraints to express this property rather than field reads yields a simpler and more efficient decision procedure. Conjunct (8) expresses that fields l and r must not point to the same nodes on S, unless they both point to null. Conjunct (9) expresses that l and r do not have self-loops on S. Finally, conjunct (10) defines the footprint S as the set of all nodes that can reach t via the parent field.

The frame predicate $\mathsf{Frame}(A, S, f, f')$ expresses that the fields f and f' coincide when they are restricted to the nodes in the set $A \backslash S$. In our formal definition of the frame predicate, we distinguish between parent fields and successor fields. For successor fields $f, f' \in L$ (respectively R), we define $\mathsf{Frame}(A, S, f, f')$ by the GRASS formula

$$\forall x. x \in A \backslash S \Rightarrow x.f = x.f' \tag{11}$$

For parent fields $p, p' \in P$, it is not sufficient if the frame predicate states that the fields p and p' coincide on the set $A \backslash S$. Instead, we also need to ensure that all information contained in the reachability predicate B for the two fields is consistent on this set. For parent fields p, p', we therefore define $\mathsf{Frame}(A, S, p, p')$ by the formula

$$\forall x, y, z. x \in A \backslash S \Rightarrow (\mathsf{B}(p, x, y, z) \Leftrightarrow \mathsf{B}(p', x, y, z)) \tag{12}$$

Note that the formula (12) is stronger than formula (11). In fact, we are only allowed to use formula (12) for the encoding of the frame rule if we make sure that the set S is parent-closed. That is, for all nodes t, t', if $t \in S$ and $\mathsf{R}(p, t', t)$, then $t' \in S$. This holds in particular if S is the footprint of an SL tree predicate whose parent field is p. There exists a more general treatment of the frame predicate that preserves reachability information and does not make assumptions about the set S. The details can be found in [20, 33]. Since the footprints of tree manipulating programs are typically defined by tree predicates and hence parent-closed, we stick to the simpler definition given by formula (12).

5 Decision Procedure for GRIT

We next describe the decision procedure for the satisfiability problem of GRIT. In the following, let F be a GRIT formula. The decision procedure works in two phases: the first phase reduces F to an equisatisfiable GRASS formula F'. The second phase reduces F' to an equisatisfiable formula in effectively propositional logic (EPR), which is then checked using an EPR decision procedure. The EPR fragment, also known as the Bernays-Schönfinkel class, consists of formula of the form $\exists \boldsymbol{x} \forall \boldsymbol{y} \varphi(\boldsymbol{x}, \boldsymbol{y})$ where φ is quantifier-free and does not contain function symbols. Satisfiability of EPR formulas can be decided in NEXPTIME and reduces to NP, if the number of universally quantified variables is bounded [25].

The first phase of the reduction involves the following sequence of steps:

1. Substitute all occurrences of the predicates Tree and Frame in F by their defining formulas given in Sec. 4. The resulting formula is a GRASS formula F_1.
2. Convert F_1 into negation normal form, yielding F_2.
3. Replace every literal of the form $\neg(f = f')$ in F_2, where f and f' are terms of sort field, by the formula $\exists x.\, \neg(\mathsf{read}(f, x) = \mathsf{read}(f', x))$. The resulting formula is F_3.
4. Skolemize F_3, yielding F'.

Clearly, each of these transformation steps produces an equisatisfiable formula with respect to the theory $\mathcal{T}_{GS}$. Note that the Skolemization step only introduces fresh Skolem constants of sort node.

Next, conjoin F' with the theory axioms defining the predicate B and the functions read and write for the theory $\mathcal{T}_{GS}$. The axiom defining the predicate B are obtained from the inference rules in the decision procedure proposed in [23]. The axioms defining the functions read and write are McCarthy's well-known read over write axioms for arrays [28][1]. We denote the resulting formula by G.

All the remaining quantifiers in G are universal quantifiers. The final step of the reduction is to instantiate all those universally quantified variables in G that appear below function symbols. The resulting formula is then in EPR (modulo function symbols appearing in ground terms, which can be eliminated using Ackermann reduction). The quantifier instantiation step exploits the careful design of the defining formulas of the tree and frame predicates, as well as the restrictions on the quantified formulas that are

[1] The complete list of all axioms can be found in the companion tech report [34].

allowed to appear in the input formula F. These restrictions guarantee that the resulting quantified constraints can be viewed as a so-called Ψ-local theory extension [16]. That is, it is sufficient to instantiate the variables below function symbols in G with a finite set of ground terms T that we can compute from G. Suppose that $G[T]$ is the resulting EPR formula. The completeness argument for the reduction to EPR works by proving that each model $\mathcal{A}$ of $G[T]$ can be embedded into some structure in $\mathcal{T}_{GS}$ that satisfies F'. Specifically, we need to be able to construct actual binary trees in those regions of $\mathcal{A}$ that have been constrained by tree predicates. This construction must preserve the cardinality of model $\mathcal{A}$ for the resulting structure to satisfy F' and hence F.

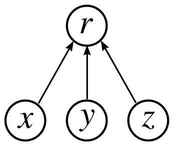
To this end, let T_G be the set of all ground terms appearing in G, and let P_F be the set of all positive ground literals appearing in F. To ensure that each tree region $\mathsf{Tree}(S, t, l, r, p) \in P_F$ contains sufficiently many nodes to construct a binary tree, we use the idea from [39] to introduce an auxiliary function fca that denotes the first common ancestor of two nodes with respect to the parent field p and footprint set S. We define this function using the following axioms:

$$\begin{aligned}
&\forall x, y.\ \mathsf{R}(p, x, t) \wedge \mathsf{R}(p, y, t) \Rightarrow \mathsf{R}(p, x, \mathit{fca}(p, x, y))\\
&\forall x, y.\ \mathsf{R}(p, x, t) \wedge \mathsf{R}(p, y, t) \Rightarrow \mathsf{R}(p, y, \mathit{fca}(p, x, y))\\
&\forall x, y, z.\ \mathsf{R}(p, x, t) \wedge \mathsf{R}(p, y, t) \wedge \mathsf{R}(p, x, z) \wedge \mathsf{R}(p, y, z) \Rightarrow \mathsf{R}(p, \mathit{fca}(p, x, y), z)\\
&\forall x, y, z, w.\ \mathsf{R}(p, w, t) \wedge \mathit{fca}(p, x, y) = w \wedge \mathit{fca}(p, x, z) = w \wedge \mathit{fca}(p, y, z) = w \Rightarrow\\
&\qquad x = y \vee x = z \vee y = z \vee w = \mathsf{null}
\end{aligned}$$

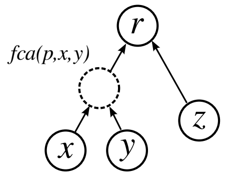
For each atom $\mathsf{Tree}(S, t, l, r, p) \in P_F$, conjoin these axioms with G to obtain G_1.

Next, we define the set of ground terms T, which we use for the instantiation, as the least set of ground terms that satisfies the following properties:

- $T_G \subseteq T$
- if $t.l \in T$ and $\mathsf{Tree}(S, c, l, r, p) \in P_F$ then $t.r \in T$
- if $t.r \in T$ and $\mathsf{Tree}(S, c, l, r, p) \in P_F$ then $t.l \in T$
- if $t.f \in T$ and $f = f' \in P_F$ then $t.f' \in T$
- if $t.f \in T$ and $\mathsf{Frame}(A, S, f, f') \in P_F$ then $t.f' \in T$
- if $t.f \in T$ and $\mathsf{Frame}(A, S, f', f) \in P_F$ then $t.f' \in T$
- if $t.\mathsf{write}(f, u, v) \in T$ then $t.f \in T$
- if $t.f \in T$ and $\mathsf{write}(f, u, v) \in T$ then $t.\mathsf{write}(f, u, v) \in T$
- if $\mathsf{write}(f, u, v) \in T$ then $u.\mathsf{write}(f, u, v) \in T$
- if $t \in T$, $t' \in T$, $\mathsf{Tree}(S, c, l, r, p) \in P_F$, and neither t nor t' contain fca then $\mathit{fca}(p, t, t') \in T$



(a) spurious model



(b) model with $\mathit{fca}(p, x, y)$

Fig. 4. Role of the fca

It is easy to see that T is polynomial in the size of T_G.

Let A be a universally quantified first-order formula. We denote by $A[T]$ the conjunction of all instances I of A that satisfy the following properties: I is obtained from A by instantiating all quantified variables of A that appear below function symbols with terms of matching sort in T. Moreover, all ground terms appearing in I are already in T. For example, if A is of the form $\forall x.\, A_1(f(x))$ and $t \in T$ then $A_1(f(t))$ is in $A[T]$ only if $f(t) \in T$. Now, let $G_1[T]$ be the formula that is obtained by substituting all universally quantified subformulas A in G_1 by $A[T]$.

By construction, the formula $G_1[T]$ is satisfiable if the input formula F is satisfiable modulo the theory $\mathcal{T}_{GS}$. Hence, our decision procedure is sound. To prove completeness, let $\mathcal{A}$ be a model of $G_1[T]$. Define the partial structure $\mathcal{A}|_T$ by restricting the interpretation of the sort node in $\mathcal{A}$ to the set $\{\, t^{\mathcal{A}} \mid t \in T \,\}$. Let $\mathsf{PMod}(G_1[T])$ be the set of all such partial structures for $G_1[T]$. Then the following lemma implies the completeness of our decision procedure.

Lemma 1. *Let $\mathcal{A}_T \in \mathsf{PMod}(G_1[T])$. Then $\mathcal{A}_T$ can be completed to a structure $\mathcal{A} \in \mathcal{T}_{GS}$ that satisfies G.*

The crucial observation in the proof of Lemma 1 is that the addition of fca ensures that in the partial models $\mathcal{A}_T$, for every tree node t, there are at most two other nodes which can reach t directly via the parent field, i.e., without visiting any other nodes. Hence, these two nodes can be chosen as the direct left and right successors of t in the model completion. We explain the importance of the first common ancestor terms for the completeness of the decision procedure through an example.

Example 1. Consider the following unsatisfiable formula:

$$\mathsf{Tree}(S, t, l, r, p) \wedge S = \{x, y, z, t\} \wedge S = \{w.\, \mathsf{R}(p, w, t)\} \wedge \neg\mathsf{R}(p, x, y) \wedge \\ \neg\mathsf{R}(p, x, z) \wedge \neg\mathsf{R}(p, y, x) \wedge \neg\mathsf{R}(p, y, z) \wedge \neg\mathsf{R}(p, z, x) \wedge \neg\mathsf{R}(p, z, y)$$

The formula is unsatisfiable because the nodes x, y, z and t cannot be arranged in a binary tree without adding auxiliary nodes to the tree (which violates the definition of S) or making at least two of x, y, z mutually reachable via p. Without the first common ancestor, the reduced formula produced by the decision procedure would admit the model shown in Fig. 4 (a). This happens because the original formula does not contain any l or r terms to trigger the instantiation of the quantifiers in the defining formula of Tree. However, with the additional fca terms and axioms, the instantiated formula implies that the tree must contain at least one additional node, as indicated in Fig. 4 (b). This yields the contradiction.

By construction, $G_1[T]$ is an EPR formula whose size is polynomial in the input formula F. It thus follows that the satisfiability problem for the quantifier-bounded fragments of GRIT is in NP. Since NP-hardness is immediate we obtain the following complexity result.

Theorem 1. *The satisfiability problem for the quantifier-bounded fragments of GRIT is NP complete.*

6 Extensions

In this section, we discuss several extensions of GRIT to support reasoning about trees and data. Such extensions are needed, for instance, to prove that a binary search tree is sorted. In general, it is possible to extend the logic with additional axioms about data as long as they preserve the locality properties that underpin the axiomatization of GRIT. We present extensions with data that we used in our experimental evaluation and we provide some general principles how to design such extensions. The extensions that we discuss preserve the decidability and complexity of GRIT.

To support reasoning about data, we extend the signature of GRIT with an additional sort data for data values, fields from node to data, and sets with data elements. The read and write functions are extended as expected. In the following, we let d range over data fields. In our implementation, we interpret the data sort in the theory of linear integer arithmetic. However, we can combine GRIT with any decidable quantifier-free first-order theory that is signature disjoint from GRIT and stably-infinite to interpret the data sort. The extensions that we discuss build on such quantifier-free combinations.

We consider three categories of extensions with data: monadic predicates on node, binary predicates on node, and projections of node sets to data sets.

Monadic predicates. Properties such as upper and lower bounds on the values contained in a tree are expressible using monadic predicates. Such formulas have the following form: $\forall x.\ x \in S \rightarrow Q(x.d)$ where Q is a predicate over data and S a node set. One example of such a predicate is the ensures clause on line 6 in Fig. 1.

Monadic predicates also form Ψ-local theory extensions. To support such extensions, the set T of ground terms for the quantifier instantiation must additionally satisfy:

- if $d \in T$ and $t \in T$ then $t.d \in T$

For each ground node term $t \in T$, we add a ground term that reads t's data. The completeness of this instantiation follows from results about axioms satisfying stratified sort restrictions [1].

Binary predicates. To define a sorted tree or a heap, we need to relate the data of a node to the data of its children. The following properties are examples of binary predicates:

- heap property: $\forall x, y \in S.\ \mathsf{R}(p, x, y) \Rightarrow x.d \leqslant y.d$
- sorted tree (left subtree): $\forall x, y \in S.\ \mathsf{B}(p, x, y.l, y) \Rightarrow x.d < y.d$

Here, we assume that S is the footprint of a tree. To ensure completeness of the decision procedure for such predicates, we first check that the relation on nodes is transitive. Without transitivity, the property cannot be generalized from direct successors in a tree to an entire path in that tree, and Lemma 1 does not hold anymore. Transitivity prevents us from expressing properties that require counting, but still allows ordering relations.

The case of the heap property is simple since it satisfies a stratified sort restriction. It does not require any additional treatment beyond the addition of data ground terms as in the case of monadic predicates. On the other hand, the sortedness property is more interesting because the variable y appears in a read term. Thus, instantiating the axiom can potentially generate new terms. However, our decision procedure performs only local instantiation. To obtain completeness we need to ensure that we have sufficiently many left and right successor terms. Therefore, the set T of ground terms must additionally satisfy:

- if $fca(p, t_1, t_2) \in T$ and $\mathsf{Tree}(S, c, l, r, p) \in P_F$ then $fca(p, t_1, t_2).l \in T$ and $fca(p, t_1, t_2).r \in T$.

Note that the axioms for the first common ancestor and the defining formula of Tree together imply that the following holds for all nodes x and y in a partial model:

$$fca(p, fca(p, x, y).l, fca(p, x, y).r) = fca(p, x, y)$$

The additional terms ensure that all nodes in a tree are assigned to the left, respectively, right subtrees of the first common ancestor nodes, enforcing sortedness across the tree.

Set projection. Lastly, we consider a way of referring to the content of a data structure. This class of extensions enables reasoning about functional correctness properties. In common cases such as implementations of sets, the content is obtained by projecting the footprint onto a data field. For instance, given the footprint S of a data structure, the content C can be defined as $C = \{v \mid \exists x \in S.\ v = x.d\}$.

This definition does not directly fit into our logic, due to the existential quantifier inside the set comprehension. We replace this quantifier by a Skolem function which we call $witness$. The $witness$ function maps an element c of C back to a node in S that stores c. The values not in C are mapped to null. $witness$ is axiomatized as follows:

$$\begin{array}{l} \forall x. x \in S \Rightarrow x.d \in C \\ \forall v. v \in C \Rightarrow witness(d, v, C) \in S \wedge v = witness(d, v, C).d \\ \forall v. v \notin C \Rightarrow witness(d, v, C) = \mathsf{null} \end{array}$$

The witness function maps the data values back to nodes. Therefore, it does not respect the stratification restriction used to prove the Ψ-locality of the monadic extensions. For completeness, the set of terms T needs to additionally satisfy:

– if $d \in T_{\mathsf{data}}$ and $v, C \in T$ then $witness(d, v, C), witness(d, v, C).d \in T$

The axioms are local since $witness$ is the inverse of d. Hence, reading the data of a witness gives a value which is already in the set of ground terms.

The set implementations which we used in our experiments do not store duplicate elements and $witness$ becomes the one-to-one inverse of the data field. In such cases, we strengthen the above axioms with $\forall x.\ x \in S \Rightarrow x = witness(d, x.d, C)$.

Limitations. As mention earlier, there is no precise characterization of the limit of extensions that preserve the locality properties on which our decision procedure is built. However, not all extensions are local. For example, the following relation between a parent and child node does not generalize to reachability: $\forall x, y \in S.\ x.p = y \Rightarrow x.d = y.d + 1$. Therefore, the height of a tree cannot be expressed.

7 Implementation and Evaluation

Implementation. We have extended our tool GRASShopper with the decision procedure for tree data structures storing integer values. The tool is implemented in OCaml and available under a BSD license. The source code distribution including all benchmarks can be downloaded from the project web page [14]. GRASShopper takes as input an annotated C-like program and generates verification conditions which are checked using Z3 [11]. Annotations include procedure contracts and loop invariants expressed in a mixed specification language that supports both SL and GRASS assertions. The tool automatically adds checks to ensure that there are no memory safety violations such as accesses to unallocated memory, memory leaks, double frees, etc.

Currently, all annotations with ghost parent pointers must be manually provided. We plan to extend GRASShopper to automatically infer these annotations. For example, each modification of a forward successor field induces a matching modification of the parent field. Furthermore, a procedure that takes the root of a tree as parameter must be augmented with an additional ghost parameter for the parent of the root. The companion report [34] contains more information on how to automate these steps.

To handle dynamic memory allocation we require that the parent of all unallocated nodes points to null and that all nodes eventually reach null via parents. These restrictions are not harmful because outside of trees we can choose the parents arbitrarily.

The translation of SL tree predicates into GRIT is currently hard-coded into the implementation of the tool. The first-order specifications of common properties and features such as sortedness and content sets are provided as predefined building blocks. Using these building blocks, adding support for a new data structure requires about 10 lines of code. We plan to implement a heuristic translation of SL tree predicates to GRIT. The tool already provides such a heuristic for list data structures. GRASShopper also incorporates optimizations and sparser term generation. For instance, we do not currently generate the fca terms. This source of incompleteness proved irrelevant in our examples. In every example, the data structure is traversed along the left and right successor nodes which ensures that sufficiently many ground terms are already present.

Evaluation. We have used GRASShopper to verify complex properties of various data structure implementations. The results of our experiments are summarized in Table 1. For each procedure, the table lists the number of lines of code, lines of specification, lines of ghost annotations, the number of generated verification conditions, and the total running time of the tool. All examples in the table have been successfully verified. The number of lines of code does not include specifications or ghost state. The specifications include contracts and loop invariants. The ghost annotations include annotations that are needed to express the specification (e.g., implicit ghost parameter), or proof automation (e.g., updates of ghost fields). We now describe our experiments in more detail.

First, we used GRASShopper to prove functional correctness of set data structures that store integer values. We considered implementations based on binary search trees and sorted lists. The experiments with lists show that the extension we present for GRIT are applicable across different data structure types. We further verified a union-find data structure. We looked at the data structure from two different perspectives. One perspective views them as shared lists, the other as unranked inverted forests. Each perspective allows us to prove different properties of the implementation. Using the tree view, we proved functional correctness, e.g., that the union operation indeed merges the equivalence classes associated with two given pointers into the data structure. The list view allows us to reason about single paths from a node n in the data structure to the root node of the tree that n belongs to (i.e., the representative of that equivalence class). Using the list view, we proved the correctness of path compaction in the find operation.

We have also considered other tree data structures for which we have proved the preservation of structural invariants under the data structure operation but not full func-

Data structure	Procedure	#L. Code	#L. Spec	#L. Ghost	# VCs	time in s
set as binary tree functional correctness	contains	17	3	3	9	3
	destroy	8	2	2	7	1
	extract_max	14	5	3	9	20
	insert	24	2	3	15	61
	remove	33	2	11	35	117
	rotate_left	8	3	4	11	15
	rotate_right	8	3	4	11	14
	traverse	7	2	3	5	9
set as sorted list functional correctness	contains	15	7	6	4	1
	delete	26	7	6	8	12
	difference	20	3	1	15	13
	insert	25	7	6	8	69
	traverse	12	7	6	2	0.1
	union	20	3	1	15	15
union-find (tree-view) functional correctness	find	12	2	1	4	0.2
	union	10	3	1	4	0.3
	create	11	3	0	3	0.1
union-find (list-view) path compaction	find	12	3	1	4	0.1
	union	9	7	1	4	3
	create	10	1	0	3	0.1
skew heap shape, heap property	insert	17	2	2	7	0.3
	union	11	2	4	12	35
	extract_max	9	2	1	11	6

Table 1. Verified data structures

tional correctness of these operations. In particular, we have proved that skew heap operations respect the heap property, i.e., that the data value of a child node is not greater than its parent's value. Skew heaps are typically used to implement priority queues. At the moment, we cannot prove functional correctness of this data structure because our tool does not yet support reasoning about the theories that are needed for specifying the priority queue operations (e.g., multisets or sequences).

8 Conclusions

We have presented a new approach for automated verification of programs that manipulate heap-allocated data structures. The approach is based on a decidable fragment of first-order logic that supports reasoning about mutable finite graphs and can express that certain subgraphs form trees. The logic makes no global assumptions about the structure of its graph models such as that the entire graph is a forest. This allows us to use the logic for automated reasoning about separation logic of trees. Furthermore, we have studied extensions of our graph logic for reasoning about data stored in heap structures. We used these extensions to automatically verify complex properties (including full functional correctness) of tree data structures such as binary search trees, skew heaps, and union-find. In the future, we will investigate how to extend our techniques to reason about nested data structures that combine trees, lists, and arrays.

References

1. A. Abadi, A. Rabinovich, and M. Sagiv. Decidable fragments of many-sorted logic. In *LPAR*, pages 17–31, Berlin, Heidelberg, 2007. Springer-Verlag.
2. P. A. Abdulla, L. Holík, B. Jonsson, O. Lengál, C. Q. Trinh, and T. Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. In *ATVA*, pages 224–239. Springer, 2013.
3. I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis of single-parent heaps. In *VMCAI*, Lect. Notes in Comp. Sci. Springer, 2007.
4. J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, 2004.
5. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.
6. J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory Safety for Systems-Level Code. In *CAV*, 2011.
7. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. A logic-based framework for reasoning about composite data structures. In *CONCUR*, 2009.
8. D. Calvanese, G. di Giacomo, D. Nardi, and M. Lenzerini. Reasoning in expressive description logics. In *Handbook of Automated Reasoning*. Elsevier, 2001.
9. A. Chlipala. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In *ICFP*, pages 391–402. ACM, 2013.
10. B. Cook, C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*. Springer, 2011.
11. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
12. K. Dudka, P. Peringer, and T. Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *CAV*, 2011.
13. P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *ACM PLDI*, 2007.
14. GRASShopper tool web page. `http://cs.nyu.edu/wies/software/grasshopper`. Accessed: May 2014.
15. C. Haase, S. Ishtiaq, J. Ouaknine, and M. J. Parkinson. Seloger: A tool for graph-based reasoning in separation logic. In *CAV*, pages 790–795, 2013.
16. C. Ihlemann, S. Jacobs, and V. Sofronie-Stokkermans. On local reasoning in verification. In *TACAS*, pages 265–281, 2008.
17. N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Computer Science Logic (CSL)*, pages 160–174, 2004.
18. R. Iosif, A. Rogalewicz, and J. Simácek. The tree width of separation logic with recursive definitions. In *CADE*, volume 7898 of *LNCS*, pages 21–38. Springer, 2013.
19. S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In *CAV*. Springer, 2013.
20. S. Itzhaky, O. Lahav, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Modular reasoning on unique heap paths via effectively propositional formulas. In *POPL*, 2014.
21. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, 2011.
22. N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.
23. S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *POPL*, pages 171–182, 2008.

24. K. R. M. Leino. Developing verified programs with dafny. In *ICSE*, pages 1488–1490. ACM, 2013.
25. H. R. Lewis. Complexity results for classes of quantificational formulas. *J. Comput. Syst. Sci.*, 21(3):317–353, 1980.
26. P. Madhusudan, G. Parlato, and X. Qiu. Decidable logics combining heap structures and data. In *POPL*, pages 611–622. ACM, 2011.
27. P. Madhusudan and X. Qiu. Efficient Decision Procedures for Heaps Using STRAND. In *SAS*, volume 6887 of *LNCS*, pages 43–59. Springer, 2011.
28. J. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.
29. H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, pages 251–266. Springer, 2007.
30. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proc. CSL, Paris 2001*, volume 2142 of *LNCS*, 2001.
31. J. A. N. Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, pages 556–566. ACM, 2011.
32. R. Piskac, T. Wies, and D. Zufferey. Automating Separation Logic Using SMT. In *CAV*, volume 8044 of *LNCS*, pages 773–789. Springer, 2013.
33. R. Piskac, T. Wies, and D. Zufferey. GRASShopper: Complete Heap Verification with Mixed Specifications. In *TACAS*. Springer, 2014.
34. R. Piskac, T. Wies, and D. Zufferey. On automating separation logic with trees and data. Technical Report NYU Technical Report TR2014-963, NYU, 2014.
35. X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242, 2013.
36. Z. Rakamaric, J. D. Bingham, and A. J. Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In *VMCAI*, volume 4349 of *LNCS*, pages 106–121. Springer, 2007.
37. J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.
38. N. Totla and T. Wies. Complete instantiation-based interpolation. In *POPL*. ACM, 2013.
39. T. Wies, M. Muñiz, and V. Kuncak. An efficient decision procedure for imperative tree data structures. In *CADE*, volume 6803 of *LNCS*, pages 476–491. Springer, 2011.
40. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.
41. G. Yorsh, A. M. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. *J. Log. Algebr. Program.*, 2007.
42. K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361. ACM, 2008.