Automating Separation Logic Using SMT

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

[1] MPI-SWS, Germany
[2] New York University, USA
[3] IST Austria

Abstract. Separation logic (SL) has gained widespread popularity because of its ability to succinctly express complex invariants of a program's heap configurations. Several specialized provers have been developed for decidable SL fragments. However, these provers cannot be easily extended or combined with solvers for other theories that are important in program verification, e.g., linear arithmetic. In this paper, we present a reduction of decidable SL fragments to a decidable first-order theory that fits well into the satisfiability modulo theories (SMT) framework. We show how to use this reduction to automate satisfiability, entailment, frame inference, and abduction problems for separation logic using SMT solvers. Our approach provides a simple method of integrating separation logic into existing verification tools that provide SMT backends, and an elegant way of combining SL fragments with other decidable first-order theories. We implemented this approach in a verification tool and applied it to heap-manipulating programs whose verification involves reasoning in theory combinations.

1 Introduction

Separation logic (SL) [25] is an extension of Hoare logic [20] for proving the correctness of heap-manipulating programs. Its great asset lies in its assertion language, which can succinctly express how data structures are laid out in memory. This language has two characteristic features: it provides 1) a *spatial conjunction* operator that decomposes the heap into disjoint regions, each of which can be reasoned about independently (this enables an elegant treatment of pointer aliasing); and 2) *inductive spatial predicates* that describe the shape of unbounded linked data structures such as lists, trees, etc. SL assertions give rise to the so-called *frame rule*, a Hoare-logic proof rule that enables compositional verification of heap-manipulating programs.

The frame rule makes separation logic attractive for developers of program verification tools [6, 8, 19, 21, 36]. However, the logic also poses a challenge to automation: it is a non-classical logic that requires specialized symbolic execution engines for encoding the behavior of programs, and specialized theorem provers for discharging the generated proof obligations. Existing SL-based tools therefore implement their own tailor-made theorem provers. This brings its own challenges.

First, extending existing verification tools that rely on specifications written in classical first-order logic with SL support is a significant effort. The tailor-made SL provers cannot be easily integrated with the theorem provers used by such tools. Second, the analysis of real-world programs involves more than just reasoning about heap structures. For instance, the combination of linked data structures and pointer arithmetic is pervasive in low-level system code [14, 19]. Other examples include the dynamic reinterpretation of memory (e.g., treating a memory region both as a linked structure and as an array of bit-vectors) and dependencies on data stored in linked structures (e.g., sortedness constraints). To deal with such programs, existing SL tools make simplifying and (deliberately) unsound assumptions about the underlying memory model, rely on interactive help from the user, or implement incomplete extensions to allow some limited support for reasoning about other theories.

The integration of a separation logic prover into an SMT solver can address these challenges. Modern SMT solvers such as CVC4 [3] and Z3 [16] already implement decision procedures for many theories that are relevant in program verification, e.g., linear arithmetic, arrays, and bit-vectors. They also implement generic mechanisms for combining these theories, treating the theory solvers as independent components. These mechanisms provide guarantees about completeness and decidability. A reduction of separation logic to first-order logic enables such complete combinations with other theories. Finally, SMT solvers are already an integral part in the tool chain of many existing verification tools. These tools could directly benefit from an integrated SL prover.

So far, a seamless integration of a separation logic prover and an SMT solver has not yet been realized. This paper represents a step towards achieving this goal.

We propose a technique for SMT-based reasoning about separation logic assertions. Our technique relies on a translation of SL formulas into a decidable fragment of first-order logic, which we refer to as the *logic of graph reachability and stratified sets* (GRASS). Formulas in this logic express properties of the structure of graphs, such as whether nodes in the graph are inter-reachable, as well as properties of sets of nodes. These sets are used to give a natural encoding of the semantics of spatial conjunction, while graph reachability enables reasoning about inductive spatial predicates without relying on induction, which is not well supported by first-order theorem provers.

We show how to use the translation to check satisfiability and entailment of SL formulas. The latter enables automated verification of programs with SL specifications. In particular, it can leverage existing infrastructure for verification condition generation provided by tools such as Boogie [2]. Using a characterization of partial models of GRASS formulas, we further demonstrate how our technique can solve frame inference and abduction problems, which are key to efficient inter-procedural analysis of heap-manipulating programs [11]. Finally, we prove that our translation enables theory combination of separation logic within the Nelson-Oppen combination framework [24].

To demonstrate the feasibility of our approach we have implemented the decision procedure for GRASS using an SMT solver. Based on this implementation we built a prototype tool for inter-procedural analysis of heap-manipulating programs. We successfully used this tool to automatically verify procedures manipulating list-like data structures against specifications expressed in separation logic. Our examples include benchmarks such as sorting algorithms whose verification relies on the combination of heap and arithmetic reasoning.

Related Work. Several decidable fragments of separation logic have been studied in the literature. Most prominent is the fragment of linked lists introduced in [5], which is now used in extended forms by many SL-based tools. The original paper describes a decision procedure for satisfiability and entailment for the fragment of linked lists.

More recently, these problems were shown to be decidable in polynomial time [15] using a graph-based algorithm.

Translations of separation logic into first-order logic have been previously studied in [12] and [9]. The result in [12] does not consider inductive predicates, which are needed for expressing properties of recursive data structures such as lists, trees, etc. The approach in [9] considers inductive predicates but does not target a decidable fragment. Neither of these approaches considers frame inference, abduction, or theory combination. In [27], Perez and Rybalchenko showed that significant performance improvements can be obtained by incorporating first-order theorem proving techniques into SL provers. More recently and concurrently to us, they have also considered the problem of theory combination [28]. The authors of [10] describe another hybrid approach in which an SL decision procedure for entailment is extended to enable reasoning about quantified constraints on data. However, [10] does not address theory combination in general. Also, neither [27] nor [10] consider frame inference or abduction.

Alternatives to separation logic that enable compositional reasoning about heap-manipulating programs but rely on classical logic include (implicit) dynamic frames [22, 31] and region logic [1, 30]. The connection between separation logic and implicit dynamic frames has been studied in [26]. Our reduction of separation logic to first-order logic is in part inspired by region logic, which also uses sets to partition the memory into disjoint regions. A crucial difference from our approach is that region logic has no inbuilt support for recursive data structures.

We build on previous results on SMT-based decision procedures for theories of reachability in graphs [23, 33, 34] and decision procedures for theories of stratified sets [37]. Our logic GRASS combines these two theories and extends them with set comprehensions that define sets of nodes in terms of properties expressed in the logic. This extension is essential to enable a succinct encoding of spatial conjunctions.

2 Preliminaries

We present our approach in many-sorted first-order logic with equality, which is the theoretical foundation of modern SMT solvers. We follow standard notation and conventions for syntax and semantics of first-order logic as defined, e.g., in [29].

Many-sorted first-order logic. A *signature* Σ is a tuple (S, Ω, Π), where S is a countable set of *sorts*, Ω is a countable set of *function symbols*, and Π is a countable set of *predicate symbols*. Each function and predicate symbol has an associated sort, which is a tuple of sorts in S. A function symbol whose sort is a single sort in S is called *constant*. For two signatures Σ_1 and Σ_2 we write $\Sigma_1 \cup \Sigma_2$ for the signature that is obtained by taking the point-wise union of Σ_1 and Σ_2. We say that Σ_1 and Σ_2 are *disjoint* if they do not share any function or predicate symbols. We write $\Sigma_1 \subseteq \Sigma_2$ if $\Sigma_1 \cup \Sigma_2 = \Sigma_2$. A Σ-term is built as usual from the function symbols in Ω and variables taken from a set $\mathcal{X}$ that is disjoint from S, Ω, and Π. Each variable $x \in \mathcal{X}$ has an associated sort in S. We also assume the standard notions of Σ-atom, Σ-literal, and Σ-formula.

Interpretations and structures. In the following, let $\Sigma = (S, \Omega, \Pi)$ be a signature. A *partial Σ-interpretation* $\mathcal{A}$ over variables $\mathcal{X}$ is a function that maps each sort $s \in S$ to a non-empty set $s^{\mathcal{A}}$ and each function symbol $f \in \Omega$ of sort $s_1 \times \cdots \times s_n \to t$ to a partial function $f^{\mathcal{A}} : s_1^{\mathcal{A}} \times \cdots \times s_n^{\mathcal{A}} \rightharpoonup t^{\mathcal{A}}$. Similarly, every predicate $p \in \Pi$ of sort $s_1 \times \cdots \times s_n$ is interpreted as a relation $p^{\mathcal{A}} \subseteq s_1^{\mathcal{A}} \times \cdots \times s_n^{\mathcal{A}}$. Finally, $\mathcal{A}$ interprets every variable $x \in \mathcal{X}$ of associated sort $s \in S$ by some element $x^{\mathcal{A}} \in s^{\mathcal{A}}$. A partial interpretation $\mathcal{A}$ is called *total interpretation* or simply *interpretation* if it interprets all function symbols by total functions. We denote by $\mathcal{A}|_{\Sigma',\mathcal{X}'}$ the Σ'-interpretation over $\mathcal{X}'$ that is obtained by restricting $\mathcal{A}$ to a signature $\Sigma' \subseteq \Sigma$ and a set of variables $\mathcal{X}' \subseteq \mathcal{X}'$. We further write $\mathcal{A}|_{\Sigma}$ for $\mathcal{A}|_{\Sigma,\emptyset}$. A *(partial) Σ-structure* is a (partial) Σ-interpretation over an empty set of variables. For a partial Σ-interpretation $\mathcal{A}$ and $\sigma \in S \cup \Omega \cup \Pi \cup \mathcal{X}$, we denote by $\mathcal{A}[\sigma \mapsto v]$ the interpretation that is like $\mathcal{A}$ but interprets σ as v.

Given a Σ-interpretation $\mathcal{A}$, the evaluation $t^{\mathcal{A}}$ of a Σ-term t in $\mathcal{A}$ is defined inductively over the structure of t, as usual. Similarly, the evaluation of a Σ-formula in $\mathcal{A}$ is obtained from the interpretation of terms in the usual way. In particular, we use the standard interpretations for equality, propositional connectives, and quantifiers. A quantified variable of sort s ranges over all elements of $s^{\mathcal{A}}$. For a formula F we denote by $F^{\mathcal{A}} \in \{0,1\}$ its truth value in $\mathcal{A}$. A formula F is *satisfied* in $\mathcal{A}$, written $\mathcal{A} \models F$, if $F^{\mathcal{A}} = 1$. In this case, we also call $\mathcal{A}$ a *model* of F. We say that F is *satisfiable*, if F has a model. For two Σ-formulas F and G, we say F *entails* G, written $F \models G$, if $\mathcal{A} \models F$ implies $\mathcal{A} \models G$ for all Σ-interpretations $\mathcal{A}$.

Theories and theory combinations. A *Σ-theory* is a class of Σ-structures. Given a Σ-theory $\mathcal{T}$, a *$\mathcal{T}$-interpretation* is a Σ-interpretation $\mathcal{A}$ such that $\mathcal{A}|_{\Sigma} \in \mathcal{T}$. A Σ-formula is called *$\mathcal{T}$-satisfiable*, if it is satisfiable in some $\mathcal{T}$-interpretation. The *quantifier-free satisfiability problem of $\mathcal{T}$* is to decide for every quantifier-free Σ-formula whether it is $\mathcal{T}$-satisfiable or not.

Let Σ be a signature with sorts S and $S' \subseteq S$. A Σ-theory is called *stably infinite* with respect to S', if each $\mathcal{T}$-satisfiable quantifier-free Σ-formula is satisfiable in a $\mathcal{T}$-interpretation $\mathcal{A}$ such that $s^{\mathcal{A}}$ has infinite cardinality, for all $s \in S'$.

Let $\mathcal{T}_i$ be a Σ_i-theory, for $i = 1, 2$, and let $\Sigma = \Sigma_1 \cup \Sigma_2$. The *combination* of $\mathcal{T}_1$ and $\mathcal{T}_2$ is the Σ-theory $\mathcal{T}_1 \oplus \mathcal{T}_2 = \{\, \mathcal{A} \mid \mathcal{A}|_{\Sigma_1} \in \mathcal{T}_1 \text{ and } \mathcal{A}|_{\Sigma_2} \in \mathcal{A}_2 \,\}$. We call $\mathcal{T}_1 \oplus \mathcal{T}_2$ the *disjoint combination* of $\mathcal{T}_1$ and $\mathcal{T}_2$ if Σ_1 and Σ_2 are disjoint.

3 Logic of Graph Reachability and Stratified Sets

In this section we formally introduce the logic of graph reachability and stratified sets (GRASS), which is the target for our reduction of separation logic. Formulas in the logic are interpreted in (function) graphs and can express properties of the graph structure as well as properties of sets of nodes in the graph that are defined in terms of properties of the graph structure.

Syntax of GRASS. Throughout the rest of this paper we assume that $\mathcal{X}$ is a countably infinite set of variables of sorts node and set. We use the lower-case symbols $x, y \in \mathcal{X}$ for variables of sort node and upper-case symbols $X, Y \in \mathcal{X}$ for variables of sort set.

The syntax of GRASS is defined in Figure 1. A GRASS formula is a propositional combination of atoms. There are two types of atoms. Atoms of type A are either equalities between terms of type T and *reachability predicates* of the form $t_1 \xrightarrow{h\backslash t_3} t_2$. The terms of type T represent nodes in the graph. They have associated sort node and

$$\begin{array}{lll} T ::= x \mid h(T) & A ::= T = T \mid T \xrightarrow{h\backslash T} T & R ::= A \mid \neg R \mid R \wedge R \mid R \vee R \\ S ::= X \mid \varnothing \mid S\backslash S \mid S \cap S \mid S \cup S \mid \{x.\, R\} & \multicolumn{2}{l}{x \text{ does not occur below } h \text{ in } R} \\ B ::= S = S \mid T \in S & F ::= A \mid B \mid \neg F \mid F \wedge F \mid F \vee F & \end{array}$$

Fig. 1. Logic of graph reachability and stratified sets (GRASS)

are constructed from variables and application of the function symbol h, which represents the (functional) edge relation of the graph. Intuitively, a reachability predicate $t_1 \xrightarrow{h\backslash t_3} t_2$ is true if there exists a path in the graph that connects t_1 and t_2 without going through t_3. Atoms of type B are equalities between terms of type S, which have an associated sort set, and membership tests. Terms of type S represent *stratified sets* [37], i.e., their elements are interpreted – here, as nodes in the graph. S-terms include *set comprehensions* of the form $\{x.\, R\}$, where R is a Boolean combination of atoms of type A. We require that the term $h(x)$ does not occur in R. This side condition is important to ensure the decidability of the logic.

We use syntactic short-hands for implication, bi-implication, etc. We further write $t_1 \xrightarrow{h} t_2$ as an abbreviation for $t_1 \xrightarrow{h\backslash t_2} t_2$ and we use short-hands for the universal set $\mathcal{U}$, which stands for $\{x.\, x = x\}$, subset inclusion $S_1 \subseteq S_2$, which stands for $S_1 \cup S_2 = S_2$, and set enumerations $\{t_1, \dots, t_n\}$, which stand for $\{x.\, x = t_1 \vee \dots \vee x = t_n\}$ where x does not occur in $t_1, \dots, t_n$. Finally, we write $X = Y \uplus Z$ for $X = Y \cup Z \wedge Y \cap Z = \varnothing$.

Example 1. Consider the formula $F \equiv Y = \{x.\, x \xrightarrow{h} y\} \wedge Z = \{x.\, x \xrightarrow{h} z\} \wedge \mathcal{U} = Y \uplus Z$. This formula describes all function graphs that consist of two disjoint connected components, one in which all nodes reach y, and one in which all nodes reach z.

Semantics of GRASS. We define the semantics of GRASS with respect to a specific theory $\mathcal{T}_{\mathsf{GS}}$. The theory $\mathcal{T}_{\mathsf{GS}}$ is the disjoint combination of a theory of reachability in function graphs $\mathcal{T}_{\mathsf{G}}$ and a theory of stratified sets $\mathcal{T}_{\mathsf{S}}$.

We first define the theory $\mathcal{T}_{\mathsf{G}}$. The structures in $\mathcal{T}_{\mathsf{G}}$ are over the signature $\Sigma_{\mathsf{G}} = (S_{\mathsf{G}}, \Omega_{\mathsf{G}}, \Pi_{\mathsf{G}})$ with sorts $S_{\mathsf{G}} = \{\mathsf{node}\}$, function symbols $\Omega_{\mathsf{G}} = \{h\}$, and predicate symbols $\Pi_H = \{\xrightarrow{h\backslash}\}$. The sort of h is $\mathsf{node} \to \mathsf{node}$ and the sort of $\xrightarrow{h\backslash}$ is $\mathsf{node} \times \mathsf{node} \times \mathsf{node}$. The structures in $\mathcal{T}_{\mathsf{G}}$ are defined as follows. For a binary relation r over a set S (respectively, a unary function from S to S), we denote by r^* the reflexive transitive closure of r. A structure $\mathcal{A}$ over signature Σ_{G} is in $\mathcal{T}_{\mathsf{G}}$ iff the following conditions are satisfied. First, the interpretation of the edge function h in $\mathcal{A}$ is constrained as follows: for all $u \in \mathsf{node}^{\mathcal{A}}$, the sets $\{\, v \in \mathsf{node}^{\mathcal{A}} \mid (u, v) \in (h^{\mathcal{A}})^* \,\}$ and $\{\, v \in \mathsf{node}^{\mathcal{A}} \mid (v, u) \in (h^{\mathcal{A}})^* \,\}$ are finite. Second, the interpretation of the reachability predicate is defined in terms of $h^{\mathcal{A}}$ as follows. For all $u, v, w \in \mathsf{node}^{\mathcal{A}}$

$$u \xrightarrow{h\backslash w}{}^{\mathcal{A}}\, v \iff (u, v) \in \{\, (u_1, h^{\mathcal{A}}(u_1)) \mid u_1 \in \mathsf{node}^{\mathcal{A}} \wedge u_1 \neq w \,\}^*$$

Note that $u \to^{\mathcal{A}} v$ iff $(u, v) \in (h^{\mathcal{A}})^*$.

Next, we define the theory of stratified sets $\mathcal{T}_{\mathsf{S}}$ [37]. The structures in $\mathcal{T}_{\mathsf{S}}$ are over the signature $\Sigma_{\mathsf{S}} = (S_{\mathsf{S}}, \Omega_{\mathsf{S}}, \Pi_{\mathsf{S}})$ with sorts $S_{\mathsf{S}} = \{\mathsf{node}, \mathsf{set}\}$, function symbols $\Omega_{\mathsf{S}} = \{\varnothing, \cap, \cup, \backslash\}$, and predicate symbols $\Pi_{\mathsf{S}} = \{\in\}$. The symbol $\varnothing$ is a constant of sort

$$x, y \in \mathcal{X} \qquad \Sigma ::= x = y \mid x \neq y \mid x \mapsto y \mid \mathsf{ls}(x, y) \mid \Sigma * \Sigma \qquad H ::= \Sigma \mid \neg H \mid H \wedge H$$

Fig. 2. $\mathsf{SLL}\mathbb{B}$: separation logic of linked lists

set and the function symbols $\cap$, $\cup$, and $\setminus$ all have sort $\mathsf{set} \times \mathsf{set} \to \mathsf{set}$. The predicate symbol $\in$ has sort $\mathsf{node} \times \mathsf{set}$. A structure $\mathcal{A}$ is in $\mathcal{T}_\mathsf{S}$ iff $\mathcal{A}$ interprets the sort set as the set of all subsets of $\mathsf{node}^\mathcal{A}$ and the symbols in Ω_S and Π_S are interpreted as expected.

Now define $\Sigma_\mathsf{GS} = \Sigma_\mathsf{G} \cup \Sigma_\mathsf{S}$ and $\mathcal{T}_\mathsf{GS} = \mathcal{T}_\mathsf{G} \oplus \mathcal{T}_\mathsf{S}$. We call the structures in $\mathcal{T}_\mathsf{GS}$ *heap structures*, referring to their use later on in the paper. Likewise, we call a Σ_GS-interpretation whose reduct is a heap structure a *heap interpretation*. We denote by $\mathcal{T}_{\mathsf{GS},\mathcal{X}}$ the set of all heap interpretations.

We next define the semantics of GRASS formulas. Let $\mathcal{A}$ be a heap interpretation. The evaluation of terms of type T and formulas of type R in $\mathcal{A}$ are defined as in first-order logic. Using these definitions we define the evaluation of a set comprehensions $\{x.\, R\}$ in $\mathcal{A}$ as follows: $\{x.\, R\}^\mathcal{A} = \{\, u \in \mathsf{node}^\mathcal{A} \mid R^{\mathcal{A}[x \mapsto u]} = 1 \,\}$.

The definition of the evaluation function is then extended by structural induction to terms of type S and formulas of type F, as expected. The notions of satisfiability and entailment are defined as for first-order logic, except that we restrict ourselves to heap interpretations, respectively, heap structures. We denote by $\mathcal{A} \models_\mathsf{GS} F$ that GRASS formula F is satisfied by heap interpretation $\mathcal{A}$.

The satisfiability problem of GRASS asks whether a given GRASS formula F is satisfiable. This problem is decidable. The decision procedure can be implemented within an SMT solver using a Nelson-Oppen combination of solvers for $\mathcal{T}_\mathsf{G}$ and $\mathcal{T}_\mathsf{S}$. We describe this procedure in Appendix A. The following theorem only states its existence.

Theorem 2. *The satisfiability problem of GRASS is NP-complete.*

4 Separation Logic of Linked Lists

We consider separation logic formulas that are given by propositional combinations of formulas in separation logic of linked lists [5] (SLL). We refer to our fragment of separation logic simply as $\mathsf{SLL}\mathbb{B}$. The syntax of the formulas in this fragment are given in Figure 2. That is, a formula is a propositional combination of *spatial formulas* Σ. A spatial formula is an equality or disequality of variables (of sort node), a *points-to predicate* $x \mapsto y$, a *list segment predicate* $\mathsf{ls}(x, y)$, or a *spatial conjunction* $\Sigma_1 * \Sigma_2$ of spatial formulas. We denote by $\mathcal{H}$ the set of all these formulas. We use syntactic sugar for disjunctions. We further write emp for $x = x$, false for $x \neq x$, and true for $\neg(x \neq x)$, where $x \in \mathcal{X}$ is some fixed variable.

The standard semantics of separation logic formulas is given with respect to a variable assignment (referred to as stack) and a partial function on memory addresses to values (referred to as the heap). In order to be able to easily relate formulas in $\mathsf{SLL}\mathbb{B}$ and GRASS, we define the semantics of $\mathsf{SLL}\mathbb{B}$ formulas in terms of heap interpretations $\mathcal{A}$. Our semantics is consistent with the standard semantics (except for one minor deviation that we explain below). The interpretation of the edge function $h^\mathcal{A}$ plays the role of

$$
\begin{array}{ll}
\mathcal{A}, X \models_{\mathsf{SL}} x = y & \text{iff } x^{\mathcal{A}} = y^{\mathcal{A}} \text{ and } X^{\mathcal{A}} = \emptyset \\
\mathcal{A}, X \models_{\mathsf{SL}} x \neq y & \text{iff } x^{\mathcal{A}} \neq y^{\mathcal{A}} \text{ and } X^{\mathcal{A}} = \emptyset \\
\mathcal{A}, X \models_{\mathsf{SL}} x \mapsto y & \text{iff } h^{\mathcal{A}}(x^{\mathcal{A}}) = y^{\mathcal{A}} \text{ and } X^{\mathcal{A}} = \{x^{\mathcal{A}}\} \\
\mathcal{A}, X \models_{\mathsf{SL}} H_1 * H_2 & \text{iff } \exists U_1, U_2.\, U_1 \cup U_2 = X^{\mathcal{A}} \text{ and } U_1 \cap U_2 = \emptyset \text{ and} \\
& \quad \mathcal{A}[X \mapsto U_1], X \models_{\mathsf{SL}} H_1 \text{ and } \mathcal{A}[X \mapsto U_2], X \models_{\mathsf{SL}} H_2 \\
\mathcal{A}, X \models_{\mathsf{SL}} \mathsf{ls}(x, y) & \text{iff } \exists n \geqslant 0.\, \mathcal{A}, X \models_{\mathsf{SL}} \mathsf{ls}^n(x, y) \\
\mathcal{A}, X \models_{\mathsf{SL}} \mathsf{ls}^0(x, y) & \text{iff } x^{\mathcal{A}} = y^{\mathcal{A}} \text{ and } X^{\mathcal{A}} = \emptyset \\
\mathcal{A}, X \models_{\mathsf{SL}} \mathsf{ls}^{n+1}(x, y) & \text{iff } \exists u \in \mathsf{node}^{\mathcal{A}}.\, \mathcal{A}[z \mapsto u], X \models_{\mathsf{SL}} x \mapsto z * \mathsf{ls}^n(z, y) \\
& \quad \text{and } x^{\mathcal{A}} \neq y^{\mathcal{A}} \text{ and } z \neq x \text{ and } z \neq y \\
\mathcal{A}, X \models_{\mathsf{SL}} H_1 \wedge H_2 & \text{iff } \mathcal{A}, X \models_{\mathsf{SL}} H_1 \text{ and } \mathcal{A}, X \models_{\mathsf{SL}} H_2 \\
\mathcal{A}, X \models_{\mathsf{SL}} \neg H & \text{iff not } \mathcal{A}, X \models_{\mathsf{SL}} H
\end{array}
$$

Fig. 3. Semantics of $\mathsf{SLL}\mathbb{B}$ in terms of heap interpretations

the heap in the standard semantics and the variable assignment in $\mathcal{A}$ plays the role of the stack. Since a heap structure $\mathcal{A}$ interprets h by a total function and the standard interpretation of separation logic is with respect to a heap that is a partial function, we must explicitly say which subset of $\mathsf{node}^{\mathcal{A}}$ we use to interpret a separation logic formula. For this purpose, we use a set variable $X \in \mathcal{X}$ whose interpretation in $\mathcal{A}$ determines this subset. We call $X^{\mathcal{A}}$ the *footprint* of the interpreted formula. The set variable X is a parameter of the semantics. The satisfaction relation is denoted by judgments of the form $\mathcal{A}, X \models_{\mathsf{SL}} H$, as defined in Figure 3.

If $\mathcal{A}, X \models_{\mathsf{SL}} H$ holds, we say that H is satisfied by $\mathcal{A}$ with respect to X, respectively, that $\mathcal{A}$ is a model of H with respect to X. Entailment between two $\mathsf{SLL}\mathbb{B}$ formulas H_1 and H_2 (written $H_1 \models_{\mathsf{SL}} H_2$) is then defined as expected.

The satisfiability problem for $\mathsf{SLL}\mathbb{B}$ asks whether a given $\mathsf{SLL}\mathbb{B}$ formula H is satisfiable in some heap interpretation $\mathcal{A}$ with respect to some set variable X. It follows from results in [15], that this problem is NP-complete.

Unlike the standard semantics of separation logic, our semantics is *precise* [13]. That is, the footprint of a spatial formula is uniquely defined in each model. In particular, (dis)equalities constrain the heap to be empty. For example, the formula $x \neq y \wedge \mathsf{ls}(x, y)$ is unsatisfiable because $x \neq y$ implies both that the heap is empty and that $\mathsf{ls}(x, y)$ is a non-empty list segment. On the other hand, the formula $x \neq y * \mathsf{ls}(x, y)$ is satisfiable and describes all heaps containing non-empty list segments from x to y, which is the meaning of $x \neq y \wedge \mathsf{ls}(x, y)$ in the standard semantics. The deviation from the standard semantics is therefore of little practical consequence and is in fact adopted by some separation logic tools. Our approach also works for the standard semantics of (dis)equalities, but the correctness proofs are more involved. We can further adapt our approach to handle other imprecise formulas such as formulas with disjunctions and conjunctions below spatial conjunction. The only problematic generalization that can-

$$\begin{aligned}
str_Y(x = y) &= (x = y,\ Y = \emptyset) \qquad str_Y(x \mapsto y) = (h(x) = y,\ Y = \{x\}) \\
str_Y(x \neq y) &= (x \neq y,\ Y = \emptyset) \qquad str_Y(\mathsf{ls}(x, y)) = (x \xrightarrow{h} y,\ Y = Btwn(x, y)) \\
str_Y(\Sigma_1 * \Sigma_2) &= \text{let } Y_1, Y_2 \in \mathcal{X} \text{ fresh and } (F_1, G_1) = tr_{Y_1}(\Sigma_1) \text{ and } (F_2, G_2) = tr_{Y_2}(\Sigma_2) \\
&\quad\ \text{in } (F_1 \wedge F_2 \wedge Y_1 \cap Y_2 = \emptyset,\ Y = Y_1 \cup Y_2 \wedge G_1 \wedge G_2) \\
tr_X(\Sigma) &= \text{let } Y \in \mathcal{X} \text{ fresh and } (F, G) = str_Y(\Sigma) \text{ in } (F \wedge X = Y,\ G) \\
tr_X(\neg H) &= \text{let } (F, G) = tr_X(H) \text{ in } (\neg F,\ G) \\
tr_X(H_1 \wedge H_2) &= \text{let } (F_1, G_1) = tr_X(H_1) \text{ and } (F_2, G_2) = tr_X(H_2) \\
&\quad\ \text{in } (F_1 \wedge F_2,\ G_1 \wedge G_2) \\
Tr_X(H) &= \text{let } (F, G) = tr_X(H) \text{ in } F \wedge G
\end{aligned}$$

Fig. 4. Translation of SLL𝔹 to GRASS

not be easily handled is to admit negation below spatial conjunction. To our knowledge, no (automated) SL tool supports such formulas because of the increased complexity.

5 Reduction of SLL𝔹 to GRASS

In the following, we present our reduction approach for automated reasoning about SLL𝔹 formulas. We show that every SLL𝔹 formula can be reduced in linear time to an equisatisfiable GRASS formula. By using our decision procedure for GRASS, this reduction yields an SMT-based decision procedure for the satisfiability and entailment problem of SLL𝔹. Furthermore, it enables theory combination of SLL𝔹 with signature disjoint theories within the Nelson-Oppen combination framework.

Translating SLL𝔹 to GRASS. We start with the translation function Tr that maps SLL𝔹 formulas to GRASS formulas. It is shown in Figure 4. The function is parameterized by a set variable X, which holds the footprint of the translated formula. The translation is defined using two auxiliary functions str and tr.

The function str maps a set variable Y and a spatial formula Σ to a pair of GRASS formulas (F, G). The formula F captures the structure of Σ, while the formula G defines auxiliary set variables that are used to link Y to the footprint of Σ. The function str is defined recursively on the structure of Σ. Note that it closely follows the semantics of spatial formulas. In particular, to define the footprint Y of a spatial conjunction $\Sigma_1 * \Sigma_2$, the function str introduces two fresh set variables to capture the footprints of Σ_1 and Σ_2, respectively, and then defines Y as the disjoint union of these two sets. Also, note that we do not need induction to translate list segments. Instead, the structure and footprint of a list segment are translated directly using reachability predicates. Here, we write $Btwn(x, y)$ as a short-hand for the set comprehension $\{z.x \xrightarrow{h\backslash y} z \wedge z \neq y\}$.

The function tr translates Boolean combinations of spatial conjunctions. At the leaf level, tr introduces fresh set variables Y to translate the meaning of spatial formulas Σ and asserts $X = Y$ in the structural constraint. The constraints G defining the auxiliary set variables are propagated to the top level where the function Tr conjoins them with the structural constraint F of the entire formula. The following lemma implies that the translation yields an equisatisfiable formula.

Lemma 3. *Let H be an $\mathsf{SLL}\mathbb{B}$ formula, $X \in \mathcal{X}$, and $\mathcal{A}$ a heap interpretation. Then $\mathcal{A}$ satisfies H with respect to X iff there exist subsets $U_1, \ldots, U_n$ of $\mathsf{node}^{\mathcal{A}}$ such that $\mathcal{A}[Y_1 \mapsto U_1, \ldots, Y_n \mapsto U_n] \models_{\mathsf{GS}} Tr_X(H)$, where $Y_1, \ldots, Y_n$ are the fresh set variables introduced in the translation $Tr_X(H)$.*

Note that the auxiliary set variables Y_i that are introduced for the translation of spatial conjunctions are implicitly existentially quantified. Hence, when a spatial conjunction appears below an odd number of negations, these existential quantifiers should become universal quantifiers. One might therefore wonder why the propagation of constraints G to the top level of the formula is still correct, since all set variables remain existentially quantified. It is here where the precise semantics of spatial formulas helps. Each constraint G is a conjunction of equalities defining the sets Y_i as finite unions of set comprehensions. Therefore, these constraints are satisfiable in any given heap interpretation. In fact, for each constraint G and heap interpretation $\mathcal{A}$, there exists exactly one assignment of the Y_i to $U_i \subseteq \mathsf{node}^{\mathcal{A}}$ that makes G true in $\mathcal{A}$. Hence, the formulas $\exists Y_1, \ldots, Y_n. F \wedge G$ and $\forall Y_1, \ldots, Y_n. G \Rightarrow F$ are equivalent.

For two $\mathsf{SLL}\mathbb{B}$ formulas H_1 and H_2, we have that H_1 entails H_2 iff $H_1 \wedge \neg H_2$ is unsatisfiable. It follows from Lemma 3 that our translation yields a decision procedure for satisfiability and entailment of $\mathsf{SLL}\mathbb{B}$ formulas.

Theorem 4. *The satisfiability and entailment problems of $\mathsf{SLL}\mathbb{B}$ are reducible in linear time to the satisfiability problem of GRASS.*

Example 5. Consider the two separation logic formulas $H_1 \equiv x \neq z * x \mapsto y * \mathsf{ls}(y, z)$, and $H_2 \equiv \mathsf{ls}(x, z)$. Both formulas describe heaps consisting of an acyclic list segment from x to z. In the case of H_1, the segment is non-empty, while H_2 also allows the empty segment, i.e., $H_1 \models_{\mathsf{SL}} H_2$. Let $X \in \mathcal{X}$ be a set variable. Then $Tr_X(H_1)$ is

$$\begin{aligned}&x \neq z \wedge h(x) = y \wedge y \xrightarrow{h} z \wedge Y_2 \cap Y_3 = \emptyset \wedge Y_4 \cap Y_5 = \emptyset \wedge X = Y_1 \wedge \\ &Y_1 = Y_2 \cup Y_3 \wedge Y_2 = \emptyset \wedge Y_3 = Y_4 \cup Y_5 \wedge Y_4 = \{x\} \wedge Y_5 = Btwn(y, z)\end{aligned}$$

which can be simplified to $x \neq z \wedge h(x) \xrightarrow{h} z \wedge X = \{x\} \uplus Btwn(h(x), z)$. We further have $Tr_X(\neg H_2) \equiv \neg(x \xrightarrow{h} z \wedge X = Y_6) \wedge Y_6 = Btwn(x, z)$. To see why $Tr_X(H_1) \wedge Tr_X(\neg H_2)$ is unsatisfiable, note that $h(x) \xrightarrow{h} z$ implies $x \xrightarrow{h} z$ and $Btwn(x, z) = \{x\} \cup Btwn(h(x), z)$.

Combining $\mathsf{SLL}\mathbb{B}$ with other theories. We next show that the theory $\mathcal{T}_{\mathsf{GS}}$ behaves well with respect to theory combination. For instance, we can combine it with a theory of integer arithmetic for interpreting memory addresses. We can then use this theory to reason about SL fragments in which we allow address arithmetic. Similar combinations enable reasoning about fragments that can express properties about data. To implement these theory combinations, we can leverage the Nelson-Oppen combination framework [24] provided in SMT solvers.

Formally, let Σ_{node} be a signature that is disjoint from Σ_{GS} and that contains at least the sort node. Let further $\mathcal{T}_{\mathsf{node}}$ be a decidable Σ_{node}-theory that is stably infinite with respect to sort node. For example, $\mathcal{T}_{\mathsf{node}}$ may be the theory of linear arithmetic, interpreting the sort node as integers. Define $\Sigma = \Sigma_{\mathsf{node}} \cup \Sigma_{\mathsf{GS}}$ and $\mathcal{T} = \mathcal{T}_{\mathsf{node}} \oplus \mathcal{T}_{\mathsf{GS}}$.

We show that our reduction of $\mathsf{SLL}\mathbb{B}$ to GRASS allows us to decide satisfiability of conjunctions $H \wedge G$ of $\mathsf{SLL}\mathbb{B}$ formulas H with quantifier-free Σ_{node}-formulas G. Such conjunctions are interpreted in Σ-interpretations, as expected. Given such a conjunction $H \wedge G$, the decision procedure checks $\mathcal{T}$-satisfiability of the formula $\mathit{reduce}(\mathit{Tr}_X(H)) \wedge G$, where $X \in \mathcal{X}$ does not appear in G. This check is implemented using a Nelson-Oppen combination of the decision procedure for $\mathcal{T}_{\mathsf{GS}}$ and the decision procedure for $\mathcal{T}_{\mathsf{node}}$. To show the completeness of this combination procedure, let $\mathcal{T}_{\mathsf{SL}}$ be the Σ_{GS}-theory defined as follows:

$$\mathcal{T}_{\mathsf{SL}} = \{\, \mathcal{A}|_{\Sigma_{\mathsf{GS}}} \mid \exists H \in \mathcal{H}, \mathcal{A} \in \mathcal{T}_{\mathsf{GS},\mathcal{X}}, X \in \mathcal{X}.\ \mathcal{A}, X \models_{\mathsf{SL}} H \,\}$$

We call $\mathcal{T}_{\mathsf{SL}}$ the theory of $\mathsf{SLL}\mathbb{B}$. Completeness then follows from the following theorem.

Theorem 6. *The theory $\mathcal{T}_{\mathsf{SL}}$ is stably infinite with respect to sort* node.

Theorem 6 follows from the fact that the theory of the fragment of GRASS that is defined by the translation function Tr is stably infinite with respect to sort node. Incidentally, this is not true for the full theory of GRASS. For example, the GRASS formula $\{x.\, x = y\} = \mathcal{U}$ has only models where the interpretation of sort node has cardinality 1. Theory combination for the full theory of GRASS is still possible using a more complex combination procedure that requires GRASS to be extended with linear cardinality constraints [35].

6 Extensions

In this section, we describe several extensions of GRASS to support symbolic execution of programs on GRASS formulas and more expressive separation logic fragments.

Arrays. One advantage of our approach is that it enables the use of separation logic in existing verification tools that already provide backends to SMT solvers, without requiring symbolic execution engines for separation logic. However, we then need a form of symbolic execution for GRASS formulas that is supported by existing tools. In particular, the logic must be able to express the effect of heap updates concisely. We can do this by extending GRASS with a theory of arrays to represent mutable data structure fields. That is, we model fields as arrays whose indices and elements are of sort node. For this purpose, we extend the signature Σ_{GS} with an additional sort field, and additional function symbols $\mathsf{sel} : \mathsf{field} \times \mathsf{node} \to \mathsf{node}$ and $\mathsf{upd} : \mathsf{field} \times \mathsf{node} \times \mathsf{node} \to \mathsf{node}$ to model field reads and writes. Also, the reachability predicate will now be of the form $\bullet \xrightarrow{\bullet \setminus \bullet} \bullet : \mathsf{field} \times \mathsf{node} \times \mathsf{node} \times \mathsf{node}$, taking a field as additional parameter. It follows from results in [33] that the quantifier-free satisfiability problem for this extension remains decidable in NP. In Appendix B, we show how to use this extension to decide validity of verification conditions with SL assertions.

Beyond singly-linked lists. To support SL fragments with inductive predicates for more diverse data structures, we consider several non-disjoint extensions of GRASS and then extend the translation function for the additional inductive predicates appropriately. For example, suppose we consider heap structures for list nodes consisting of two fields: a pointer n to the next node in the list and a data field d storing an integer value. Now suppose we want to extend the SLL$\mathbb{B}$ with an inductive predicate $\mathsf{sls}(x, y)$ representing a heap region consisting of a list segment from x to y, whose data values are sorted. Automated reasoning about formulas with such a predicate is more difficult to achieve using conventional SL provers because the predicate relates the memory layout with constraints on the stored data. We can easily support such a predicate in our approach by relying on the capabilities of the underlying SMT solver. To extend our translation from Section 5 to the sorted list predicate it suffices to define:

$$str_Y(\mathsf{sls}(x,y)) = (x \xrightarrow{n} y \land \forall z, w \in Y.\, z \xrightarrow{n} w \Rightarrow d(z) \leqslant d(w),\ Y = Btwn(x,y))$$

Under mild assumptions, it follows from results in [23] that the quantified constraint expressing the sortedness property is a local theory extension [32] and remains in a decidable fragment. This allows us to reduce reasoning about sorted lists to reasoning in a disjoint combination of $\mathcal{T}_{\mathsf{GS}}$ with the theory of free function symbols (for the data field) and the theory of linear arithmetic. Similar reductions can be given for predicates encoding data structures with more complex linking patterns, such as doubly-linked lists, lists with head pointers, nested lists, etc. For example, the translation for the usual doubly-linked list predicate $\mathsf{dlls}(x, a, y, b)$ over forward pointer field n and backward pointer field p (see, e.g. [4]) is as follows:

$$\begin{aligned} str_Y(\mathsf{dlls}(x,a,y,b)) = (\, & x \xrightarrow{n} y \land (x = y \land a = b \lor p(x) = a \land n(b) = y \land b \in Y) \land \\ & \forall z \in Y.\, n(z) \in Y \Rightarrow p(n(z)) = z,\ Y = Btwn(x,y)\ ) \end{aligned}$$

The quantified constraint in the translation again constitutes a local theory extension that remains decidable and can be handled efficiently. One can also provide translations for inductive predicates describing tree data structures by using an appropriate first-order theory for reachability in trees, such as the one presented in [34].

7 Frame Inference and Abduction

Many operations in SL-based program analyses, including the application of the frame rule, involve more general forms of entailment tests referred to as frame inference [7,18] and abduction [11]. The *frame inference problem* is to compute for a pair of SLL$\mathbb{B}$ formulas (H, G), a formula F such that $H \models_{\mathsf{SL}} G * F$ holds, if such F exists. We call F the *frame* and we denote such frame inference problems by $H \models_{\mathsf{SL}} G * F?$. Likewise, the *abduction problem* is to find an *anti-frame* F for (H, G) such that $H * F \models_{\mathsf{SL}} G$. We denote abduction problems by $H * F? \models_{\mathsf{SL}} G$. In the following, we explain how to solve frame inference and abduction problems using our decision procedure for GRASS in combination with a model-generating SMT solver.

Inverse translation. Our technique for frame inference and abduction uses a characterization of a GRASS formula F in terms of a finite set of partial interpretations

$\mathsf{PMod}_X(F)$ that we obtain from the models of F, where X is some set variable occurring in F. We use this set of partial models to define an *inverse translation function* that maps F to an $\mathsf{SLL}\mathbb{B}$ formula $Tr_X^{-1}(F)$. The purpose of the set variable X is to carve out a specific partial substructures of each model of F that is then captured by $Tr_X^{-1}(F)$.

We start by defining $\mathsf{PMod}_X(F)$. Let F be a GRASS formula and let $X \in \mathcal{X}$ be a set variable. Let further $N \subseteq \mathcal{X}$ be the set of all free variables of sort node in F and define $V = N \cup \{X\}$. For $\mathcal{A} \in \mathcal{T}_{\mathsf{GS},\mathcal{X}}$, define $N^{\mathcal{A}} = \{\, x^{\mathcal{A}} \mid x \in N \,\}$. Further, define $\mathcal{A}_{F,X}$ as the partial Σ_{GS}-interpretation for variables V that is obtained from $\mathcal{A}$ by defining $\mathsf{node}^{\mathcal{A}_{F,X}} = N^{\mathcal{A}}$ and restricting the interpretation of all symbols $\Omega_{\mathsf{GS}} \cup \Pi_{\mathsf{GS}} \cup V$ in $\mathcal{A}$ to $N^{\mathcal{A}}$. We then define $\mathsf{PMod}_X(F) = \{\, \mathcal{A}_{F,X} \mid \mathcal{A} \models_{\mathsf{GS}} F \,\}$.

To simplify the presentation, we restrict ourselves to a specific class of GRASS formulas: we say that F is X*-closed*, if for all $\mathcal{B} \in \mathsf{PMod}_X(F)$ and $u \in X^{\mathcal{B}}$, $h^{\mathcal{B}}(u)$ is defined or there exists some $v \in \mathsf{node}^{\mathcal{B}}$ such that $v \neq u$ and $u \to^{\mathcal{B}} v$. In the following, we assume that F is X-closed. The inverse translation can be generalized to arbitrary GRASS formulas. However, this requires the introduction of additional Skolem constants (respectively, explicit existential quantifiers in $\mathsf{SLL}\mathbb{B}$).

Let $\mathcal{B} \in \mathsf{PMod}(F)$. Define a partial function $succ^{\mathcal{B}} : \mathsf{node}^{\mathcal{B}} \rightharpoonup \mathsf{node}^{\mathcal{B}}$ as follows. For all $u \in \mathsf{node}^{\mathcal{B}}$, let $succ^{\mathcal{B}}(u) = v$, where $v \in \mathsf{node}^{\mathcal{B}}$ is the unique node such that $v \neq u$ and for all $w \in \mathsf{node}^{\mathcal{B}}$, if $w \neq u$ and $u \to^{\mathcal{B}} w$, then $u \xrightarrow{h\backslash w}{}^{\mathcal{B}} v$, if such a node v exists. Otherwise, $succ^{\mathcal{B}}(u)$ is undefined. For every $u \in \mathsf{node}^{\mathcal{B}}$, let $x_u \in N$ such that $x_u^{\mathcal{B}} = u$. Now define a spatial conjunction $tr_X^{-1}(\mathcal{B})$ of $\mathsf{SLL}\mathbb{B}$ atoms as follows. First, for all distinct $x, y \in N$, if $x^{\mathcal{B}} = y^{\mathcal{B}}$, then $tr_X^{-1}(\mathcal{B})$ contains the spatial conjunct $x = y$, otherwise it contains $x \neq y$. Second, for every $u \in X^{\mathcal{B}}$, $tr_X^{-1}(\mathcal{B})$ contains a spatial conjunct Σ_u defined as follows: if $h^{\mathcal{B}}(u) = v$ for some $v \in N^{\mathcal{A}}$, then $\Sigma_u = (x_u \mapsto x_v)$; otherwise, $\Sigma_u = \mathsf{ls}(x_u, x_v)$ where v is such that $succ^{\mathcal{B}}(u) = v$.

Note that the set $\mathsf{PMod}(F)$ is finite up to isomorphism. If it is empty, we define $Tr_X^{-1}(F) = \mathsf{false}$. Otherwise, let $\mathcal{B}_1, \ldots, \mathcal{B}_n$ be representatives of all isomorphism classes of $\mathsf{PMod}(F)$. Then define $Tr_X^{-1}(F) = tr_X^{-1}(\mathcal{B}_1) \vee \ldots \vee tr_X^{-1}(\mathcal{B}_n)$.

The following lemma states the correctness of this inverse translation function.

Lemma 7. *Let $X \in \mathcal{X}$ and F be an X-closed GRASS formula. Then for all $\mathcal{A} \in \mathcal{T}_{\mathsf{GS},\mathcal{X}}$:*

1. *if $\mathcal{A} \models_{\mathsf{GS}} F$, then $\mathcal{A}, X \models_{\mathsf{SL}} Tr_X^{-1}(F)$, and*
2. *if $\mathcal{A}, X \models_{\mathsf{SL}} Tr_X^{-1}(F)$, then $\mathcal{A}_{F,X} \in \mathsf{PMod}_X(F)$.*

Note that we can compute $\mathsf{PMod}_X(F)$ by solving the All-SAT problem for F using a model-generating SMT solver that implements the decision procedure for $\mathcal{T}_{\mathsf{GS}}$. From each model $\mathcal{A}$ of F that is generated by the solver, we compute the partial model $\mathcal{A}_{F,X}$. This partial model then serves as a blocking clause for the solver to eliminate all models of F from the search space that are mapped to the isomorphism class of $\mathcal{A}_{F,X}$. If we apply this technique without further optimizations, then the computed set $\mathsf{PMod}_X(F)$ (and hence the formula $Tr_X^{-1}(F)$) will be (worst-case) exponential in the size of F. This is because each partial model fixes an arrangement of equalities between the variables in N. The enumeration process can be improved by generalizing each computed partial model before it is further processed, e.g., by dropping inequalities that are not implied by F. Only the generalized partial models are then used as blocking clauses, respectively, in the inverse translation function.

$$\begin{tikzcd} \mathcal{B}_1 : x \arrow[r] & y{,}z \ , \ Z = \emptyset & \mathcal{B}_2 : x \arrow[r] & y \arrow[r, dashed] & z \ , \ Z = \{y\} \end{tikzcd}$$

Fig. 5. The set $\mathsf{PMod}_Z(F)$ for formula F in Example 8

Example 8. Consider the GRASS formula

$$F \equiv x \neq z \wedge x \xrightarrow{h} z \wedge h(x) = y \wedge X = \mathit{Btwn}(x, z) \wedge Y = \{x\} \wedge Z = X \backslash Y$$

The set X contains all nodes on the path from x to z, excluding z. The path exists because of $x \xrightarrow{h} z$. Hence Z contains all nodes in X except for x itself. The set $\mathsf{PMod}_Z(F)$ consists of two isomorphism classes of partial models represented by $\mathcal{B}_1$ and $\mathcal{B}_2$ depicted in Figure 5. The solid edges denote the interpretation of h, the dashed edges denote the partial function $\mathit{succ}^{\mathcal{B}_i}$ for the nodes on which h is undefined. The partial models show that F is Z-closed. We then have $\mathit{tr}_Z^{-1}(\mathcal{B}_1) = x \neq z * x \neq y * y = z$ and $\mathit{tr}_Z^{-1}(\mathcal{B}_2) = x \neq z * x \neq y * y \neq z * \mathsf{ls}(y, z)$. From this we obtain $\mathit{Tr}_Z^{-1}(F) = \mathit{tr}_Z^{-1}(\mathcal{B}_1) \vee \mathit{tr}_Z^{-1}(\mathcal{B}_2) \equiv x \neq z * x \neq y * \mathsf{ls}(y, z)$.

Solving frame inference and abduction problems. We now show how we use the inverse translation function to solve frame inference problems. This technique can then be easily adapted for abduction.

A formula $H \in \mathcal{H}$ is called *positive* if it does not contain negations. For a positive formula H, we always have that $\mathit{Tr}_X(H)$ is X-closed. To ensure that we can use the inverse translation function from the previous section, we therefore restrict ourselves to frame inference problems in the positive fragment of SLL$\mathbb{B}$.

Let H and G be two positive SLL$\mathbb{B}$ formulas and suppose that $H \models_{\mathsf{SL}} G * F?$ has a solution. To compute a solution, define the GRASS formula

$$\mathit{Frame}_Z(H, G) = \mathit{Tr}_X(H) \wedge \mathit{Tr}_Y(G) \wedge Z = X \backslash Y$$

where $X, Y, Z \in \mathcal{X}$ are distinct set variables. Note that, the set variable Z describes the footprint of the frame. Moreover, the formula $\mathit{Frame}_Z(H, G)$ is Z-closed. Hence, the SLL$\mathbb{B}$ formula $\mathit{Tr}_Z^{-1}(\mathit{Frame}_Z(H, G))$ is a valid frame for (H, G).

Theorem 9. *For all positive SLL$\mathbb{B}$ formulas H and G, and $Z \in \mathcal{X}$, if $H \models_{\mathsf{SL}} G * F?$ has a solution, then $H \models_{\mathsf{SL}} G * \mathit{Tr}_Z^{-1}(\mathit{Frame}_Z(H, G))$. Moreover, if $H \models_{\mathsf{SL}} G * F'$ for some F', then $\mathit{Tr}_Z^{-1}(\mathit{Frame}_Z(H, G)) \models_{\mathsf{SL}} F'$.*

It remains to check whether $H \models_{\mathsf{SL}} G * F?$ has a solution. For this purpose, define the GRASS formula

$$\mathit{NoFrame}(H, G) = \mathit{Tr}_X(H) \wedge \mathit{Trf}_X(\neg G)$$

where $X \in \mathcal{X}$ and Trf is like Tr, except that the constraints $X = Y$ in the case for $\mathit{tr}_X(\Sigma)$ are replaced by $Y \subseteq X$.

Theorem 10. *For all positive SLL$\mathbb{B}$ formulas H and G, the instance $H \models_{\mathsf{SL}} G * F?$ of the frame inference problem has a solution iff $\mathit{NoFrame}(H, G)$ is unsatisfiable.*

We check unsatisfiability of $\mathit{NoFrame}(H, G)$ using our decision procedure for GRASS. If the check succeeds, we compute the solution according to Theorem 9. In order to adapt this technique for solving abduction problems $H * F? \models_{\mathsf{SL}} G$, it suffices to replace $Z = X \backslash Y$ in $\mathit{Frame}_Z(H, G)$ by $Z = Y \backslash X$, and the constraints $Y \subseteq X$ in $\mathit{NoFrame}(H, G)$ by $X \subseteq Y$.

8 Implementation and Experiments

We have implemented our decision procedure for GRASS together with the translation of SLL$\mathbb{B}$ in a prototype prover. We have further developed a verification tool called GRASShopper that builds on top of this prover[4]. Currently we use Z3 [16] as the underlying SMT solver because our implementation relies on Z3's model-based quantifier instantiation mechanism (MBQI).

To decide satisfiability of a GRASS formula F, we proceed as described in Appendix A and generate an equisatisfiable Σ_{GS}-formula G, which we then check for $\mathcal{T}_{\mathsf{GS}}$-satisfiability. We have not yet implemented a dedicated solver for the theory of graph reachability $\mathcal{T}_{\mathsf{G}}$. Instead, we use the finite first-order axiomatizations of this theory that are described in [23, 33]. To decide satisfiability of G, we conjoin the theory axioms with G and then partially instantiate quantified variables in the resulting formula with ground terms occurring in G. We only instantiate variables that occur below function symbols in the axioms of $\mathcal{T}_{\mathsf{G}}$. This keeps the size of the formulas that are given to the SMT solver reasonably small. The partial instantiation is guaranteed to be complete because $\mathcal{T}_{\mathsf{G}}$ is a local theory [32]. Details about this result can also be found in [33]. The partially instantiated axioms are in the EPR fragment of first-order logic (aka the Bernays-Schönfinkel-Ramsey class). The EPR fragment can be decided quite efficiently using Z3's MBQI mechanism. Stratified sets can be encoded directly in Z3 using combinatory array logic [17]. However, according to the Z3 developers, the array theory does currently not behave well with MBQI. We therefore also partially instantiate the axioms of stratified sets to remain in the EPR fragment.

Our tool GRASShopper uses the prover to verify list-manipulating programs written in a simple imperative language. The programs are expected to be annotated with procedure contracts and loop invariants expressed in separation logic. Each procedure is verified in isolation. To handle loops and procedure calls efficiently, the tool implements a frame rule that avoids explicit inference of frames. Instead, we encode frames implicitly in the formula that is given to the SMT solver. More details about this implementation can be found in Appendix B. Currently, GRASShopper supports singly, doubly-linked, and sorted list predicates. We are planing to add support for user-defined predicates in the future. Since our prover yields a decision procedure for checking the generated verification conditions, we use the SMT solver to produce counterexamples for faulty programs, which our tool can then visualize.

We have applied our prototype to verify partial correctness specifications (including absence of run-time errors) of typical list-manipulating programs, including sorting algorithms. The considered programs contain loops and (recursive) procedure calls.

[4] The tool is available at `http://cs.nyu.edu/wies/software/grasshopper`.

program	sl		dl		rec sl		sls		program	sl		dl		rec sl		sls	
	#	t	#	t	#	t	#	t		#	t	#	t	#	t	#	t
concat	4	0.1	5	1.3	6	0.6	5	0.2	insert	6	0.2	5	1.5	5	0.2	6	0.4
copy	4	0.2	4	3.9	6	0.8	7	3.5	reverse	4	0.1	4	0.5	6	0.2	4	0.2
filter	7	0.6	5	1.1	8	0.4	5	1.1	remove	8	0.2	8	0.8	7	0.2	7	0.5
free	5	0.1	5	0.3	4	0.1	5	0.1	traverse	4	0.1	5	0.3	3	0.1	4	0.2
insertion sort							10	0.7	double all							7	2.2
merge sort							25	6.8	pairwise sum							10	20

Table 1. Experimental results for the verification of list-manipulating programs. The columns marked "sl" refer to singly-linked list versions of the benchmarks, the "dl" columns to doubly-linked lists, and the "rec sl" columns to recursive implementations with singly-linked lists. Finally, the columns "sls" refer to sorted singly-linked lists. The columns "#" give the number of queries to the SMT solver and the "t" columns refer to the total running time in seconds.

Some of the programs consist of multiple procedures. Table 1 shows the results of the experiments. For example, the program "pairwise sum" takes two sorted lists as input and creates a new list whose entries are the pairwise sums of the entries in the input lists. We then show that the resulting list is again sorted. For the sorting algorithms, we proved that the output list is sorted but we did not check that it is a permutation of the input list. To verify the programs manipulating doubly-linked and sorted lists we used a Nelson-Oppen combination of $\mathcal{T}_{\mathsf{GS}}$ with the theory of equality and uninterpreted function symbols, and the theory of linear arithmetic.

9 Conclusions

We presented a reduction of decidable separation logic fragments to a decidable first-order logic fragment called GRASS. Our reduction enables the seamless integration of an SL prover into an SMT solver, which has promising applications in program verification. We demonstrated the feasibility of our approach using a prototype implementation. Future directions include the development of dedicated theory solvers for graph reachability and stratified sets, which underlie the decision procedure for GRASS.

References

1. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, volume 5142 of *LNCS*, pages 387–411, 2008.
2. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.
3. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, pages 171–177, 2011.
4. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.
5. J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, 2004.

6. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.
7. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.
8. J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory Safety for Systems-Level Code. In *CAV*, 2011.
9. F. Bobot and J.-C. Filliâtre. Separation predicates: a taste of separation logic in first-order logic. In *ICFEM*, 2012.
10. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In *ATVA*, volume 7561 of *LNCS*, pages 167–182. Springer, 2012.
11. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.
12. C. Calcagno and M. Hague. From separation logic to first-order logic. In *FoSSaCs'05*, pages 395–409. Springer, 2005.
13. C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007.
14. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *TACAS*, 2007.
15. B. Cook, C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*. Springer, 2011.
16. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
17. L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, pages 45–52. IEEE, 2009.
18. D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, pages 213–226. ACM, 2008.
19. K. Dudka, P. Peringer, and T. Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *CAV*, 2011.
20. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
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*, volume 6617 of *LNCS*, pages 41–55, 2011.
22. I. T. Kassios. The dynamic frames theory. *Formal Asp. Comput.*, 23(3):267–288, 2011.
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. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM TOPLAS*, 1(2):245–257, 1979.
25. 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.
26. M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3), 2012.
27. J. A. N. Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, pages 556–566. ACM, 2011.
28. J. A. N. Pérez and A. Rybalchenko. Separation Logic Modulo Theories. Technical Report arXiv:1303.2489, arXiv.org, 2013.
29. S. Ranise, C. Ringeissen, and C. G. Zarba. Combining data structures with nonstably infinite theories using many-sorted logic. In *FroCos*, volume 3717 of *LNCS*, pages 48–64, 2005.
30. S. Rosenberg, A. Banerjee, and D. A. Naumann. Decision procedures for region logic. In *VMCAI*, volume 7148 of *LNCS*, pages 379–395, 2012.

31. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2, 2012.
32. V. Sofronie-Stokkermans. Hierarchic reasoning in local theory extensions. In *CADE*, pages 219–234, 2005.
33. N. Totla and T. Wies. Complete instantiation-based interpolation. In *POPL*. ACM, 2013.
34. 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.
35. T. Wies, R. Piskac, and V. Kuncak. Combining theories with shared set operations. In *FroCoS*, volume 5749 of *LNCS*, pages 366–382. Springer, 2009.
36. 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.
37. C. G. Zarba. Combining sets with elements. In N. Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 762–782. Springer, 2004.

A Deciding Satisfiability of GRASS

In the following, we present the decision procedure for the satisfiability problem of GRASS. The decision procedure works by reduction to the quantifier-free satisfiability problem of $\mathcal{T}_{\mathsf{GS}}$. The latter is decidable and can be implemented in an SMT solver. To show this, we need the following property.

Lemma 11. *The theories $\mathcal{T}_{\mathsf{G}}$ and $\mathcal{T}_{\mathsf{S}}$ are stably infinite with respect to sort* node.

Since disjoint combinations of stably infinite theories are stably infinite, we immediately obtain:

Corollary 12. *The theory $\mathcal{T}_{\mathsf{GS}}$ is stably infinite with respect to sort* node.

The theory $\mathcal{T}_{\mathsf{G}}$ is not first-order definable because of the finiteness constraints on the edge relation h and the fact that reachability predicates are defined in terms of transitive closure of relations. However, there exists a finite set of first-order axioms whose finite models are all in $\mathcal{T}_{\mathsf{G}}$. This axiomatization is the key to solving the quantifier-free satisfiability problem of $\mathcal{T}_{\mathsf{G}}$, which is NP-complete. A corresponding decision procedure is presented, e.g., in [33]. From results in [37] and Lemma 11 it then follows that $\mathcal{T}_{\mathsf{GS}}$ is decidable using a standard Nelson-Oppen combination of the decision procedures for $\mathcal{T}_{\mathsf{G}}$ and $\mathcal{T}_{\mathsf{S}}$. The decision procedure obtained in this way remains in NP.

Theorem 13. *The quantifier-free satisfiability problem of $\mathcal{T}_{\mathsf{GS}}$ is NP-complete.*

We now describe the decision procedure for GRASS. Let *reduce* be the function that maps a GRASS formula F to a quantifier-free first-order formula G as follows:

1. Eliminate all disequalities between set expressions in F by transforming F into negation normal form and then exhaustively applying the following rewrite rule:

$$S_1 \neq S_2 \rightsquigarrow x \in S_1 \setminus S_2 \cup S_2 \setminus S_1 \qquad \text{where } x \in \mathcal{X} \text{ fresh}$$

   Let F_1 be the resulting GRASS formula.

2. Eliminate all set comprehensions in F_1 by exhaustively applying the following rewrite rule:
$$C[\{x. R\}] \rightsquigarrow C[X] \wedge (\forall x. x \in X \Leftrightarrow R) \qquad \text{where } X \in \mathcal{X} \text{ fresh}$$
Let F_2 be the resulting Σ_{GS}-formula.
3. Instantiate all universal quantifiers in F_2 as follows. Let $t_1, \dots, t_n$ be the terms of sort node appearing in F_2 that do not contain quantified variables. Then exhaustively apply the following rewrite rule to F_2:
$$(\forall x. x \in X \Leftrightarrow R) \rightsquigarrow (t_1 \in X \Leftrightarrow R[t_1/x]) \wedge \dots \wedge (t_n \in X \Leftrightarrow R[t_n/x])$$
Let G be the resulting quantifier-free Σ_{GS}-formula.

Lemma 14. *For all GRASS formulas F, F is satisfiable iff $reduce(F)$ is $\mathcal{T}_{\mathsf{GS}}$-satisfiable.*

Lemma 14 implies that satisfiability of a GRASS formula F can be decided by checking $\mathcal{T}_{\mathsf{GS}}$-satisfiability of its reduction $reduce(F)$. The size of the formula $reduce(F)$ is at most quadratic in the size of F. Together with Theorem 13 we then obtain Theorem 2.

Example 15. Consider the GRASS formula
$$F \equiv \{x. x \xrightarrow{h} y\} = \mathcal{U} \wedge y \xrightarrow{h} z \wedge \neg(w \xrightarrow{h} z)$$
This formula is unsatisfiable, which we prove with our decision procedure. After step 2 of the reduction, we obtain the first-order formula:
$$F_2 \equiv S = U \wedge y \xrightarrow{h} z \wedge \neg(w \xrightarrow{h} z) \wedge (\forall x. x \in S \Leftrightarrow x \xrightarrow{h} y) \wedge (\forall x. x \in U \Leftrightarrow x = x)$$
Instantiating the quantifiers in step 3 yields the quantifier-free formula
$$\begin{aligned} G \equiv\ & S = U \wedge y \xrightarrow{h} z \wedge \neg(w \xrightarrow{h} z) \wedge \\ & (y \in S \Leftrightarrow y \xrightarrow{h} y) \wedge (z \in S \Leftrightarrow z \xrightarrow{h} y) \wedge (w \in S \Leftrightarrow w \xrightarrow{h} y) \wedge \\ & (y \in U \Leftrightarrow y = y) \wedge (z \in U \Leftrightarrow z = z) \wedge (w \in U \Leftrightarrow w = w) \end{aligned}$$
To see that this formula is unsatisfiable in $\mathcal{T}_{\mathsf{GS}}$, we simplify G to the equivalent formula:
$$\begin{aligned} G' \equiv\ & S = U \wedge y \xrightarrow{h} z \wedge \neg(w \xrightarrow{h} z) \wedge y \in U \wedge z \in U \wedge w \in U \wedge \\ & (y \in S \Leftrightarrow y \xrightarrow{h} y) \wedge (z \in S \Leftrightarrow z \xrightarrow{h} y) \wedge (w \in S \Leftrightarrow w \xrightarrow{h} y) \end{aligned}$$
Note that $S = U \wedge w \in U$ implies $w \in S$, which in turn implies $w \xrightarrow{h} y$. Together with $y \xrightarrow{h} z$ and transitivity of $\rightarrow$ this implies $w \xrightarrow{h} z$, which gives the contradiction.

B Verifying Heap-Manipulating Programs

The main use case of decision procedures for fragments of separation logic is to automate the verification of heap-manipulating programs against SL specifications. In this section, we introduce a simple imperative language of list-manipulating programs and then explain how we automate the verification of such programs using the techniques developed in this paper.

B.1 List-Manipulating Guarded Commands

We define a language of guarded commands as shown in Fig. 6. Note that **while** loops are annotated with loop invariants expressed in $\mathsf{SLL}\mathbb{B}$. As expected, a loop invariant only needs to describe the footprint of the loop, i.e., those nodes in the heap that are accessed by the loop.

$x, y \in \mathcal{X}$	$E ::= x \mid \mathsf{nil}$	heap expressions
$I \in \mathcal{H}$	$G ::= E = E \mid E \neq E$	Boolean expressions
	$C ::= x := E$	variable assignment
	$\mid x := [y]$	heap lookup
	$\mid [x] := E$	heap update
	$\mid \mathbf{new}(x)$	allocation
	$\mid \mathbf{dispose}(x)$	deallocation
	$\mid \mathbf{assume}(G)$	guard
	$\mid C\ C$	sequential composition
	$\mid C \mathbin{[]} C$	non-deterministic choice
	$\mid \{I\}\ \mathbf{while}(G)\ C$	while loop with invariant

Fig. 6. Syntax of list-manipulating guarded commands

We give an operational semantics of a command C in terms of a transition relation $\overset{C}{\rightsquigarrow}$ on states. States are heap interpretations. In addition, we have the special *error state* fail, which indicates a failed computation. We consider a dedicated set variable Alloc whose interpretation represents the set of all currently allocated heap nodes. All program states interpret the constant nil as an unallocated node. The transition relation is then the smallest relation satisfying the following conditions: we have $\mathcal{A} \overset{C}{\rightsquigarrow} \mathcal{B}$ if $\mathcal{A} = \mathcal{B} = \mathsf{fail}$, or $\mathcal{A}$ is a heap interpretation and the following conditions hold, depending on the structure of P:

1. $(C = x := E)$ $\mathcal{B} = \mathcal{A}[x \mapsto E^{\mathcal{A}}]$;
2. $(C = x := [y])$ $\mathcal{B} = \mathcal{A}[x \mapsto h^{\mathcal{A}}(y^{\mathcal{A}})]$ if $y^{\mathcal{A}} \in \mathsf{Alloc}^{\mathcal{A}}$, and otherwise $\mathcal{B} = \mathsf{fail}$;
3. $(C = [x] := E)$ $\mathcal{B} = \mathcal{A}[h \mapsto h^{\mathcal{A}}[x^{\mathcal{A}} \mapsto E^{\mathcal{A}}]$ if $x^{\mathcal{A}} \in \mathsf{Alloc}^{\mathcal{A}}$, and otherwise $\mathcal{B} = \mathsf{fail}$;
4. $(C = \mathbf{new}(x))$ $\mathcal{B} = \mathcal{A}[\mathsf{Alloc} \mapsto \mathsf{Alloc}^{\mathcal{A}} \cup \{u\}, x \mapsto u]$ for some $u \in \mathsf{node}^{\mathcal{A}} \setminus (\mathsf{Alloc}^{\mathcal{A}} \cup \{\mathsf{nil}^{\mathcal{A}}\})$;
5. $(C = \mathbf{dispose}(x))$ $\mathcal{B} = \mathcal{A}[\mathsf{Alloc} \mapsto \mathsf{Alloc}^{\mathcal{A}} \setminus \{x^{\mathcal{A}}\}]$ if $x^{\mathcal{A}} \in \mathsf{Alloc}^{\mathcal{A}}$, and otherwise $\mathcal{B} = \mathsf{fail}$;
6. $(C = \mathbf{assume}(G))$ $\mathcal{B} = \mathcal{A}$ and $\mathcal{A} \models G$;
7. $(C = C_1\ C_2)$ $\mathcal{A} \overset{C_1}{\rightsquigarrow} \circ \overset{C_2}{\rightsquigarrow} \mathcal{B}$; and
8. $(C = C_1 \mathbin{[]} C_2)$ $\mathcal{A} \overset{C_1}{\rightsquigarrow} \mathcal{B}$ or $\mathcal{A} \overset{C_2}{\rightsquigarrow} \mathcal{B}$.

9. $(C = \{I\}$ **while**$(G)\ C')$ $\mathcal{A} \models \neg G$ and $\mathcal{A} = \mathcal{B}$, or $\mathcal{A} \models G$, $\mathcal{A} \overset{C'}{\rightsquigarrow} \mathcal{A}'$, and $\mathcal{A}' \overset{C}{\rightsquigarrow} \mathcal{B}$.

We consider Hoare triples of the form $\{P\}C\{Q\}$, where C is a guarded command and P, Q are $\mathsf{SLL}\mathbb{B}$ formulas. We focus on partial correctness, i.e., a Hoare triple $\{P\}C\{Q\}$ is valid iff for all normal program states $\mathcal{A}$, if $\mathcal{A}, \mathsf{Alloc} \models_{\mathsf{SL}} P$ and $\mathcal{A} \overset{C}{\rightsquigarrow} \mathcal{B}$ for some $\mathcal{B}$, then $\mathcal{B} \neq \mathsf{fail}$ and $\mathcal{B}, \mathsf{Alloc} \models_{\mathsf{SL}} Q$. We say that a Hoare triple is *conditionally valid* if there exists a Hoare proof of its validity that uses the annotated invariants in the command C.

Example 16. The following Hoare triple consists of a guarded command that computes the in-place reversal of a singly-linked list pointed to by x.

```
{ls(x, nil)}
y := nil
{ls(x, nil) ∗ ls(y, nil)}
while(x ≠ nil)
    t := x
    x := [x]
    [t] := y
    y := t
{ls(y, nil)}
```

The precondition specifies that x points to an acyclic singly-linked list. The postcondition specifies that, after termination of the command, the resulting heap consists only of a singly acyclic list segment pointed to by y.

B.2 Checking Conditional Validity of Hoare Triples

To automatically check the conditional validity of a Hoare triple $\{P\}C\{Q\}$, we generate a verification condition expressed in GRASS. This verification condition is unsatisfiable iff the Hoare triple is conditionally valid.

Loop-free commands. We start by showing that it is possible to compute verification conditions in the form of symbolic weakest preconditions using the machinery that is provided by tools such as Boogie [2]. For this purpose, we restrict ourselves to loop-free guarded commands. Let F be a GRASS formula and let C be a loop-free guarded command, the symbolic weakest precondition $\mathsf{wp}(C, F)$ of F and C describes the set of all program states $\mathcal{A}$ such that for all program states $\mathcal{B}$, if $\mathcal{A} \overset{C}{\rightsquigarrow} \mathcal{B}$, then $\mathcal{B} \neq \mathsf{fail}$ and $\mathcal{B} \models F$. It is defined recursively on the structure of C as shown in Figure 7. Note that we use the extension of GRASS with arrays, to handle pointer updates concisely.

The following theorem states that we can use symbolic weakest preconditions to decide the validity of Hoare triples with loop-free guarded commands.

Theorem 17. *A Hoare triple $\{P\}C\{Q\}$ is valid iff the GRASS formula $\neg\mathsf{Alloc}(\mathsf{nil}) \wedge Tr_{\mathsf{Alloc}}(P) \wedge \neg\mathsf{wp}(C, \neg\, Tr_{\mathsf{Alloc}}(\neg Q))$ is unsatisfiable.*

$$\begin{aligned}
\mathsf{wp}(x := E, F) &= F[E/x] \\
\mathsf{wp}(x := [y], F) &= y \in \mathsf{Alloc} \land F[\mathsf{sel}(h, E)/x] \\
\mathsf{wp}([x] := E, F) &= x \in \mathsf{Alloc} \land F[\mathsf{upd}(h, x, E)/h] \\
\mathsf{wp}(\textbf{new}(x), F) &= x' \notin \mathsf{Alloc} \land x' \neq \mathsf{nil} \Rightarrow F[x'/x, (\mathsf{Alloc} \cup \{x'\})/\mathsf{Alloc}] \\
&\qquad \text{where } x' \in \mathcal{X} \text{ fresh} \\
\mathsf{wp}(\textbf{dispose}(x), F) &= x \in \mathsf{Alloc} \land F[(\mathsf{Alloc} \backslash \{x\})/\mathsf{Alloc}] \\
\mathsf{wp}(\textbf{assume}(G), F) &= G \Rightarrow F \\
\mathsf{wp}(C_1\ C_2, F) &= \mathsf{wp}(C_1, \mathsf{wp}(C_2, F)) \\
\mathsf{wp}(C_1 \mathbin{[]} C_2, F) &= \mathsf{wp}(C_1, F) \land \mathsf{wp}(C_2, F)
\end{aligned}$$

Fig. 7. Symbolic weakest preconditions of GRASS formulas for loop-free guarded commands

The general case. To decide conditional validity of general Hoare triples, we use a form of symbolic forward execution with a special encoding of frame axioms that does not rely on frame inference. Our tool GRASShopper uses a straightforward generalization of this approach for a language that also supports procedures.

Given a Hoare triple $\{P\}C\{Q\}$ we define a function VC that computes the verification condition for $\{P\}C\{Q\}$. This function is shown in Fig. 8. The function VC is defined in terms of the auxiliary function vc. The function vc takes a guarded command C, a precondition F, and variable substitution σ as input. It returns a tuple (F', σ', VC_C) where F' is the strongest postcondition of F with respect to C, σ is a variable substitution, and VC_C is a verification condition that encodes checks for memory safety and inductiveness of all loop invariants in C. The function vc introduces fresh Skolem constants to represent the new values of updated variables. The input substitution σ provides the mapping of all variables to their most recent values in F. Similarly, the output substitution σ' provides the values of variables in the post states captured by F'. We denote by $id_{\mathcal{X}}$ the identity function on $\mathcal{X}$, which is the initial substitution given to vc.

The function vc is defined recursively on the structure of commands. The cases for atomic commands, sequential composition, and non-deterministic choice are straightforward. We discuss the verification condition generation for loops in more detail.

The first step in $vc(\{I\}\ \textbf{while}(G)\ C, F, \sigma)$ is to compute the verification condition VC_C for the loop body C. This is done using a recursive call to the function VC. Note that we use local reasoning to analyze the loop body, i.e., the recursive call considers only the nodes described by the loop invariant I as allocated. The computed verification condition is combined with the check that the invariant is implied by the current state F, yielding the formula VC'. Note that we use the translation function Trf from Section 7 to check that I describes a subheap of F. The next step is to compute the new substitution σ' for the values of variables in the post states. Here $mod(C)$ denotes the set of all variables that are modified by the loop body C.

The most tricky part is the computation of the post condition. Intuitively, we have to identify the part in F that corresponds to I and then replace it by a new version of I that now refers to the modified variables in σ', keeping the frame that is not modified by

$$\begin{aligned}
VC(\{P\}C\{Q\}) &= \text{let } (P', \sigma, VC_C) = vc(C, \mathsf{nil} \notin \mathsf{Alloc} \wedge Tr_{\mathsf{Alloc}}(P), id_{\mathcal{X}}) \\
&\quad\ \text{in } VC_C \vee (P' \wedge (Tr_{\mathsf{Alloc}}(\neg Q))\sigma) \\
vc(x := E, F, \sigma) &= (x' = E\sigma \wedge F, \sigma[x \mapsto x'], \mathsf{false}) \\
vc(x := [y], F, \sigma) &= ((x' = \mathsf{sel}(h, y))\sigma \wedge F, \sigma[x \mapsto x'], F \wedge (y \notin \mathsf{Alloc})\sigma) \\
vc([x] := E, F, \sigma) &= ((h' = \mathsf{upd}(h, x, E))\sigma \wedge F, \sigma[h \mapsto h'], F \wedge (x \notin \mathsf{Alloc})\sigma) \\
vc(\mathbf{new}(x), F, \sigma) &= \text{let } F' = F \wedge (x' \notin \mathsf{Alloc} \wedge x' \neq \mathsf{nil} \wedge \mathsf{Alloc}' = \mathsf{Alloc} \cup \{x'\})\sigma \\
&\quad\ \text{in } (F', \sigma[x \mapsto x', \mathsf{Alloc} \mapsto \mathsf{Alloc}'], \mathsf{false}) \\
vc(\mathbf{dispose}(x), F, \sigma,) &= \text{let } F' = F \wedge (\mathsf{Alloc}' = \mathsf{Alloc} \setminus \{x\})\sigma \\
&\quad\ \text{in } (F', \sigma[\mathsf{Alloc} \mapsto \mathsf{Alloc}'], F \wedge (x \notin \mathsf{Alloc})\sigma) \\
vc(\mathbf{assume}(G), F, \sigma) &= (G\sigma \wedge F, \sigma, \mathsf{false}) \\
vc(C_1\ C_2, F, \sigma) &= \text{let } (F_1, \sigma_1, VC_1) = vc(C_1, F, \sigma) \\
&\qquad\quad (F_2, \sigma_2, VC_2) = vc(C_2, F_1, \sigma_1) \\
&\quad\ \text{in } (F_2, \sigma_2, VC_1 \vee VC_2) \\
vc(C_1 \,[]\, C_2, F, \sigma) &= \text{let } (F_1, \sigma_1, VC_1) = vc(C_1, F, \sigma) \\
&\qquad\quad (F_2, \sigma_2, VC_2) = vc(C_2, F, \sigma) \\
&\qquad\quad (F', \sigma') = join(\sigma, \sigma_1, \sigma_2, F_1, F_2) \\
&\quad\ \text{in } ((F', \sigma', (VC_1 \vee VC_2)\sigma') \\
vc(\{I\}\ \mathbf{while}(G)\ C, F, \sigma) &= \text{let } VC_C = VC(C, \{I\}\, \mathbf{assume}(G)\ C\, \{I\}) \\
&\qquad\quad VC' = VC_C \vee F \wedge (Trf_{\mathsf{Alloc}}(\neg I))\sigma \\
&\qquad\quad \sigma' = \sigma[x \mapsto x' \mid x \in mod(C)] \\
&\qquad\quad F_1' = F \wedge (Tr_X(I))\sigma \wedge (\neg G \wedge Tr_{X'}(I))\sigma' \\
&\qquad\quad F_2' = FR(X, X', \sigma(\mathsf{Alloc}), \sigma'(\mathsf{Alloc}), \sigma(h), \sigma'(h)) \\
&\quad\ \text{in } (F_1' \wedge F_2', \sigma', VC') \\
FR(X, X', A, A', h, h') &= \ X \subseteq A \wedge A' = X' \uplus A \setminus X \wedge \mathsf{nil} \notin A' \wedge \\
&\quad\ \forall x.\, x \in A \setminus X \Rightarrow \mathsf{sel}(h', x) = \mathsf{sel}(h, x) \wedge \\
&\quad\ \forall x\, y\, z.\, x \xrightarrow{h \setminus ep_{X,h}(x)} y \Rightarrow (x \xrightarrow{h \setminus z} y \Leftrightarrow x \xrightarrow{h' \setminus z} y) \wedge \\
&\quad\ \forall x\, y\, z.\, x \in A \setminus X \wedge ep_{X,h}(x) = x \Rightarrow (x \xrightarrow{h \setminus z} y \Leftrightarrow x \xrightarrow{h' \setminus z} y) \\
join(\sigma, \sigma_1, \sigma_2, F_1, F_2) &= \text{let } M_1 = \{\, x \mid \sigma(x) = \sigma_1(x) \text{ and } \sigma_2(x) \neq \sigma(x) \,\} \\
&\qquad\quad M_2 = \{\, x \mid \sigma(x) \neq \sigma_1(x) \text{ and } \sigma_1(x) \neq \sigma_2(x) \,\} \\
&\qquad\quad F_1' = F_1 \wedge \bigwedge_{x \in M_1} \sigma_1(x) = \sigma_2(x) \\
&\qquad\quad F_2' = F_2 \wedge \bigwedge_{x \in M_2} \sigma_2(x) = \sigma_1(x) \\
&\qquad\quad \sigma' = \sigma[x \mapsto \sigma_2(x) \mid x \in M_1][x \mapsto \sigma_1(x) \mid x \in M_2] \\
&\quad\ \text{in } (F_1' \vee F_2', \sigma')
\end{aligned}$$

Fig. 8. Verification condition generation. All primed variables that are introduced in the function vc are assumed to be fresh

$$\forall x.\, x \xrightarrow{h} ep_{X,h}(x)$$

$$\forall x.\, ep_{X,h}(x) \in X \vee ep_{X,h}(x) = x$$

$$\forall x\, y.\, x \xrightarrow{h} y \wedge y \in X \Rightarrow ep_{X,h}(x) \in X \wedge x \xrightarrow{h \backslash y} ep_{X,h}(x)$$

Fig. 9. Axioms defining the entry point function

C intact. We could do this using our frame inference technique presented in Section 7. However, we here present an alternative technique that encodes the frame implicitly in the verification condition. Consider the formula F_1'. It is the conjunction of F with two formulas: (1) the invariant describing the footprint of the loop before the loop is entered, and (2) the invariant describing the footprint of the loop after the loop has terminated. The memory region of (1) is captured by the set variable X, and the memory region of (2) by X'. The constraint F_2' now relates X and X' and states that the frame, which is described by $\mathsf{Alloc} \backslash X$, does not change. We describe this formula in more detail.

The formula $FR(X, X', A, A', h, h')$ consists of three parts. First, it states that everything in X is allocated when the loop is entered, and that the allocated objects after termination of the loop are exactly those that are in the frame or in X'. Second, the first quantified constraint states that the heap graph does not change in the frame. In principle, these two parts are already a precise formalization of the frame condition. However, we need to eliminate the quantifier in the second part so that the SMT solver can effectively deal with the resulting verification condition. We do so using the ground instantiation techniques described in Section 8. However, once we eliminate the quantifier using finite ground instantiation, we lose completeness because the nodes in the frame can now change their relative reachability to each other. To alleviate this, we add two additional quantified constraints specifying that the relative order of nodes in the frame is preserved. For this purpose, we introduce the auxiliary function $ep_{X,h}$, which we refer to as the *entry point function*. Given a node x, $ep_{X,h}(x)$ denotes the first node in X that is reachable along the path starting from x if such a node exists, and otherwise it denotes x itself. This is formalized using the axioms in Fig. 9. The last two constraints in $FR(X, X', A, A', h, h')$ now specify that the order of nodes is preserved for the path segments between any node x and its entry point into X, respectively, the full path if no node in X is reachable from x.

The axioms defining the entry point function are conjoined with the final verification condition. We then partially instantiate the quantified variables as described in Section 8, yielding a formula in the EPR fragment that can be decided by the SMT solver. The quantified constraints encoding the frame and the entry point function satisfy locality conditions that ensure completeness of the instantiation.