BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

A DECISION PROCEDURE FOR STRONG-SEPARATION LOGIC

ROZHODOVACÍ PROCEDURA PRO SILNĚ-SEPARAČNÍ LOGIKU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. TOMÁŠ DACÍK**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

Master's Thesis Specification

Student: **Dacík Tomáš, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Mathematical Methods
Title: **A Decision Procedure for Strong-Separation Logic**
Category: Formal Verification
Assignment:

1. Study separation logic (SL), strong-separation logic (SSL), and possibilities of deciding formulae of SL and SSL.
2. Propose a decision procedure for SSL having at least some potential advantages compared with the existing decision procedures (e.g., in terms of their generality, ease of implementation, and/or scalability).
3. Describe the proposed decision procedure and show its correctness.
4. Implement the proposed decision procedure in a prototype tool and experimentally evaluate it.
5. Summarise the obtained results and discuss their possible future improvements.

Recommended literature:

- Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Proc. of LICS'02, IEEE CS, 2002.
- O'Hearn, P.W.: Separation Logic. Communications of the ACM, 62(2), ACM, 2019.
- Katelaan, J., Jovanovic, D., Weissenbacher, G.: A Separation Logic with Data: Small Models and Automation. In: Proc. of IJCAR'18, LNAI 10900, Springer, 2018.
- Pagel, J., Zuleger, F.: Strong-Separation Logic. In: Proc. of ESOP'21, LNCS 12648, Springer 2021.
- Pagel, J.: Decision Procedures for Separation Logic: Beyond Symbolic Heaps. Ph.D. thesis, Vienna University of Technology, 2020.

Requirements for the semestral defence:

- Point 1 and Point 2 at least for some suitable logical fragment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Vojnar Tomáš, prof. Ing., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: July 29, 2022
Approval date: November 3, 2021

Abstract

Separation logic (SL) is one of the most successful tools for verification of programs that manipulate dynamically allocated memory. Its expressive power, however, comes at a cost of undecidability when several of its features are combined, especially separating implications. To circumvent this problem, the recently introduced strong-separation logic (SSL) uses a stricter definition of the semantics, making it decidable, while remaining suitable for verification. However, there is currently no implementation of a decision procedure for SSL. In this work, we propose a decision procedure for SSL based on a translation to first-order formulae that can be later solved by a specialised solver. Our experimental results on restricted fragments where SL and SSL coincide show that our approach can effectively solve formulae obtained from verification tools based on SL and also outperform all other existing translation-based decision procedures. Moreover, during our experiments, we found cases of unsoundness of the heuristics implemented in the decision procedure for SL that is a part of the well-known CVC5 SMT solver. Based on our reports, those heuristics has been fixed.

Abstrakt

Separační logika (SL) patří mezi nejúspěšnější nástroje pro verifikaci programů pracujících s dynamicky alokovanou pamětí. Její vysoká expresivita ovšem přináší nerozhodnutelnost pokud formule kombinují více jejích spojek, především separační implikace. Jako řešení byla navrhnuta takzvaná silně-separační logika (SSL), která díky striktnější definici sémantiky rozšiřuje rozhodnutelný fragment a přitom zůstává vhodná pro verifikaci programů. V současnosti ale neexistuje žádná implementace rozhodovací procedury pro tuto logiku. Tato práce se zaměřuje na návrh a implementaci rozhodovací procedury pro SSL založené na překladu vstupní formule na formuli v prvořádové logice, jejíž splnitelnost je poté možné ověřit pomocí specializovaných nástrojů. Experimentální výsledky na omezeném fragmentu, kde SL a SSL splývají, ukazují, že navržený nástroj je schopen efektivně řešit formule pocházející z verifikačních nástrojů a výrazně překonává všechny ostatní existující rozhodovací procedury, které jsou také založené na překladu. Během experimentů jsme také odhalili několik případů nekorektnosti heuristik použitých v rozhodovací proceduře pro SL implementované v nástroji CVC5. Na základě našich hlášení byly tyto heuristiky opraveny.

Keywords

Separation logic, strong-separation logic, decision procedure, SMT

Klíčová slova

Separační logika, silně-separační logika, rozhodovací procedura, SMT

Reference

DACÍK, Tomáš. *A Decision Procedure for Strong-Separation Logic*. Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

Rozšířený abstrakt

Logika se v posledních letech stala velmi užitečným nástrojem v mnoha oblastech informatiky, především v oblasti automatizované verifikace softwaru a hardwaru. Formule v různých logikách lze použít nejen jako formální jazyk pro specifikaci korektního chování analyzovaného systému, ale také jako pomocnou technologii v programech, které korektnost ověřují – například pro reprezentaci nekonečných množin konfigurací programu nebo pro redukci výpočetně těžkých problémů, které se při verifikaci objevují, na problémy v logice.

Typickým problémem v logice je *splnitelnost* formule, která se ptá, zda pro danou formuli φ existuje objekt (zvaný model), který ji splňuje. V posledních letech bylo věnováno značné úsilí do vývoje nástrojů pro ověřování splnitelnosti ve výrokové logice (takzvané SAT solvery) a v teoriích prvořádové logiky (implementované v takzvaných SMT solverech). Přestože oba problémy jsou NP-těžké a jejich obecné efektivní řešení je tedy považováno za nedosažitelné, moderní nástroje dokáží efektivně řešit velké množství formulí pocházejících z praktických aplikací. Tyto aplikace zahrnují například ověřování verifikačních podmínek vygenerovaných při deduktivní verifikaci nebo automatické generování testovacích vstupů pro reálné programy.

Mimo klasické logiky existují další logiky specializované pro usuzování o různých aspektech počítačových programů. Příkladem je *separační logika* (SL) [31], která je hlavním předmětem této práce. Separační logika poskytuje obecný rámec pro modulární usuzování o sdílených zdrojích a jejich disjunktnosti. V nejčastějším případě je tímto sdíleným zdrojem dynamicky alokovaná paměť. Modulární usuzování je zajištěno novou logickou spojkou zvanou *separační konjunkce* – formule $\psi_1 * \psi_2$ vyjadřuje, že paměťovou haldu lze rozdělit na dvě části tak, že první splňuje ψ_1 a druhá ψ_2. Další novou spojkou je *separační implikace* (často nazývaná pro svůj vzhled *magic wand* – kouzelná hůlka). Formule $\varphi \mathrel{-\!\!*} \psi$ je splněna haldou, pro kterou platí, že pokud je rozšířena o model formule φ, výsledná halda splňuje ψ. Další ingrediencí separační logiky jsou *induktivní predikáty*, které popisují datové struktury neomezené délky, jako jsou seznamy nebo stromy, jejich varianty (např. dvousměrně vázané seznamy) a kombinace (např. stromy se zřetězenými listy). Typickým příkladem je predikát $\mathsf{ls}(x, y)$ reprezentující acyklický jednosměrně vázaný seznam. Konkrétním případem formule je $\mathsf{ls}(x, y) * y \mapsto x$ vyjadřující, že haldu lze rozdělit na acyklický seznam z lokace x do lokace y, a ukazatel z lokace y do lokace x – formule tedy vyjadřuje cyklický seznam.

Vysoká expresivita separační logiky sebou ovšem přináší vysokou složitost, v případě některých fragmentů dokonce nerozhodnutelnost. S. Demri nedávno ukázal, že kombinace všech výše zmíněných ingrediencí (induktivních predikátů, separační konjunkce a separační implikace) a booleovských spojek je nerozhodnutelná [12]. Řada verifikačních nástrojů tak pracuje s jednoduššími fragmenty logiky, které typicky neobsahují separační implikaci. Separační implikace se ovšem přirozeně objevuje například ve verifikačních podmínkách generovaných symbolickou exekucí [1] nebo v tzv. bi-abduktivní analýze [10].

Motivováni výše zmíněnou nerozhodnutelností, J. Pagel a F. Zuleger nedávno představili tzv. silně-separační sémantiku, při které se výše zmíněný fragment stává rozhodnutelným v polynomiálním prostoru [25]. Vzniklá *silně-separační logika* (SSL) koresponduje s klasickou separační logikou na tzv. *pozitivním fragmentu* neobsahujícím negaci a separační implikaci, a lze se na ni tedy dívat jako na „zpětně kompatibilní“ rozšíření klasické SL. V práci [25] je představen koncept *abstraktních paměťových stavů* (konečné abstrakce nad potenciálně nekonečnými množinami modelů) a navržena rozhodovací procedura založená na jejich enumeraci. Tato procedura ovšem slouží především pro důkaz rozhodnutelnosti a nebyla nikdy implementována.

Cílem této práce je navrhnout a implementovat rozhodovací proceduru pro SSL. Nově navržená rozhodovací procedura pracuje na jiném principu – převádí vstupní formuli v separační logice na ekvisplnitelnou formuli v prvořádové logice. Motivací tohoto přístupu je snaha efektivně využít moderních nástrojů pro řešení SMT problému. Několik podobných překladů již bylo navrženo pro klasickou separační logiku, tato práce ovšem výrazně rozšiřuje fragment, který lze přeložit, o omezené použití separační implikace a libovolnou kombinaci booleovských a prostorových spojek. Navíc je v práci navrženo několik metod snižujících velikost přeložené formule, například díky výpočtům dolních a horních omezení na délky seznamů.

Navržená rozhodovací procedura je implementována v novém nástroji Astral a díky korespondenci klasické SL a SSL umožňuje řešit i řadu formulí v klasické separační logice. Mimo jiné například formule obsahující seznamy a libovolně kombinované disjunkce a separační konjunkce, což je podle autorů [5] fragment, který není žádnými dalšími nástroji podporován.

Experimenty na fragmentu, kde SL a SSL splývají, ukazují, že Astral je schopen efektivně řešit formule pocházející z verifikační nástrojů a překonat ostatní existující rozhodovací procedury založené na překladu do SMT. Během experimentálního srovnání s rozhodovací procedurou pro fragment se separační implikací, ale bez induktivnách predikátů, implementovanou v nástroji cvc5, jsme také odhalili chybně vyřešené formule obsahující separační implikace. Ukázalo se, že se jedná o důsledek několika nekorektních heuristik a tyto heuristiky byly posléze na základě našich hlášení opraveny.

A Decision Procedure for Strong-Separation Logic

Declaration

Hereby I declare that this master thesis was prepared as an original author's work under the supervision of prof. Ing. Tomáš Vojnar, Ph.D. The supplementary information was provided by doc. Mgr. Adam Rogalewicz, Ph.D. and Associate Prof. Dipl.-Math. Dr.techn. Florian Zuleger. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Tomáš Dacík
July 29, 2022

Acknowledgements

I would like to thank my supervisor Tomáš Vojnar for numerous pieces of advice to this thesis and for a great opportunity to work on such an interesting research topic. I also wish to express my thanks to Florian Zuleger and Adam Rogalewicz for consultations, and to all members of the VeriFIT research group for an inspiring working environment. Furthermore, I would like to thank my family for their support during my studies.

I acknowledge the support received from the project Snappy of the Czech Science Foundation.

Contents

Chapter 1

Introduction

In recent years, logic proved to be a very useful tool in many fields of computer science, including in the area of automated software and hardware verification. Formulae in various logics can be used not only as formal languages for specification of the correct behaviour of the analysed system, but they can also serve as a backend technology in tools that attempt to verify the specification – e.g., to succinctly represent infinite sets of program configurations, or to reduce computationally hard problems that appear during verification to problems in logics, which can be solved by specialised solvers.

One of the most common problems in logic is satisfiability of a formula φ which asks whether there exists an object (called model) that satisfies φ. In recent years, a significant research effort has been invested into development of satisfiability solvers for propositional logic (so-called SAT solvers) and various theories in first-order logic (implemented in so-called SMT solvers). While both problems are NP-hard (and some SMT problems even harder) and therefore considered as intractable in general, existing solvers can effectively handle large classes of formulae originating from practical applications. Those applications are, e.g., discharging preconditions generated by deductive verification tools, checking entailment or emptiness in abstract interpretation based on logic, automatic generation of test cases for real-life programs, and many others.

Besides the classical logics, there are also logics developed to reason about specific aspects of computer programs, such as *separation logic* (SL) [31], which is the main subject of this thesis. It is a logical framework for *modular* reasoning about shared resources and their disjointness. In the most common setting, the shared resource is a heap-allocated memory. The modular reasoning is due to a new connective called the *separating conjunction* – a formula $\psi_1 * \psi_2$ states that a heap can be split into two disjoint parts such that the formula ψ_1 is satisfied in the first part and ψ_2 is satisfied in the second. Another new connective is the *separating implication* (often called as the "magic wand"). A formula $\varphi \mathrel{-\!\!*} \psi$ is satisfied by a heap such that for each its extension satisfying φ, their composition satisfies ψ. The last ingredient are inductive predicates describing data structures of unbounded size such as lists or trees. For example, the predicate $\mathsf{ls}(x, y)$ is used to express an acyclic singly-linked list, i.e., a sequence of pointers from x to y. A concrete example of an SL formula is $\mathsf{ls}(x, y) \ * \ y \mapsto x$ which states that a heap can be decomposed into an acyclic list from x to y, and a pointer from y to x. In other words, it expresses a cyclic list.

However, the high expressive power of separation logic comes with the price of high complexity and even undecidability when several of the aforementioned features are combined together. In particular, as recently shown by Demri [12], a quantifier-free fragment of SL combining separating conjunctions, magic wands, and list-segment predicates is unde-

cidable under the classical semantics. Most verification tools therefore sacrifice the magic wand. Magic wands do, however, naturally appear in verification conditions generated by symbolic execution [1] and in the so-called *bi-abductive analysis* [10].

To tackle the undecidability and allow verification tools to automate magic wands, Pagel and Zuleger proposed a so-called *strong-separating* semantics under which the mentioned fragment becomes decidable in PSPACE [25]. The resulting *strong-separation logic* (SSL) coincides with the classical SL on the so-called *positive fragment* that does not contain negations and magic wands. SSL therefore can be seen as a backward compatible extension of the classical SL. In [25], they propose a concept of *abstract memory states* (AMS is a finite abstraction over possibly infinite sets of models) and a decision procedure based on their enumeration. However, the algorithm serves as a proof of decidability and was never implemented.

This thesis presents a first implementation of a decision procedure for a fragment of SSL. Rather than performing a custom enumeration of AMSs, we perform a translation to an equisatisfiable first-order formula to leverage capabilities of existing SMT solvers. Such translations already exist for classical SL, but we significantly extend the fragment being translated. The extensions cover limited usage of magic wands and arbitrary mixing of boolean and spatial connectives. We also propose several new heuristics to decrease the size of translated formulae, e.g., by computing bounds on lengths of list-segment predicates.

The proposed decision procedure was implemented in a new solver called Astral. Due to coincidence of the classical SL and SSL, Astral can be also used to solve a wide class of SL formulae. Those include, e.g., formulae with list-segment predicates which are mixing disjunctions and separating conjunctions that are according to authors of [5] currently not supported by any existing tool.

Experimental results on simpler fragments show that our approach can effectively solve formulae obtained from verification tools based on SL and also outperform other existing translation-based decision procedure implemented in tools Sloth [17] and Grasshopper [28]. We have also compared our tool with the cvc5 SMT solver which implements a decision procedure for SL with magic wands but without inductive predicates. During those experiments, we found and reported several incorrect results for formulae containing magic wands. Those turned to be results of unsound heuristics and were later fixed based on our reports.

Structure of the thesis. The rest of the thesis is structured as follows. Chapter 2 introduces a notation used throughout the thesis and give an overview of the classical separation logic and existing decision procedures. Strong separation logic is then presented in Chapter 3. Chapter 4 proposes a new translation-based decision procedure for a fragment of SSL and proves its correctness. In Chapter 5, we propose several optimisations of the translation and in Chapter 6 we discuss its implementation in the tool called Astral. Chapter 7 is devoted to an experimental evaluation. Finally, Chapter 8 concludes the thesis and suggests several directions of the future research.

Chapter 2

Preliminaries

This chapter presents the theoretical background of the thesis. First, we introduce basic mathematical notation used throughout the thesis. Further, we briefly recall syntax and semantics of first-order logic and the problem of satisfiability modulo theory. Then we introduce separation logic and give an overview of existing decision procedures for it.

2.1 Mathematical Notation

Partial functions. We write $f : X \rightharpoonup Y$ to denote a *partial function* from X to Y. Let f be a partial function, we use $f(x) = \bot$ to denote the fact that f is undefined for x, and we write $\mathsf{dom}(f)$ and $\mathsf{img}(f)$ to denote the domain and the image of f, respectively. The function is *total* if $\mathsf{dom}(f) = X$. A *restriction* of f to a set $A \subseteq X$ is a partial function $f|_A$ defined as $f(x)$ if $x \in A$ and undefined otherwise. The size of a function f is defined as the size of its domain, i.e., $|f| = |\mathsf{dom}(f)|$.

We sometimes use a set notation to define partial functions. For example, the set $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ represents a partial function that maps each x_i to y_i and is undefined for other values.

Graphs and paths. Let $G = (V, \rightarrow)$ be a directed graph. A *path* $\pi \in V^+$ is a sequence of vertices $\langle v_0, v_1, \ldots, v_n \rangle$ such that for all $0 \leq i < n$ it holds that $v_i \rightarrow v_{i+1}$. The *domain* of the path π is the set $\mathsf{dom}(\pi) = \{v_0, v_1, ..., v_{n-1}\}$ and the length of the path is defined as $|\pi| = |\mathsf{dom}(\pi)| = n$. In particular, for every vertex $v \in V$ there is the *empty path* $\pi = \langle v \rangle$ with $\mathsf{dom}(\pi) = \emptyset$ and $|\pi| = 0$. A path is *simple* if it does not contain any vertex more than once. All simple paths are therefore acyclic. We write $x \overset{\pi}{\rightsquigarrow} y$ to denote the fact that π is a simple path from x to y.

Formulae. We use several notations related to formulae, no matter whether they are from separation or first-order logic. Let φ be a formula. We write $\varphi[t/x]$ to denote the formula obtained from φ by simultaneously replacing all free occurrences of the variable x with the term t. We write $\mathsf{vars}(\varphi)$ to denote the set of all free variables in φ and call φ *closed* if $\mathsf{vars}(\varphi) = \emptyset$. Further, we write $\mathsf{subformulae}(\varphi)$ to denote all sub-formulae of φ. Moreover, we use the predicate $\mathsf{distinct}(x_1, \ldots, x_n)$ to denote that all variables x_i are pairwise different, i.e., as syntactic sugar for $\bigwedge_{i \neq j} x_i \neq x_j$.

2.2 First-Order Logic and Satisfiability Modulo Theory

This section briefly recalls the syntax and the semantics of *single-sorted first-order logic with equality* (FOL) and the problem of *satisfiability modulo theory* (SMT). The section is based on [8].

2.2.1 Syntax and Semantics

Syntax. A *signature* Σ is a set of function and predicate symbols with associated arities. We assume that each signature contains the binary equality symbol $=$. A function symbol with arity 0 is called a *constant*. Let $\mathcal{X}$ be a set of variables disjoint from Σ. A Σ-*term* t is either a variable or an application of an n-ary function symbol f to an n-tuple of terms. A Σ-*atom* (atomic formula) is either a boolean constant ($\top, \bot$), an equality of two terms, or an application of an n-ary predicate p to an n-tuple of terms. A Σ-*formula* is constructed from atomic formulae using classical boolean connectives ($\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$) and quantifiers ($\forall$, $\exists$).

Semantics. Let Σ be a signature. A Σ-*interpretation* $\mathcal{M}$ is a pair $(\mathcal{D}, (\cdot)^{\mathcal{M}})$ where $\mathcal{D}$ is a non-empty set called the *domain* of $\mathcal{M}$ and $(\cdot)^{\mathcal{M}}$ is a total function called the *assignment* that maps each n-ary function symbol f to an n-ary total function $f^{\mathcal{M}} : \mathcal{D}^n \rightarrow \mathcal{D}$, each n-ary predicate symbol p to an n-ary predicate $p^{\mathcal{M}} \subseteq \mathcal{D}^n$, and also each variable $x \in \mathcal{X}$ to an element $x^{\mathcal{M}} \in \mathcal{D}$. The symbol $=$ is always interpreted as the equality on $\mathcal{D}$.

The *evaluation* of a term t in an interpretation $\mathcal{M}$ is denoted as $t^{\mathcal{M}}$ and is defined inductively over the structure of the term t in the usual way. Similarly, the evaluation of a formula φ in an interpretation $\mathcal{M}$ is defined. We say that a formula φ is *satisfied* in an interpretation $\mathcal{M}$ (or equivalently that $\mathcal{M}$ is a *model* of φ), denoted as $\mathcal{M} \models \varphi$, if φ evaluates to true in $\mathcal{M}$.

Satisfiability and validity. A Σ-formula φ is *satisfiable* if there is a Σ-interpretation $\mathcal{M}$ such that $\mathcal{M} \models \varphi$, φ is called *valid* if for all Σ-interpretations $\mathcal{M}$ it holds that $\mathcal{M} \models \varphi$. Satisfiability and validity are dual, a closed formula φ is valid iff $\neg\varphi$ is unsatisfiable.

2.2.2 Satisfiability Modulo Theory

Theories and the SMT problem. A Σ-*theory* $\mathcal{T}$ is a set of closed Σ-formulae called *axioms*. A Σ-interpretation $\mathcal{M}$ is called a $\mathcal{T}$-*interpretation* if $\mathcal{M} \models \mathcal{A}$ for all axioms $\mathcal{A} \in \mathcal{T}$. A theory $\mathcal{T}$ is *consistent* if there exists a $\mathcal{T}$-interpretation. A formula φ is called $\mathcal{T}$-*satisfiable*, if there exists a $\mathcal{T}$-interpretation $\mathcal{M}$ in which φ is satisfied, denoted as $\mathcal{M} \models_{\mathcal{T}} \varphi$. The problem of *satisfiability modulo theory* (SMT) asks to determine whether φ is $\mathcal{T}$-satisfiable or not, given a fixed theory $\mathcal{T}$.

SMT solvers. Commonly used theories are, e.g, linear integer arithmetic (LIA), real arithmetic or the theory of fixed-size bit vectors. Algorithms for deciding those theories are implemented in so-called SMT solvers. Usually, they implement a dedicated sub-solver for each theory. For some theories, those sub-solvers may be modularly combined using, e.g., the Nelson-Oppen combination method. Prominent examples of SMT solvers are Z3 [21] and CVC5 [3].

Definitions of common theories as well as an input language of SMT solvers are standardised in the SMT-LIB format [4]. The input format is formalised in *many-sorted* FOL in which domains of interpretations are split into multiple sub-domains called *sorts* (they

roughly correspond to basic types in programming languages). In this thesis, we, for simplicity, present our translation of separation logic in single-sorted FOL. Its actual implementation in many-sorted setting is, however, a very straightforward modification.

2.2.3 Generalised Theory of Arrays

As an example of a first-order theory, we will describe the *generalised theory of arrays* [20] that we will also use as the "target language" of our translation of separation logic.

The basic theory of arrays $\mathcal{T}_A$ has the signature $\Sigma_A = \{\cdot[\cdot], \cdot\langle\cdot \triangleleft \cdot\rangle\}$ where a term $a[i]$ represents a *read* from the array a at the position i and a term $a\langle i \triangleleft v\rangle$ represents a modification of the array a by *writing* the value v at the position i. This intuitive behaviour of *reading* and *writing* to an array is captured by the following axioms.

- $\forall a, i, j.\ i = j \ \rightarrow\ a[i] = a[j]$ (array congruence)
- $\forall a, v, i, j.\ i = j \ \rightarrow\ a\langle i \triangleleft v\rangle[j] = v$ (read-over-write 1)
- $\forall a, v, i, j.\ i \neq j \ \rightarrow\ a\langle i \triangleleft v\rangle[j] = a[j]$ (read-over-write 2)

The theory of arrays is undecidable, but its quantifier-free fragment is decidable in NP.

The generalised theory of arrays $\mathcal{T}_A^+$ [20] adds combinators which allow one to express certain universal properties without relying on quantifiers. A combinator $\mathsf{K}(x)$ represents a constant array whose all elements are x. For an n-ary function f, a combinator $\mathsf{map}_f(a_1, \ldots, a_n)$ represents an array obtained by applying the function f point-wise to arrays $a_1, \ldots, a_n$. It can therefore express operations such as point-wise addition of two integer arrays. Those combinators are axiomatised by the following axioms (the second is, in fact, an axiom scheme).

- $\forall x, i.\ \mathsf{K}(x)[i] = x$
- $\forall a_1, \ldots, a_n, i.\ \mathsf{map}_f(a_1, \ldots, a_n)[i] = f(a_1[i], \ldots, a_n[i])$ for each n-ary function f

As for the basic theory of arrays, the generalised version is decidable in NP. A decision procedure for $\mathcal{T}_A^+$ is implemented in the SMT solver Z3 [20].

Encoding finite sets as arrays. The generalised theory of arrays can be used to encode basic operations over finite sets. This will be useful when translating separation logic to express properties such as the requirement that the domains of two heaps are disjoint. Given a finite universe U, a set $X \subseteq U$ can be encoded as an array representing its characteristic function, i.e., mapping each element $x \in U$ to a boolean value representing its membership in X. In this encoding, a constant set can be represented as:

$$\{x_1, x_2, ..., x_n\} \ \triangleq\ \mathsf{K}(\bot)\langle x_1 \triangleleft \top\rangle\langle x_2 \triangleleft \top\rangle \ ... \ \langle x_n \triangleleft \top\rangle$$

Basic set operations and predicates can be expressed as follows.

$$\begin{aligned} \overline{X} &\triangleq \mathsf{map}_\neg(X) & X = \emptyset &\triangleq X = \mathsf{K}(\bot) \\ X \cup Y &\triangleq \mathsf{map}_\vee(X, Y) & x \in X &\triangleq X[x] \\ X \cap Y &\triangleq \mathsf{map}_\wedge(X, Y) & X \subseteq Y &\triangleq \mathsf{map}_\rightarrow(X, Y) = \mathsf{K}(\top) \end{aligned}$$

The theory of finite sets with cardinality constraints is also supported natively by the CVC5 SMT solver [2], but it is not standardised in the SMT-LIB standard.

2.3 Separation Logic

Separation logic (SL) was developed to reason about imperative programs manipulating dynamically allocated memory [31], including the so-called shape analysis capturing the shapes of memory-allocated structures, and it quickly becomes probably the most successful approach in this area. Meanwhile, many various flavours of SL were introduced [26], some of them for reasoning about shared resources other than the memory such as concurrency [23], but heap-manipulating programs are still the most common domain.

This section presents an introduction into the classical semantics of separation logic and discusses its existing decision procedures. A flavour of SL called strong-separation logic which is studied in this thesis is introduced later in Section 3.

2.3.1 Syntax

Let **Var** be an infinite set of variables with a distinguished variable $\mathsf{nil} \in \mathbf{Var}$. The syntax of first-order separation logic is given by the following grammar where $x, y \in \mathbf{Var}$:

$$
\begin{aligned}
\varphi_{atom} &::= x = y \mid x \neq y && \text{(pure atoms)}\\
&\mid \mathsf{emp} \mid x \mapsto y && \text{(spatial atoms)}\\
\varphi &::= \varphi_{atom}\\
&\mid \varphi * \varphi \mid \varphi \mathrel{-\!\!*} \varphi && \text{(spatial connectives)}\\
&\mid \varphi \wedge \varphi \mid \neg\varphi && \text{(boolean connectives)}\\
&\mid \exists x.\ \varphi && \text{(quantifiers)}
\end{aligned}
$$

A pure atomic formula is either an equality $x = y$ or a disequality $x \neq y$. A spatial atomic formula is either the empty heap predicate emp, which intuitively expresses that the heap does not contain any pointers, or a points-to assertion $x \mapsto y$ intuitively expressing that a heap consists of exactly one pointer from the location x to the location y[1]. The formulae are obtained using quantifiers, boolean connectives and spatial connectives $*$ (*separating conjunction*) and $\mathrel{-\!\!*}$ (*separating implication* also called the *magic wand*). Intuitively, a formula $\psi_1 * \psi_2$ states that a heap can be split into two (disjoint) parts such that ψ_1 is satisfied in the first of them and ψ_2 is satisfied in the second. Similarly, a formula $\varphi \mathrel{-\!\!*} \psi$ intuitively states that each (disjoint) extension of a heap by another heap satisfying φ yields a heap satisfying ψ. Concrete flavours of SL may differ in the way how disjointness of two heaps is defined.

For a set of formulae $\Phi = \{\varphi_1, \varphi_2, ..., \varphi_n\}$, we define an n-ary version of the separating conjunction:

$$
*\,\Phi = \begin{cases} \mathsf{emp} & \text{if n} = 0 \\ \varphi_1 * \varphi_2 * ... * \varphi_n & \text{if } n > 0 \end{cases}
$$

A frequently used fragment of SL is the so-called *symbolic heap fragment*. A formula φ is a symbolic heap if it is of the form $\Pi \wedge \Sigma$ where $\Pi \triangleq \bigwedge \psi_i$ is a conjunction of pure atoms called the *pure part* and $\Sigma \triangleq *\,\psi_i$ is a separating conjunction of spatial atoms called the *spatial part*. Although the fragment is significantly restricted, it is still expressive enough to be useful for program verification, e.g., for symbolic execution in the Smallfoot analyser [7] and many other similar analysers.

[1] In a more general setting, points-to assertions can be of the form $x \mapsto \langle y_1, \ldots, y_n \rangle$ intuitively expressing that a heap consists of a pointer from x to an object consisting of fields $y_1, \ldots, y_n$.

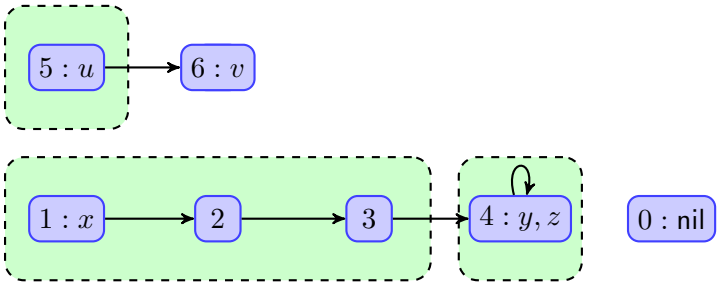


Figure 2.1: An example of a graph representation of a stack-heap model (s, h). It holds that $(s, h) \models \mathsf{ls}(x, y) * \mathsf{ls}(u, v) * z \mapsto y$. The corresponding decomposition of the heap h is depicted using green boxes.

2.3.2 Memory Model

We will interpret SL over *stack-heap models*. Let **Loc** be a countably infinite set of memory locations with some fixed linear order. A stack-heap model is a pair (s, h) where *stack* is a finite partial function $s : \mathbf{Var} \rightharpoonup \mathbf{Loc}$ such that $s(\mathsf{nil}) \neq \bot$, and *heap* is a finite partial function $h : \mathbf{Loc} \rightharpoonup \mathbf{Loc}$ such that $h(s(\mathsf{nil})) = \bot$. For a heap h, we define the set of its locations as $\mathsf{locs}(h) = \mathsf{dom}(h) \cup \mathsf{img}(h)$.

As demonstrated in Figure 2.1, a stack-heap model (s, h) can be represented as a directed graph where vertices are heap locations and edges represent heap pointers. To capture also the stack, each vertex is labelled by variables that are mapped to it. This correspondence is formalised by the following definition of an *induced graph* of a model.

Definition 2.1 (Induced graph)**.** *Let (s, h) be a stack-heap model. Its induced graph $G[(s, h)] = (V, \rightarrow, s^{-1})$ is defined as follows:*

- $V = \mathsf{locs}(h) \cup \mathsf{img}(s)$
- $u \rightarrow v \;\Leftrightarrow\; h(u) = v$
- $s^{-1}(v) = \{x \in \mathbf{Var} \mid s(x) = v\}$

In the rest of this thesis, we identify the model and its graph representation. While, in the definition, we strictly require that each stack-heap model contains the nil location, we omit it in examples where it is not relevant.

We introduce several notations related to stack-heap models. Let (s, h) be a model and let ℓ be a location. We say that variables x and y *alias* if $s(x) = s(y)$. We call ℓ *anonymous* if $s^{-1}(\ell) = \emptyset$ (it is not referred from the stack) and *named* otherwise. We say that the heap h contains a pointer from x to y if $h(x) = y$. We call ℓ *allocated* if $\ell \in dom(h)$ (it has some successor) and dangling if it holds that $\ell \in \mathsf{img}(h) \setminus \mathsf{dom}(h)$ (the predecessor of ℓ is allocated, but ℓ itself is not). A pointer $x \mapsto y$ is dangling if its target location y is dangling.

Example 2.1. Let us consider the stack-heap model (s, h) from Figure 2.1. Throughout this thesis, we will usually consider locations to be natural numbers, i.e., $\mathbf{Loc} := \mathbb{N}$. In the model, the variables y and z alias. Locations 2 and 3 are the only anonymous locations here, and locations 0 and 6 are the only locations that are not allocated. The only dangling location is the location 6 because it is in the image of h, but not in its domain. The pointer $5 \mapsto 6$ is therefore dangling. The location 0 is not part of $\mathsf{locs}(h)$ but it is included among vertices of $G[(s, h)]$.

$$
\begin{array}{ll}
(s,h) \models x = y & \text{iff } s(x) = s(y) \\
(s,h) \models x \neq y & \text{iff } s(x) \neq s(y) \\
(s,h) \models \mathsf{emp} & \text{iff } h = \emptyset \\
(s,h) \models x \mapsto y & \text{iff } h = \{s(x) \mapsto s(y)\} \\
\\
(s,h) \models \varphi_1 \wedge \varphi_2 & \text{iff } (s,h) \models \varphi_1 \text{ and } (s,h) \models \varphi_2 \\
(s,h) \models \neg\varphi & \text{iff } (s,h) \not\models \varphi \\
(s,h) \models \exists x.\, \varphi & \text{iff there exists } \ell \in \mathbf{Loc} \text{ such that } (s \cup \{x \mapsto \ell\}, h) \models \varphi \\
\\
(s,h) \models \varphi_1 * \varphi_2 & \text{iff } \exists h_1, h_2.\ (s,h_1) \models \varphi_1, (s,h_2) \models \varphi_2, h_1 + h_2 \neq \bot \text{ and } h = h_1 + h_2 \\
(s,h) \models \varphi \mathrel{-\!\!*} \psi & \text{iff } \forall h_1.\ \text{if } (s,h_1) \models \varphi \text{ and } h + h_1 \neq \bot, \text{ then } (s, h + h_1) \models \psi
\end{array}
$$

Figure 2.2: The classical semantics of separation logic.

2.3.3 Semantics

The semantics of separation logic over stack-heap models is given in Figure 2.2. An equality $x = y$ is satisfied by a stack-heap model interpreting both variables in the same way. The semantics of disequality is analogical. A points-to assertion $x \mapsto y$ is satisfied in a heap consisting of a single pointer which, moreover leads from x to y. The semantics of boolean connectives and the existential quantifier is defined in the usual way. The semantics of spatial connectives is based on a notion of *disjointness* of two heaps (the semantics of strong-separation logic defined later in Section 3 will differ in its definition of disjointness). In the classical SL, heaps h_1 and h_2 are disjoint if their domains are disjoint. A *disjoint union* of heaps is defined as follow:

$$
h_1 + h_2 = \begin{cases} h_1 \cup h_2 & \text{if } \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset \\ \bot & \text{otherwise} \end{cases}
$$

We now give several examples of separation logic formulae to show differences in the semantics of the classical and separation conjunction, and also to provide some intuition behind the magic wand.

Example 2.2. Let $\varphi_1 \triangleq x \mapsto y * x \mapsto z$. The formula φ_1 is unsatisfiable because it requires the location x to be allocated in both sub-heaps, which is forbidden by the semantics of the separating conjunction. On the other hand, the formula $\varphi_2 \triangleq x \mapsto y * z \mapsto y$ is satisfiable. Notice that φ_2 implicitly asserts that the variables x and z represent different locations.

Example 2.3. The heap $h = \{s(x) \mapsto s(y), s(y) \mapsto s(\mathsf{nil})\}$ does not satisfy the formula $\varphi_3 \triangleq x \mapsto y$ (no matter what the stack is). This is because a points-to assertion expresses the fact that "a heap consist of a pointer", rather than "a heap contains a pointer". Of course, the so-called *intuitionistic* points-to assertion $\varphi_4 \triangleq x \mapsto y * \mathsf{true}$ can be to used to express that a heap contains the pointer.

Example 2.4. A formula $\varphi_5 \triangleq x \mapsto y \wedge y \mapsto z$ states that a heap consists a pointer from x to y and from y to z, simultaneously. The formula is therefore satisfiable only when those pointers are unified, i.e., it can be satisfied by the only stack-heap model (s, h) such that $s(x) = s(y) = s(z)$ and $h = \{s(x) \mapsto s(x)\}$.

Example 2.5. Let $\varphi_6 \triangleq (x \mapsto \mathsf{nil}) \mathrel{-\!\!*} \mathsf{false}$. The formula is satisfied in a model (s, h) if for all its extensions satisfying $x \mapsto \mathsf{nil}$ (there is zero or one such an extension depending on whether h already allocates x or not), it holds that their composition satisfies false. Since no model satisfies false, this means that h has to allocate x to ensure that it has no disjoint extension satisfying $x \mapsto \mathsf{nil}$. The formula therefore states "location x is allocated". This can be also expressed using quantifiers $\exists \ell.\ x \mapsto \ell$. If neither the magic wands nor quantifiers are supported, the property cannot be expressed.

Inductive Predicates Separation logic also allows one to specify inductive predicates to describe data structures of unbounded size (such as lists or trees), their variants (such as doubly linked lists) and combinations (such as nested lists or trees with linked leaves). In concrete flavours of SL, those predicates can be either built in the logic, or, in a more general setting, the logic may allow to define custom inductive predicates.

Inductive predicates can be defined by a system of inductive definitions which consists of rules of the form $p(x_1, x_2, ..., x_n) ::= \varphi$. For example, a possibly empty, acyclic singly-linked list predicate $\mathsf{ls}(x, y)$ can be defined by the following system of definitions:

$$\mathsf{ls}(x, y) ::= x = y \ \wedge\ \mathsf{emp}$$
$$\mathsf{ls}(x, y) ::= \exists z.\ x \neq y \wedge \Big(x \mapsto z \ *\ \mathsf{ls}(z, y)\Big)$$

The definition says that a model (s, h) satisfies a predicate $\mathsf{ls}(x, y)$ either if the heap is empty and $s(x) = s(y)$, or there exists a location z such that there is a pointer from x to z and the rest of the heap is a list segment from z to y. The condition in the second definition that x and y are different forbids cyclic lists. Similarly, a tree with a root r can be defined by the following system:

$$\mathsf{tree}(r) ::= r = \mathsf{nil} \ \wedge\ \mathsf{emp}$$
$$\mathsf{tree}(r) ::= \exists l, r.\ x \mapsto \langle l, r \rangle \ *\ \mathsf{tree}(l) \ *\ \mathsf{tree}(r)$$

Example 2.6. The formula $(x \mapsto y) * (y \mapsto z) \wedge \neg\big(\mathsf{ls}(x, z)\big)$ is satisfiable. While this does not have to be obvious at the first sight, let us consider the stack heap model (s, h) with $s(x) = s(z)$ and $h = \{s(x) \mapsto s(y), s(y) \mapsto s(z)\}$. The formula is satisfied in this model because list-segments have to be acyclic.

2.3.4 Decision Procedures for Separation Logic

There exist many decision procedures for various fragments and flavours of separation logic. The first studied fragment were symbolic heaps with lists; in [6], a proof system for satisfiability and entailment was proposed. Both satisfiability and entailment for this fragment were later shown to be solvable in polynomial time [11]. A model-based approach for this fragment which is partially based on the Z3 solver was proposed in [22] and implemented in the tool called Asterix.

A translation of SL to SMT was first proposed in [28] and [29] for boolean combinations of symbolic heaps with lists and trees, respectively. Those approaches use intermediate logics that are later translated to SMT. Another translation, closer to our approach proposed in the following, was described in [17], which establishes a small-model property for separation logic with data predicates and performs a direct translation implemented in the tool Sloth. A similar translation was designed in [24] for SSL with data but not implemented. The work, however, considers only a fragment on which SL and SSL coincide.

All those translations consider only such fragments of SL where boolean connectives cannot appear under separating conjunction, and the magic wand cannot appear at all. A fragment with the magic wand, arbitrary combinations of boolean and spatial connectives, but no inductive predicates is supported by the SMT solver cvc5 that implements a specialised theory solver for this fragment [30]. The solver is based on a translation to second-order logic with quantifiers over bounded sets which is then solved by a lazy quantifier instantiation. As shown in [12], adding only the list-segment predicate to this fragment leads to undecidability.

A separation logic with quantifiers (restricted to the $\exists^*\forall^*$ quantifier-prefix) was studied in [13]. The majority of solvers, however, work within quantifier-free fragments. An example is Songbird which constructs induction proofs using lemma synthesis [33].

Inductive definitions. All methods mentioned so far assumed only inductive predicates that were built in the logic. A generalisation is to allow *user-defined* inductive predicates (usually of some restricted form) that can describe more complex data structures such as double-linked lists, cyclic lists, or trees and various combinations of the mentioned. Solvers proposed for those logics are based, e.g., on the *cyclic proof systems* (Cyclist [9]) or various kinds of automata – tree automata are used in tools Slide [15] and Spen [14], and a specialised type of automata, called *heap automata*, is used in Harrsh [19].

Chapter 3

Strong-Separation Logic

Strong-separation logic (SSL) was recently introduced to overcome undecidability results of separation logic with the classical semantics in the presence of magic wands, negations, and list-segment predicates. To emphasise the difference, we will further call separation logic with the classical semantics as *weak-separation logic* (WSL). This chapter formally introduces SSL based on [25] where one can also found all omitted proofs. We will first introduce its syntax and semantics and compare it with the semantics of WSL. Then, we will describe *abstract memory states* that can be used as a building block of a decision procedure for SSL, and also to prove several properties of SSL. Namely, we will prove that it has a small-model property, i.e., that each satisfiable formula has a model of a linear size. This property is essential for an effective translation of SSL to SMT.

3.1 Syntax

We will concentrate on a quantifier-free fragment of SL where the list segment is the only built-in inductive predicate[1]. The syntax of this fragment is given by the following grammar:

$$
\begin{aligned}
\varphi_{atom} &::= x = y \mid x \neq y && \text{(pure atoms)}\\
&\mid x \mapsto y \mid \mathsf{ls}(x, y) && \text{(spatial atoms)}\\
\varphi &::= \varphi_{atom}\\
&\mid \varphi * \varphi \mid \varphi \mathrel{-\!\!\circledast} \varphi && \text{(spatial connectives)}\\
&\mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \wedge_{\neg} \varphi \mid \neg\varphi && \text{(boolean connectives)}
\end{aligned}
$$

There are several differences from the syntax given in the introduction and non-standard choices. Instead of the magic wand, we use its existential variant called *septraction*. The reason is that its existential character is more natural when working with satisfiability. The syntax does also not contain the empty predicate emp as it can be expressed using other atoms.

An important subset of SSL is its so-called *positive fragment* denoted as SSL^{+}. A formula φ is *positive* if it does not contain a negation. In the positive fragment, however, a so-called *guarded negation* $\wedge_{\neg}$ can be used. A formula $\varphi \wedge_{\neg} \psi$ is semantically equivalent to the formula $\varphi \wedge \neg\psi$, but we rather treat the guarded negation as a standalone binary

[1] This is, however, not a limitation of the strong-separation semantics – an extension of SSL including trees can be found in [24].

$$
\begin{array}{ll}
(s,h) \models x = y & \text{iff } s(x) = s(y) \text{ and } \mathsf{dom}(h) = \emptyset \\
(s,h) \models x \neq y & \text{iff } s(x) \neq s(y) \text{ and } \mathsf{dom}(h) = \emptyset \\
(s,h) \models x \mapsto y & \text{iff } h = \{s(x) \mapsto s(y)\} \\
(s,h) \models \mathsf{ls}(x,y) & \text{iff } \mathsf{dom}(h) = \emptyset \text{ and } s(x) = s(y) \text{ or there exist } n \geq 1, \ell_0, ..., \ell_n \text{ such that} \\
 & \quad \mathsf{distinct}(\ell_0, ..., \ell_n), h = \{\ell_0 \mapsto \ell_1, ..., \ell_{n-1} \mapsto \ell_n\}, s(x) = \ell_0, \\
 & \quad \text{and } s(y) = \ell_n \\
(s,h) \models \varphi_1 \wedge \varphi_2 & \text{iff } (s,h) \models \varphi_1 \text{ and } (s,h) \models \varphi_2 \\
(s,h) \models \varphi_1 \wedge_\neg \varphi_2 & \text{iff } (s,h) \models \varphi_1 \text{ and } (s,h) \not\models \varphi_2 \\
(s,h) \models \varphi_1 \vee \varphi_2 & \text{iff } (s,h) \models \varphi_1 \text{ or } (s,h) \models \varphi_2 \\
(s,h) \models \neg\varphi & \text{iff } (s,h) \not\models \varphi
\end{array}
$$

Figure 3.1: The semantics of atomic formulae and boolean connectives. On this fragment it holds that $(s,h) \models^{\mathsf{st}} \varphi$ iff $(s,h) \models^{\mathsf{wk}} \varphi$, and we therefore write simply $\models$.

$$
\begin{array}{ll}
(s,h) \models^{\mathsf{wk}} \varphi_1 * \varphi_2 & \text{iff } \exists h_1, h_2.\ (s,h_1) \models^{\mathsf{wk}} \varphi_1, (s,h_2) \models^{\mathsf{wk}} \varphi_2, h_1 + h_2 \neq \bot, \text{ and } h = h_1 + h_2 \\
(s,h) \models^{\mathsf{st}} \varphi_1 * \varphi_2 & \text{iff } \exists h_1, h_2.\ (s,h_1) \models^{\mathsf{st}} \varphi_1, (s,h_2) \models^{\mathsf{st}} \varphi_2, h_1 \uplus^s h_2 \neq \bot, \text{ and } h = h_1 \uplus^s h_2 \\
(s,h) \models^{\mathsf{wk}} \varphi_1 \mathrel{-\!\!\circledast} \varphi_2 & \text{iff } \exists h_1.\ (s,h_1) \models^{\mathsf{wk}} \varphi_1, h + h_1 \neq \bot, \text{ and } (s, h + h_1) \models^{\mathsf{wk}} \varphi_2 \\
(s,h) \models^{\mathsf{st}} \varphi_1 \mathrel{-\!\!\circledast} \varphi_2 & \text{iff } \exists h_1.\ (s,h_1) \models^{\mathsf{st}} \varphi_1, h \uplus^s h_1 \neq \bot, \text{ and } (s, h \uplus^s h_1) \models^{\mathsf{st}} \varphi_2
\end{array}
$$

Figure 3.2: The weak-separation ($\models^{\mathsf{wk}}$) and strong-separation ($\models^{\mathsf{st}}$) semantics of spatial connectives.

connective. In full SSL, the disjunction is redundant, but we add it to the syntax to increase expressivity of the positive fragment.

The idea of the guarded negation comes from [24] and is not considered in [25]. All proofs related to the full SSL, however, remain sound because the guarded negation can be easily expressed in SSL. Proofs about positive formulae require to consider an additional case of the guarded negation. This case is usually straightforward since all properties of models of a positive guard φ also hold for all models of a formula $\varphi \wedge_\neg \psi$. The guarded negation is, in particular, useful to express validity of an entailment $\varphi \models \psi$ as unsatisfiability of the formula $\varphi \wedge_\neg \psi$:

$$\varphi \models \psi \text{ is valid } \Leftrightarrow\ \neg\varphi \vee \psi \text{ is valid } \Leftrightarrow\ \varphi \wedge \neg\psi \text{ is unsatisfiable.}$$

We write $\mathsf{vars}(\varphi)$ to denote the set of all variables in φ and define the set $\mathsf{vars}^+(\varphi)$ of variables that can be allocated as $\mathsf{vars}^+(\varphi) = \mathsf{vars}(\varphi) \setminus \{\mathsf{nil}\}$.

3.2 Weak- and Strong-Separation Semantics

We will define two logics – weak-separation logic (WSL) using the satisfaction relation $\models^{\mathsf{wk}}$ and strong-separation logic (SSL) using the satisfaction relation $\models^{\mathsf{st}}$. The semantics of atomic formulae and boolean connectives is given in Figure 3.1, and it is identical for both logics.

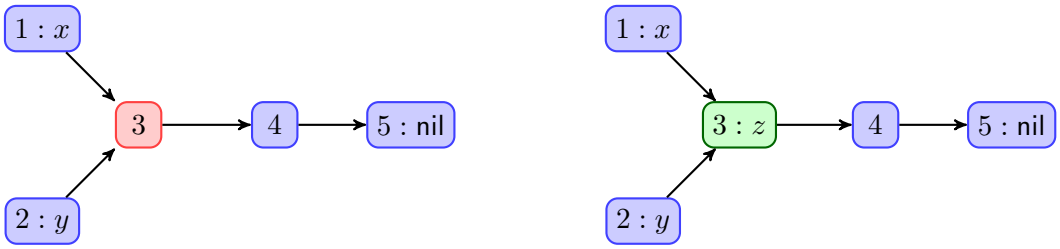


Figure 3.3: An example of two models of the formula $\varphi \triangleq (\mathsf{ls}(x, \mathsf{nil}) * \mathsf{true}) \wedge (\mathsf{ls}(y, \mathsf{nil}) * \mathsf{true})$ under the classical semantics. Under the strong-separation semantics, φ is satisfied only in the right model that can be split at the named location 3 to separate overlaid list-segments. The left model cannot be split using the operator $\uplus^s$ to satisfy the formula since its location 3 is not named.

Notice that, in the semantics of pure atoms, we additionally require that they can be satisfied on the empty heap only. This is the so-called *precise semantics* of pure atoms, and it is orthogonal to the strong-separation semantics. The semantics defined in this a way is common for translation-based decisions procedures [17, 28]. It does not change the expressivity, merely the way how formulae are written – instead of writing $x = y \wedge \varphi$, one can write $x = y * \varphi$ to express that the equality can be satisfied on the empty heap, which can always be split off from any heap. A symbolic heap formula now has the form $* \psi_i$ where all ψ_i are atomic formulae.

As will become clear later, the strong-separating conjunction cannot be used to define the list-segment predicate inductively because it would require all of its locations to be named. One can therefore either use a weak-separation conjunction or define list non-inductively. We chose the latter approach according to [25]. The list-segment predicate is defined to hold on a heap consisting of a possibly empty sequence of pointers starting with x and ending with y such that all locations in this sequence are distinct. Consequently, a list-segment cannot be cyclic or lasso-shaped. We may define the empty heap predicate and boolean constants as syntactic sugar[2]:

$$\mathsf{emp} \triangleq \mathsf{nil} = \mathsf{nil} \qquad \mathsf{false} \triangleq \mathsf{emp} \wedge_{\neg} \mathsf{emp} \qquad \mathsf{true} \triangleq \neg\mathsf{false}$$

The semantics of spatial connectives is defined in Figure 3.2, and, for both of them, it differs in the used notions of disjointness and disjoint union of heaps. Recall that, in the classical semantics, the disjoint union of two heaps is defined as the union of those heaps under the condition that their domains are disjoint:

$$h_1 + h_2 = \begin{cases} h_1 \cup h_2 & \text{if } \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset \\ \bot & \text{otherwise} \end{cases}$$

Strongly-disjoint union $\uplus^s$, parametrised by a stack s, also restricts images of heaps – it requires that each location shared by both heaps is named (i.e., at least one variable is mapped to it), formally, the strongly-disjoint union is defined as:

$$h_1 \uplus^s h_2 = \begin{cases} h_1 + h_2 & \text{if } \mathsf{locs}(h_1) \cap \mathsf{locs}(h_2) \subseteq \mathsf{img}(s) \\ \bot & \text{otherwise} \end{cases}$$

[2]While the constant false is expressible in the positive fragment, the constant true is not. Otherwise, it would be easy to introduce the negation even in the positive fragment using the guarded negation.

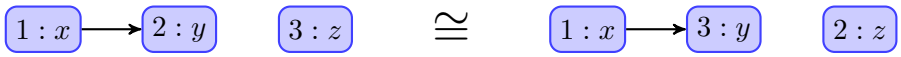


Figure 3.4: An example of two isomorphic stack-heap models. The isomorphism is given by the bijection σ such that $\sigma(2) = 3$, $\sigma(3) = 2$, and $\sigma(x) = x$ otherwise.

Notice that if $h_1 \uplus^s h_2$ is defined, then $h_1 + h_2$ is also defined, but not vice versa. This is demonstrated in Figure 3.3. It can be shown that the operator $\uplus^s$ gives rise to a separation algebra and it is therefore suitable for definition of semantics of separation logic [25].

We can define the magic wand using septraction and negation. Unlike the septraction, the magic wand is not expressible in the positive fragment.

$$\varphi \mathrel{-\!\!*} \psi \triangleq \neg(\varphi \mathrel{-\!\!\circledast} \neg\psi)$$

We can also define a list segment of length at least one and a proper list segment of length at least two. Notice that both of them lie in the positive fragment.

$$\mathsf{ls}_{\geq 1}(x, y) \triangleq \mathsf{ls}(x, y) * x \neq y$$
$$\mathsf{ls}_{\geq 2}(x, y) \triangleq \mathsf{ls}_{\geq 1}(x, y) \wedge_{\neg} x \mapsto y$$

Example 3.1. The formula $x \mapsto y \wedge x = y$ is unsatisfiable as the left-hand side requires a heap to be of size one and the right-hand size requires it to be empty. A correct way to express a *self-loop* pointer at x under the precise semantics is to write $x \mapsto y * x = y$.

Example 3.2. The formulae $x \neq y$ and $\neg(x = y)$ are not equivalent. The second can be satisfied by an arbitrary non-empty heap even if $s(x) = s(y)$.

Example 3.3. The formula $\varphi \triangleq (x \mapsto \mathsf{nil}) \mathrel{-\!\!\circledast} \mathsf{true}$ is satisfied by models that can be extended by a pointer from x to nil, i.e., by models that do not allocate x. This formula can also be expressed using the magic wand as $\neg((x \mapsto \mathsf{nil}) \mathrel{-\!\!*} \mathsf{false})$.

Satisfiability and entailment. As can be seen in Figure 3.3, satisfiability of an SSL formula may depend on how many variables are available to label splitting points of a heap. Satisfiability and entailment are therefore parametrised by a set of variables $\mathbf{x} \subseteq \mathbf{Var}$. Let $[\![\varphi]\!]_{\mathbf{x}}$ be the set of all models of φ over $\mathbf{x}$, i.e., $[\![\varphi]\!]_{\mathbf{x}} = \{(s, h) \mid \mathsf{dom}(s) = \mathbf{x} \ \wedge \ (s, h) \models^{\mathsf{st}} \varphi\}$. The formula φ with $\mathsf{vars}(\varphi) \subseteq \mathbf{x}$ is satisfiable over variables $\mathbf{x}$ if $[\![\varphi]\!]_{\mathbf{x}} \neq \emptyset$. An entailment $\varphi \models^{\mathsf{st}}_{\mathbf{x}} \psi$ is valid w.r.t. $\mathbf{x}$ if $[\![\varphi]\!]_{\mathbf{x}} \subseteq [\![\psi]\!]_{\mathbf{x}}$.

Two stack-heap models are *isomorphic* if they are identical up to renaming of locations.

Definition 3.1. *Two models (s_1, h_1) and (s_2, h_2) are isomorphic, $(s_1, h_1) \cong (s_2, h_2)$, if there exists a bijection on locations $\sigma : \mathbf{Loc} \leftrightarrow \mathbf{Loc}$ such that:*

1. *For all $x \in \mathbf{Var}$, it holds that $s_1(x) = \sigma(s_2(x))$.*
2. *For all $\ell \in \mathbf{Loc}$, it holds that $h_1(x) = \sigma(h_2(x))$.*

An example of two isomorphic models is given in Figure 3.4. It holds that SSL formulae cannot distinguish isomorphic models. This is not a consequence of the strong-separation semantics, but it follows from the fact that SSL cannot speak about concrete memory locations – it cannot express formulae such as $1 \mapsto 2$.

Lemma 3.1 (Isomorphic models [25])**.** *Let φ be a formula. Further, let (s_1, h_1) and (s_2, h_2) be two models such that $(s_1, h_1) \cong (s_2, h_2)$. Then $(s_1, h_1) \models^{\mathsf{st}} \varphi$ iff $(s_2, h_2) \models^{\mathsf{st}} \varphi$.*

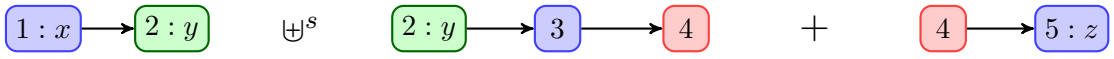


Figure 3.5: An example of a model (s, h) decomposed at locations 2 and 4. It holds that $(s, h) \models^{\mathsf{wk}} \mathsf{ls}^{wk}_{\geq 3}(x, z)$, but not $(s, h) \models^{\mathsf{st}} \mathsf{ls}^{wk}_{\geq 3}(x, z)$ because the location 4 is not named and the depicted decomposition is therefore not possible in SSL.

3.3 Comparison of Weak- and Strong-Separation Semantics

In this section, we compare the semantics of WSL and SSL. We will show that they coincide on the positive fragment. As the positive fragment subsumes frequently used fragments such as the symbolic heap fragment, this demonstrates a certain kind of *backward compatibility* of SSL. The second part of this section is devoted to examples where the strong-separation semantics actually makes a difference.

Recall that a location is dangling in a heap if it is in its image but not in its domain. If we have a model of a positive formula, it holds that all its dangling locations are named by stack variables.

Lemma 3.2 ([25])**.** *Let φ be a positive formula and let $(s, h) \models^{\mathsf{wk}} \varphi$ be its model. Then all dangling locations of the heap h are named, i.e.,* $\mathsf{dangling}(h) \subseteq s(\mathsf{vars}(\varphi))$.

Proof idea. By structural induction on φ, which actually proves a stronger statement that all dangling, joint (having multiple predecessors), and source (having no predecessors) locations are named. The case of the guarded negation $\psi_1 \wedge_\neg \psi_2$ uncovered in [25] follows directly from the inductive hypothesis for ψ_1. □

If we have two weakly-disjoint heaps, they can overlap only on locations that are dangling in at least one of them. Together with the previous lemma, this ensures that weakly-disjoint models of positive formulae can overlap only on named locations and they are therefore also strongly-disjoint. Therefore, there is no difference between the weak- and the strong-separation semantics for positive formulae.

Lemma 3.3 ([25])**.** *Let φ_1 and φ_2 be positive formulae and let $(s, h_1) \models^{\mathsf{wk}} \varphi_1$, $(s, h_2) \models^{\mathsf{wk}} \varphi_2$ be their models. Then $h_1 + h_2 \neq \bot$ iff $h_1 \uplus^s h_2 \neq \bot$.*

Proof.

(⇒) We want to prove that all shared locations of h_1 and h_2 are named. Let ℓ be a location shared by both heaps, i.e., $\ell \in \mathsf{locs}(h_1) \cap \mathsf{locs}(h_2)$. Then ℓ is dangling either in h_1 or h_2 as it cannot be in the domains of both of them. By Lemma 3.2, it holds that $\ell \in \mathsf{img}(s)$ and consequently $h_1 \uplus^s h_2 \neq \bot$.

(⇐) Follows directly from the definition of $\uplus^s$.

□

Theorem 3.1 (WSL and SSL coincide on the positive fragment [25])**.** *Let φ be a positive formula an let (s, h) be a model. Then $(s, h) \models^{\mathsf{wk}} \varphi$ iff $(s, h) \models^{\mathsf{st}} \varphi$.*

Proof idea. By structural induction on φ using Lemma 3.3 to prove cases of spatial connectives. □

The only formulae where the strong-separation semantics makes a difference are therefore those containing an (unguarded) negation.

It turns out that SSL cannot speak about concrete sizes of heaps without using additional variables. As an example, let us consider the following family of formulae for $n \geq 3$:

$$\mathsf{ls}^{wk}_{\geq n}(x, y) \;\triangleq\; \mathsf{ls}(x, y) \wedge (\underbrace{\neg\mathsf{emp} * \cdots * \neg\mathsf{emp}}_{n \text{ times}})$$

Under the weak-separation semantics, a formula $\mathsf{ls}^{wk}_{\geq n}(x, y)$ expresses that the heap is a list segment that can be split into n non-empty parts, i.e., a list segment of length at least n. In SSL, this is not necessarily true as can be seen in Figure 3.5 for $n = 3$. The list-segment in the figure has length greater than three, but cannot be split to three non-empty sub-heaps using the operator $\uplus^s$ since it does not contain enough named locations. In fact, a list segment of a length greater than three is not expressible in SSL. This is used in the AMS abstraction described in the next section. Regarding satisfiability, a formula $\mathsf{ls}^{wk}_{\geq n}(x, y)$ is satisfiable only if the considered set of variables provides enough variables to name all $n - 1$ locations needed to split the list segment.

Convention. In the rest of the thesis, we will be interested in SSL only and we will therefore write just $\models$ instead of $\models^{\mathsf{st}}$. Because of the correspondence on the positive fragment, we will assume an input set of variables $\mathbf{x}$ to be implicitly equal to $\mathsf{vars}(\varphi)$ when dealing with positive formulae.

3.4 Abstract Memory States

An *Abstract memory state* (AMS) is an abstraction over a stack-heap model which keeps just enough information to decide whether the model satisfies a formula or not. In [25], AMSs are used to prove essential theoretical results such as a small-model property of SSL and also as a building block of a decision procedure for it. The main idea of the decision procedure is to represent the possibly infinite set of stack-heap models $[\![\varphi]\!]_{\mathbf{x}}$ by a finite set of abstract memory states $\alpha(\varphi)$ whose emptiness can be decided in polynomial space.

In this section, we will gradually show how a model (s, h) can be abstracted to its induced abstract memory state $\mathsf{ams}(s, h)$. The cornerstone of this abstraction is a *memory chunk* – a minimal non-empty sub-heap $h' \subseteq h$ such that h' can be cut off h according to the strong-separation semantics.

Definition 3.2 (Memory chunk)**.** *Let (s, h) be a model and let h_1 be a heap. We say that the heap h_1 is a sub-heap of h, denoted as $h_1 \sqsubseteq h$, if there exists a heap h_2 such that $h_1 \uplus^s h_2 = h$. We call h_1 a memory chunk of h if it is non-empty, minimal sub-heap of h, i.e., there is no non-empty $h'_1 \neq h_1$ such that $h'_1 \sqsubseteq h_1$.*

We classify chunks into two categories – a chunk h_c is *positive* if there exists an atomic formula φ such that $(s, h_c) \models \varphi$. Otherwise, the chunk is *negative*. Notice that all positive chunks are either cyclic pointers or non-empty list segments. The decomposition of a model to its chunks always exists and is uniquely determined.

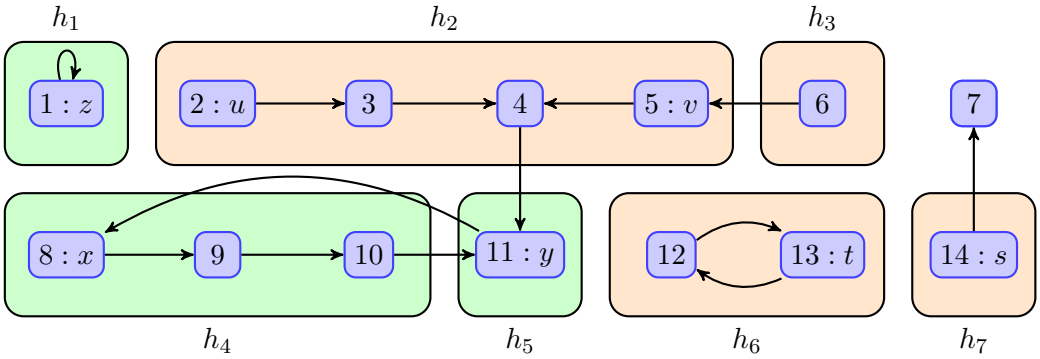


Figure 3.6: An example of a model and its decomposition into chunks.The positive chunks h_1, h_4, and h_5 are marked by the green colour, and the negative chunks by the orange colour.

Lemma 3.4 (Decomposition to chunks [25])**.** *Let (s, h) be a model and let $h_1, \ldots, h_n$ be its chunks. Then $h = h_1 \uplus^s \cdots \uplus^s h_n$.*

Proof idea. The claim follows from the fact that all sub-heaps form a boolean algebra with chunks being atoms of this algebra. □

Example 3.4. An example of a decomposition of a model into its chunks and their classification is shown in Figure 3.6. The chunk h_1 is positive since it is a model of a formula $z \mapsto z$. The negative chunk h_2 consists of two overlaid list-segments that cannot be further split according to the strong-separation semantics. The negative chunk h_3 is a so-called *garbage* chunk because it consists of the memory location 6 that cannot be reached using stack variables. The chunks h_4 and h_5 are positive as they are models of the formulae $\mathsf{ls}(x, y)$ and $y \mapsto x$, respectively. The chunk h_6 is negative because list segments cannot be cyclic. Finally, the chunk h_7 is negative since its sink location 7 is anonymous.

We can use decomposition into chunks to abstract a model to an abstract memory state.

Definition 3.3. *An abstract memory state is a quadruple $\mathcal{A} = (V, E, \rho, \gamma)$ where*

- *$V = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ is a partition of some finite set of variables,*
- *$E : V \rightharpoonup V \times \{=1, \geq 2\}$ is a partial function such that, for all $\mathbf{v} \in \mathsf{dom}(E)$, it holds that $\mathsf{nil} \notin \mathbf{v}$,*
- *ρ is a set of disjoint subsets of V such that, for all $R \in \rho$, it holds that (1) R is disjoint from $\mathsf{dom}(V)$ and (2) $\mathsf{nil} \notin R$,*
- *$\gamma \in \mathbb{N}$ is a natural number.*

The components have the following interpretation. The elements of the partition V are called *vertices*. The partition abstracts some stack s. Instead of storing the mapping of s, it only keeps information about which variables alias – two variables x and y are in the same equivalence class of V iff $s(x) = s(y)$. The function E represents *edges* of AMS induced by positive chunks. An edge $(x, y, =1)$ represents a chunk consisting of a single pointer from x to y, and, similarly, an edge $(x, y, \geq 2)$ abstracts a chunk which is a list segment of a

length at least two (this abstraction follows from the fact that SSL cannot speak about list segments of length greater than two without using additional variables).

The last two components are related to negative chunks. The component ρ represents *negative-allocation constraints*. It is a set of disjoint sets R where each R is a set of vertices that are allocated within the same negative chunk. Finally, the number γ is called the *garbage-chunk count*, and it corresponds to the number of negative chunks that do not allocate any variables.

To define an induced AMS of a model (s, h) formally, we need several auxiliary definitions. Let s be a stack. We define an *alias-equivalence* $=_s$ w.r.t. s as $x =_s y \Leftrightarrow s(x) = s(y)$. We write $[x]_s$ to denote the equivalence class of $=_s$ containing x. We also define the set of equivalence classes of $=_s$ allocated in a chunk h_c as $\mathsf{alloc}_s^{=}(h_c) = \{[x]_s \mid s(x) \in \mathsf{dom}(h_c)\}$.

Definition 3.4 (Induced AMS of a model)**.** *Let (s, h) be a model. Let $\mathsf{chunks}^+(s, h)$ and $\mathsf{chunks}^-(s, h)$ be its positive and negative chunks, respectively. We define the induced AMS of the model (s, h), $\mathsf{ams}(s, h) = (V, E, \rho, \gamma)$, as:*

- $V = \{[x]_s \mid x \in \mathsf{dom}(s)\}$
- $E([x]_s) = \begin{cases} ([y]_s, = 1) & \text{if } (s, h_c) \models x \mapsto y \text{ for some } h_c \in \mathsf{chunks}^+(s, h) \\ ([y]_s, \geq 2) & \text{if } (s, h_c) \models \mathsf{ls}_{\geq 2}(x, y) \text{ for some } h_c \in \mathsf{chunks}^+(s, h) \\ \bot & \text{otherwise} \end{cases}$
- $\rho = \{\mathsf{alloc}_s^{=}(h_c) \mid h_c \in \mathsf{chunks}^-(s, h)\}$
- $\gamma = |\mathsf{chunks}^-(s, h)| - |\rho|$

Lemma 3.5 ([25])**.** *Let (s, h) be a stack-heap model. Then $\mathsf{ams}(s, h)$ is an AMS.*

Example 3.5. An example of a model (s, h) and its AMS $\mathcal{A} = \mathsf{ams}(s, h) = (V, E, \rho, \gamma)$ is depicted in Figure 3.7. There are three positive chunks in the model: h_1, h_2, and h_6. The chunks h_1 and h_2 are list segments of length greater or equal than two and are therefore abstracted using edges with label ≥ 2. The chunk h_6 consists of a single pointer and is therefore represented using an edge with label $= 1$. The negative chunk h_4 is the only chunk which does not allocate any variables, and therefore we have that garbage-chunk count $\gamma = 1$. Finally, there are two negative chunks allocating some variables. The chunk h_3 allocates $R_1 = \{\{u, v\}, \{s\}, \{t\}\}$, and the chunk h_5 allocates $R_2 = \{\{w\}\}$. The negative-allocation constraints are $\rho = \{R_1, R_2\}$.

Deciding SSL using AMSs. We conclude this section by sketching a decision procedure based on AMSs. The decision procedure is based on the following theorem.

Theorem 3.2 (Refinement theorem [25])**.** *Let φ be a formula and let (s, h_1) and (s, h_2) be models such that $\mathsf{ams}(s, h_1) = \mathsf{ams}(s, h_2)$. Then $(s, h_1) \models \varphi$ iff $(s, h_2) \models \varphi$.*

Given an input φ and $\mathbf{x}$, the decision procedure first guesses a stack s (there are only finitely many stacks with the domain $\mathbf{x}$) and then computes the set of abstract memory states $\alpha_s(\varphi) = \{\mathsf{ams}(s, h) \mid h \text{ is a heap such that } (s, h) \models \varphi\}$ inductively on the structure of the formula φ. Observe that given the stack s, the set of vertices of an AMS $\mathsf{ams}(s, h)$ is finite for an arbitrary heap h. Consequently, there is also finitely many edges and finitely many allocation constraints. To finish the construction, we need to provide an upper bound

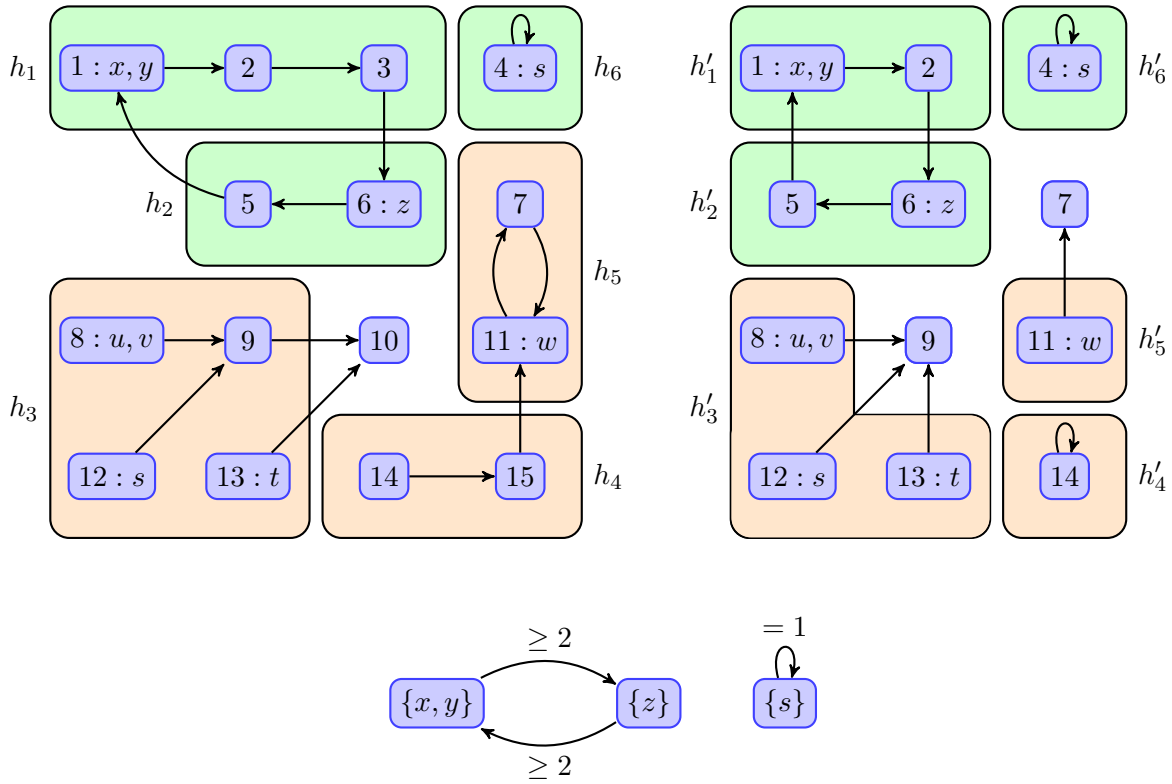


Figure 3.7: An example of a stack-heap model (s, h) (top left) and its reduction $\mathsf{reduce}(s, h)$ (top right). It holds that both models induce the same AMS (bottom) and therefore cannot be distinguished by SSL formulae.

on the number of garbage chunks. This is given by the *chunk size* $\lceil\varphi\rceil$ which gives an upper bound on the number of chunks needed to satisfy and/or falsify the formula φ:

$$\begin{aligned}
\lceil x = y\rceil = \lceil x \neq y\rceil = \lceil x \mapsto y\rceil = \lceil \mathsf{ls}(x, y)\rceil &= 1 \\
\lceil \psi_1 * \psi_2\rceil &= \lceil\psi_1\rceil + \lceil\psi_2\rceil \\
\lceil \psi_1 \mathrel{-\!\circledast} \psi_2\rceil &= \lceil\psi_2\rceil \\
\lceil \psi_1 \wedge \psi_2\rceil = \lceil \psi_1 \wedge_\neg \psi_2\rceil = \lceil \psi_1 \vee \psi_2\rceil &= \mathsf{max}(\lceil\psi_1\rceil, \lceil\psi_2\rceil) \\
\lceil \neg\psi\rceil &= \lceil\psi\rceil
\end{aligned}$$

Now, a refined version of Theorem 3.2 can be proved.

Theorem 3.3 (Refined refinement theorem [25]). *Let φ be a formula with $\lceil\varphi\rceil = k$. Let $m \geq k$ and $n \geq k$ and let (s, h_1), (s, h_2) be models with $\mathsf{ams}(s, h_1) = (V, E, \rho, m)$ and $\mathsf{ams}(s, h_2) = (V, E, \rho, n)$. Then $(s, h_1) \models \varphi$ iff $(s, h_2) \models \varphi$.*

Based on this theorem, a finite abstraction of the set $\alpha_s(\varphi)$ can be defined and its non-emptiness (corresponding to satisfiability of φ) can be checked in polynomial space. Since the construction is rather technical, we refer to [25] for more details.

3.5 Small-Model Property

Using Theorem 3.3, we can prove a *small-model property* for SSL and its variant for SSL$^+$. The small-model property states that each satisfiable formula has a model of linear size. Since the property is crucial for our later proposed translation to SMT, we will modify the proofs from [25] to show more precise bounds.

We will first define a *reduction* of a model (s, h) which is obtained by reducing each chunk h_c of (s, h) to a chunk h'_c such that the composition of reduced chunks will yield the same induced AMS as the original model. By Theorem 3.2, those models will satisfy exactly the same formulae.

Definition 3.5. *Let (s, h) be a model and let $h_1, \ldots, h_n$ be its chunks. We define its reduction,* $\mathsf{reduce}(s, h) = (s, h')$ *where* $h' = h'_1 \uplus^s \cdots \uplus^s h'_n$ *and where each h'_i is obtained from h_i by the chunk reduction defined below. Let* $\mathsf{alloc}_s(h_c) = \{s(x) \mid s(x) \in \mathsf{dom}(h_c)\}$*. If a chunk is positive, then its reduction is defined as:*

$$\mathsf{reduce_c}(h_i) = \begin{cases} h_i & \text{if } (s, h_i) \models x \mapsto y \text{ for some } x, y \in \mathbf{Var} \\ \{s(x) \mapsto h(s(x)), h(s(x)) \mapsto s(y)\} & \text{if } (s, h_i) \models \mathsf{ls}_{\geq 2}(x, y) \text{ for some } x, y \in \mathbf{Var} \end{cases}$$

If a chunk is negative, then the reduction is defined as:

$$\mathsf{reduce_c}(h_i) = \begin{cases} \{s(x_1) \mapsto \ell, \ldots, s(x_n) \mapsto \ell\} & \text{if } \mathsf{alloc}_s(h_i) = \{s(x_1), \ldots, s(x_n)\} \text{ for } n > 0 \\ & \text{and } \ell = h(\mathsf{min}\{s(x_1), \ldots, s(x_n)\}) \\ \{\ell \mapsto \ell\} & \text{if } \mathsf{alloc}_s(h_i) = \emptyset \text{ and } \ell = \mathsf{min}(\mathsf{dom}(h_i)) \end{cases}$$

Example 3.6. The reduction of a model is demonstrated in Figure 3.7 where each chunk h_i of the left model is reduced to a chunk h'_i of the right model. First, observe that the stack of both models is the same. Now, we will describe the reduction of the individual chunks. The chunk h_1 is a list segment of length three. and its reduction therefore removes the location 3. The chunk h_2 is a list segment of length two, and therefore it remains unchanged. Similarly, also the pointer chunk h_6 remains unchanged.

The negative chunks h_3 and h_5 that allocate some variables are replaced by minimal negative chunks that allocate those variables without changing the stack. Finally, the garbage chunk h_4 is replaced by the minimal garbage chunk – an anonymous self-loop pointer. It can be easily verified that both models induce the same AMS in the bottom part of the figure.

Now, we have to show that the reduction is well-defined and preserves the induced AMS. Since we use our own definition of the reduction, we will prove those properties thoroughly.

Lemma 3.6. *Let (s, h) be a model. Then* $\mathsf{reduce}(s, h)$ *is well-defined.*

Proof. We will first show that all reduced chunks are well-defined. If a chunk is a single pointer, then its reduction is well-defined because it does not change the chunk. If a chunk is a list segment of length at least two, we need to show that $h(s(x))$ is defined. This follows from the fact that the chunk has length at least two. There are no other types of positive chunks.

If a negative chunk h_i allocates some variables, then $\mathsf{reduce}_c(h_i)$ maps all allocated variables to the location ℓ where $\ell = h(\mathsf{min}\{s(x_1), \ldots, s(x_n)\})$. The set $\{s(x_1), \ldots, s(x_n)\}$ is non-empty, and its minimum is defined because of our assumption that there exists

some fixed linear order on the location domain **Loc**. Since all $s(x_i)$ are allocated in the chunk, it holds that $h(\mathsf{min}\{s(x_1), \ldots, s(x_n)\})$ is defined. Finally, if a chunk h_i is garbage (it does not allocate any variables), then we replace it by a single self-pointer on the minimal location from $\mathsf{dom}(h_1)$. The set $\mathsf{dom}(h_1)$ is non-empty because the chunk is defined to be a non-empty heap.

It remains to show that $\mathsf{reduce}_c(h_1) \uplus^s \cdots \uplus^s \mathsf{reduce}_c(h_n)$ is defined. It is enough to show that $\mathsf{dom}(\mathsf{reduce}_c(h_i)) \subseteq \mathsf{dom}(h_i)$ and that $\mathsf{locs}(\mathsf{reduce}_c(h_i)) \subseteq \mathsf{locs}(h_i)$. This holds because all cases of the reduction can only remove locations from domains and images. □

Lemma 3.7. *Let φ be a formula. Further, let (s, h) be a model and let $\mathsf{reduce}(s, h)$ be its reduction. Then $(s, h) \models \varphi$ iff $\mathsf{reduce}(s, h) \models \varphi$.*

Proof. Let $\mathcal{A}_1 = \mathsf{ams}(s, h)$ and let $\mathcal{A}_2 = \mathsf{ams}(\mathsf{reduce}(s, h))$. We will show that $\mathcal{A}_1 = \mathcal{A}_2$. The rest follows from Theorem 3.2. First, observe that the reduction does not change the stack, and both models therefore induce AMSs with the same set of vertices. Then, observe that the reduction preserves the number of chunks. Further, it holds that each positive chunk of some model defines exactly one edge of its induced AMS. Let h_i be a positive chunk, then $\mathsf{reduce}_c(h_i)$ is also a positive chunk and moreover defines exactly the same edge as h_i.

Let h_i be a negative chunk such that it allocates some variables. First observe that $\mathsf{reduce}_c(h_i)$ is also a negative chunk as its sink location ℓ is anonymous – if it would not, h_i would not be a chunk since it could be further decomposed by cutting off $\{s(x_1) \mapsto \ell\}$. The reduction also allocates exactly the same variables and therefore produces the same negative-allocation constraint.

Finally, let h_i be a garbage chunk. Then, $\mathsf{reduce}_c(h_i)$ is also a garbage chunk. Therefore, the garbage-chunk count of $\mathcal{A}_1$ is equal to the garbage-chunk count of $\mathcal{A}_2$. Thus, $\mathcal{A}_1 = \mathcal{A}_2$. □

Now, we are ready to prove the small-model property. We will start with the case of a positive formula. The bound is based on the fact that a model of a positive formula consists of positive chunks only and that the size of a reduced chunk is at most two.

Lemma 3.8 ([25])**.** *Let φ be a positive formula and let $(s, h) \models \varphi$ be its model. Further, let $h = h_1 \uplus^s \cdots \uplus^s h_n$ be the decomposition of the model into its chunks. Then all chunks h_i are positive.*

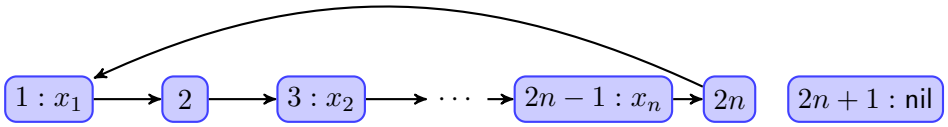
Theorem 3.4 (Small-model property for SSL$^+$)**.** *Let φ be a satisfiable positive formula. Then there exists a model (s, h) s.t. $(s, h) \models \varphi$ and $|\mathsf{locs}(h)| \leq 2n{+}1$ where $n = |\mathsf{vars}^+(\varphi)|$.*

Proof. Since φ is satisfiable, there exists some model $(s, h) \models \varphi$ with $\mathsf{dom}(s) = \mathsf{vars}(\varphi)$. Let $(s, h') = \mathsf{reduce}(s, h)$ be its reduction. By Lemma 3.7, we have that $(s, h') \models \varphi$. By Lemma 3.8, both h and h' consist of positive chunks only. There is at most $|\mathsf{vars}^+(\varphi)|$ chunks of heap h' because each positive chunk has to allocate at least one variable and nil cannot be allocated. Finally, each reduced chunk consists of at most two unique locations (since we consider the worst case when all variables are allocated, its named sink does not count as it is already allocated and counted in some other chunk). One additional location is needed for nil. Therefore we have that $|\mathsf{locs}(h')| \leq 2 \cdot |\mathsf{vars}^+(\varphi)| + 1$. □

Example 3.7. To demonstrate that the bound is tight for positive formulae, let us consider the family of formulae defined as for $n \geq 2$:

$$\varphi_n \triangleq \mathsf{ls}_{\geq 2}(x_1, x_2) * \mathsf{ls}_{\geq 2}(x_2, x_3) * \cdots * \mathsf{ls}_{\geq 2}(x_{n-1}, x_n) * \mathsf{ls}_{\geq 2}(x_n, x_1)$$

A formula φ_n is satisfiable only by a cycle consisting of n list-segments, each of them having length at least two.



Outside the positive fragment, we have to consider also the input set of variables $\mathbf{x}$ and the number of garbage chunks $\lceil\varphi\rceil$ needed to satisfy the formula.

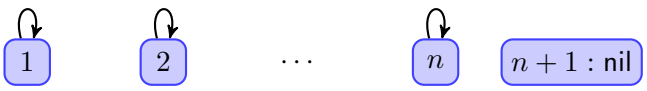
Theorem 3.5 (Small-model property for SSL)**.** *Let φ be a satisfiable formula. Then there exists a model (s,h) such that $(s,h) \models \varphi$ and $|\mathsf{locs}(h)| \leq 2n + \lceil\varphi\rceil + 1$ where $n = |\mathbf{x} \setminus \{\mathsf{nil}\}|$.*

Proof. We proceed similarly as for the positive fragment. Since φ is satisfiable, there exists some model (s,h) with $\mathsf{dom}(s) = \mathbf{x}$. Let $(s,h') = \mathsf{reduce}(s,h)$ be its reduction. By Lemma 3.7, we have that $(s,h') \models \varphi$. In the worst case, we have that there is at most $k = |\mathbf{x} \setminus \{\mathsf{nil}\}|$ allocated variables and consequently at most k non-garbage chunks. Observe that, in such a case, each chunk allocates exactly one variable. Therefore there are at most two locations in each non-garbage negative chunk. From Theorem 3.4 and the fact that a variable cannot be allocated in two chunks, we have that each non-garbage chunk of the reduced model have at most two unique locations (no matter whether it is positive or negative). All non-garbage chunks therefore have at most $2k$ locations. By Theorem 3.3, there is at most $\lceil\varphi\rceil$ garbage chunks needed to satisfy φ. Finally, one additional location is needed for nil. Therefore $|\mathsf{locs}(h')| \leq 2 \cdot |\mathbf{x} \setminus \{\mathsf{nil}\}| + \lceil\varphi\rceil + 1$. □

Example 3.8. To demonstrate that the bound is tight, let us consider the set of variables $\mathbf{x} = \{\mathsf{nil}\}$ and the family of formulae defined as:

$$\varphi_n \triangleq \underbrace{\neg\mathsf{emp} * \cdots * \neg\mathsf{emp}}_{n \text{ times}}$$

For a formula φ_n, it holds that $\mathbf{x} \setminus \{\mathsf{nil}\} = \emptyset$ and $\lceil\varphi_n\rceil = n$. The minimal model satisfying φ_n is the following:



Based on small-model properties, we define a *location bound* of a formula φ w.r.t. the set of variables $\mathbf{x}$:

$$\mathsf{bound}(\varphi, \mathbf{x}) = \begin{cases} 2 \cdot |\mathsf{vars}^{+}(\varphi)| + 1 & \text{if } \varphi \text{ is positive} \\ 2 \cdot |\mathbf{x} \setminus \{\mathsf{nil}\}| + \lceil\varphi\rceil + 1 & \text{otherwise} \end{cases}$$

Usually, a tighter location bound can be computed based on the structure of the formula φ. This computation is discussed in Section 5.

Chapter 4

Decision Procedure for SSL

This chapter presents the main contribution of this thesis – a new decision procedure for a fragment of strong-separation logic. As was already sketched in the introduction, we will not follow the enumeration-based approach presented in [25], but we will rather propose a translation of SSL to SMT to leverage capabilities of modern SMT solvers. Our translation is inspired by previous works targeting boolean combinations of symbolic heaps of WSL [17, 28] and the same fragment of SSL [24]. We extend this fragment in several non-trivial ways:

1. We add support for the septraction connective. In the original translations, it was always enough to consider a single heap to find a model of a formula. In the presence of septractions, additional heaps are needed to find witnesses of their satisfaction. We limit ourselves to a fragment where septractions do not appear under negations (both guarded and classical). It is therefore not possible to express arbitrary magic wands, but one can, for example, check validity of entailments such as $\varphi \models \psi \mathrel{-\!\!*} \chi$ after applying boolean transformations to represent its counterexample as $\varphi \wedge (\psi \mathrel{-\!\!\circledast} \neg\chi)$.

2. We allow arbitrary mixing and nesting of spatial and boolean connectives except unguarded negation. In the original translations, boolean operators cannot appear under separating conjunctions. The main complication is the disjunction which breaks a so-called unique footprint property used in the original translations to effectively translate separating conjunctions. When allowing disjunctions to appear under separating conjunctions, we need to overwork the original approach to work with multiple footprints. This may lead to an exponential size of the translated formula.

3. We allow arbitrary appearance of negations (except the limitation related to septractions). Unlike [24], which translates a fragment of SSL on which it coincides with WSL, we need to also consider the strong-separation semantics of spatial connectives. A negation appearing under spatial connectives was mentioned as a hard challenge in [28]. Our changes from Point 2 make its support easier – but for a price of even more significant exponential blow-up caused by an extensive enumeration of possible footprints. We present some heuristics to tackle this, but there remains a lot of space for future work, e.g., to perform enumeration lazily as in [30].

Further, we propose a more effective translation of list-segment predicates than in [17, 24] – we improve its size from $\mathcal{O}(n^4)$ to $\mathcal{O}(n^3)$. On the other hand, we do not consider trees and data predicates. However, we plan to focus on those extensions in our future work.

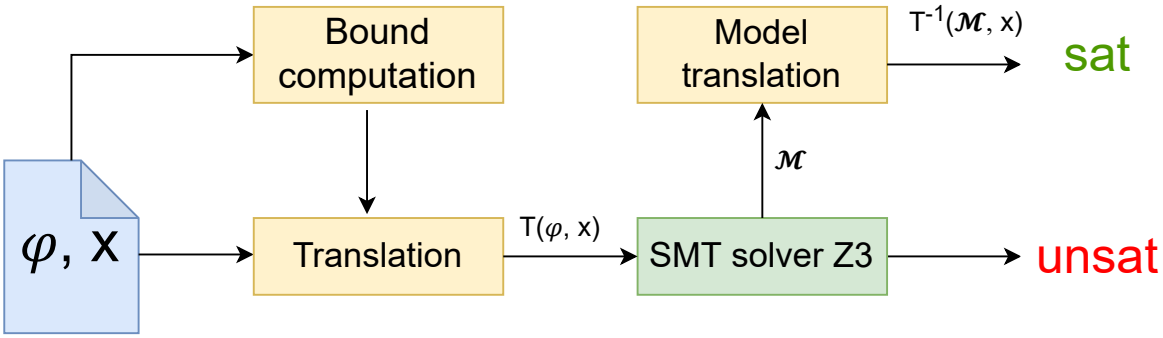


Figure 4.1: A schematic illustration of the proposed decision procedure.

Chapter outline. In the subsequent sections we describe how we translate particular ingredients of SSL – list-segment predicates, separating conjunctions, and septractions. We put those ingredients together in Section 4.5. The section also briefly discusses complexity issues related with the translation and defines a fragment of SSL that can be effectively translated using our approach. Finally, we prove the correctness of the translation in Section 4.6.

4.1 Overview

A high-level overview of our decision procedure is given in Figure 4.1. The input is a formula φ and a set of variables $\mathbf{x}$. The decision procedure first computes its location bound and bounds on lengths of list-segment predicates. Throughout this chapter, we consider the most general bounds. An improved bound computation is discussed in Section 5. Using those bounds, the input formula φ is translated to a first-order formula $\mathsf{T}(\varphi, \mathbf{x})$ in a combined theory of sets and arrays. The formula is then solved by an SMT solver. If the solver returns `unsat`, we are done. If the solver finds a first-order model $\mathcal{M}$, we will perform an inverse translation of this model to obtain an equivalent stack-heap model $\mathsf{T}^{-1}(\mathcal{M}, \mathbf{x})$.

Idea of the encoding. SSL naturally speaks about partial functions, but those are not supported in SMT. We will therefore use a pair of an array $\mathbf{h}$ an a set D to encode a partial heap function – the array $\mathbf{h}$ encodes the mapping of the heap, and the set D encodes its domain. A stack image of a variable $x \in \mathbf{x}$ is encoded simply by a constant symbol x of the same name. If the translated formula is satisfiable, its model $\mathcal{M}$ can be converted to a stack-heap model $\mathsf{T}^{-1}(\mathcal{M}, \mathbf{x})$ using an inverse translation.

Definition 4.1 (Inverse translation)**.** *Let $\mathcal{M}$ be a first-order model. We define its inverse translation $\mathsf{T}^{-1}(\mathcal{M}, \mathbf{x}) = (s, h)$ as:*

$$s(x) = \begin{cases} x^{\mathcal{M}} & \text{if } x \in \mathbf{x} \cup \{\mathsf{nil}\} \\ \bot & \text{otherwise} \end{cases} \qquad h(\ell) = \begin{cases} \mathbf{h}[\ell]^{\mathcal{M}} & \text{if } \ell \in D^{\mathcal{M}} \\ \bot & \text{otherwise} \end{cases}$$

In our translation, we utilise the small model property of SSL to restrict the infinite domain of locations $\mathbf{Loc}$ to its finite subset $\mathbf{L} = \{\ell_1, \ell_2, \ldots, \ell_n\}$ consisting of n distinct location constants. Because SSL formulae cannot distinguish isomorphic models, it does not matter which particular subset we choose.

The definition of the location domain is ensured by the following formula[1]:

$$\Delta_n^{\mathbf{L}} \triangleq \exists \ell_1, ..., \ell_n.\ \mathsf{distinct}(\ell_1, ..., \ell_n) \wedge \forall \ell.\ \bigvee_{1 \leq i \leq n} \ell = \ell_i$$

In order for $\mathsf{T}^{-1}(\mathcal{M}, \mathbf{x})$ be a correctly-defined stack heap model, we need to ensure that it does not allocate nil. Together with the definition of $\mathbf{L}$, we call this as the *well-formedness* constraint:

$$\Delta_n^{\mathrm{WF}} \triangleq \Delta_n^{\mathbf{L}} \wedge \mathsf{nil} \notin D$$

Before we will continue with the definition of the translation in Section 4.5, we will describe ideas used to translate individual ingredients of SSL.

4.2 Translation of List-Segment Predicates

The translation of list-segment predicates is complicated by the fact that they essentially speak about reachability which is not expressible in first-order logic. Fortunately, we can leverage the small-model property and use a form of bounded reachability parametrised by the number of locations. We first define an alternative semantics of list-segment predicates in terms of paths in induced graphs.

Lemma 4.1. *Let (s, h) be a model and let $G[(s, h)]$ be its induced graph. It holds that $(s, h) \models \mathsf{ls}(x, y)$ iff there exists a simple path π such that $x \overset{\pi}{\rightsquigarrow} y$ and $\mathsf{dom}(\pi) = \mathsf{dom}(h)$.*

Proof.

($\Rightarrow$) By a case distinction on the semantics of $\mathsf{ls}(x, y)$. If the list segment is empty, then $s(x) = s(y)$ and $\mathsf{dom}(h) = \emptyset$. Then there exists the path $\pi = \langle x \rangle$ with $\mathsf{dom}(\pi) = \emptyset$. Otherwise, there are distinct locations $\ell_0, \ldots, \ell_n$ such that $h = \{\ell_0 \mapsto \ell_1, ..., \ell_{n-1} \mapsto \ell_n\}$ and $s(x) = \ell_0, s(y) = \ell_n$. Thus, there is a simple path $\pi = \langle \ell_0, \ldots \ell_n \rangle$ with the domain $\mathsf{dom}(\pi) = \{\ell_0, \ldots, \ell_{n-1}\} = \mathsf{dom}(h)$, which concludes this direction of the proof.

($\Leftarrow$) Analogically, by considering the case of the empty path and the case of a non-empty simple path.

□

Now we will define two predicates expressing the existence of a simple path from x to y and a fact that some set D equals to the domain of this path. Both predicates will be parametrised by an interval $[m, n]$ limiting possible lengths of the considered paths. In this chapter, we will always use the most general intervals, i.e., $[0, n]$ where n is the location bound of an input formula. The intuitive meaning of predicates is the following:

$\mathsf{reach}_{[m,n]}(\mathbf{h}, x, y)$	There exists a simple path $x \overset{\pi}{\rightsquigarrow} y$ with $m \leq \lvert\pi\rvert \leq n$.
$\mathsf{path}_{[m,n]}(\mathbf{h}, D, x, y)$	$\begin{cases} D = \mathsf{dom}(\pi) & \text{if there is a simple path } x \overset{\pi}{\rightsquigarrow} y \text{ with } m \leq \lvert\pi\rvert \leq n, \\ D = \emptyset & \text{if there is no such path.} \end{cases}$

[1] In the actual implementation of our translation in many-sorted logic used by SMT solvers, we can equivalently declare $\mathbf{L}$ to be a datatype with n constant constructors $\ell_1, \ldots, \ell_n$.

In the case when there is no path, the predicate asserts D be the empty set. This is for the consistency with later defined footprints (Definition 4.3).

Definition of reachability predicates. The definition of reachability predicates will be based on the following lemma characterising paths in induced graphs. Because the successor of a vertex is given by a partial function, it is uniquely determined. Consequently, if there exists a simple path, it is uniquely determined.

Lemma 4.2. *Let (s, h) be a model and let $G[(s, h)]$ be its induced graph. Let π be a path from x to y in $G[(s, h)]$. Then this path is uniquely determined as $\pi = \langle x, h(x), \ldots, h^{|\pi|}(x)\rangle$.*

Proof. By induction on the length of the path π. If π is empty, i.e., $\pi = \langle x\rangle$, then π is clearly uniquely determined. If π has length $n+1$, then its prefix $\pi' = \langle x, h(x), \ldots, h^n(x)\rangle$ is, by the inductive hypothesis, uniquely determined. Since there is at most one successor of each vertex, the only way to obtain a path of length $n+1$ is to extend π' by an edge $h^n(x) \to h^{n+1}(x)$ which yields a uniquely determined path $\pi = \langle x, h(x), \ldots, h^{n+1}(x)\rangle$. □

Notice that in stack-heap models, each vertex has *at most* one successor, but in our SMT encoding, each vertex has *exactly one successor* since arrays are total. As a consequence of the previous lemma, there is a path from x to y of length i iff $\mathbf{h}^i[x] = y$[2]. The reachability in a number of steps given by some interval can be then defined using enumeration over all lengths in the interval:

$$\mathsf{reach}^i(\mathbf{h}, x, y) \triangleq \mathbf{h}^i[x] = y$$
$$\mathsf{reach}_{[m,n]}(\mathbf{h}, x, y) \triangleq \bigvee_{m \le i \le n} \mathsf{reach}^i(\mathbf{h}, x, y)$$

To define the predicate path, we first define a predicate $\mathsf{reachable}^{<i}(\mathbf{h}, D, x)$ which asserts that D is the set of all locations reachable from x in less than i steps. The predicate again uses the fact that the successor of a vertex is uniquely determined.

$$\mathsf{reachable}^{<i}(\mathbf{h}, D, x) \triangleq \begin{cases} D = \emptyset & \text{if } i = 0 \\ D = \{x, \mathbf{h}[x], \mathbf{h}^2[x], ..., \mathbf{h}^{i-1}[x]\} & \text{if } i > 0 \end{cases}$$

Now we will define the predicate path. The most tricky part is to ensure that it will indeed always assert that D is the domain of *the simple path* – although each vertex has exactly one successor, there could still be multiple i such that $\mathbf{h}^i[x] = y$. Only the smallest such i defines a simple path. This is not a problem for reachability, but we need to select the correct i to compute the correct domain of the list segment. Instead of postulating the shortest path from x to y, we use the fact that the unique simple path is a prefix of all other paths from x to y. Therefore, the simple path is the only path from x to y that does not go through the location y:

$$\mathsf{path}_{[m,n]}(\mathbf{h}, D, x, y) \triangleq \bigvee_{m \le i \le n} \Big(\mathsf{reach}^i(\mathbf{h}, x, y) \ \wedge \ \mathsf{reachable}^{<i}(\mathbf{h}, D, x) \ \wedge \ y \notin D\Big)$$
$$\vee \ \Big(\neg\mathsf{reach}^i(\mathbf{h}, x, y) \ \wedge \ D = \emptyset\Big)$$

[2]The term $\mathbf{h}^i[x]$ denotes i-timed iterated reading from the array $\mathbf{h}$. This can be formally defined using recursion as $\mathbf{h}^i[x] = x$ if $i = 0$, and $\mathbf{h}[\mathbf{h}^{i-1}[x]]$ otherwise.

The predicate performs an enumeration over all paths from x to y and forces D to be the domain of a path that does not contain y. If there is no path from x to y, it sets D to be empty. It remains to formally show that the introduced predicates have their intended meanings.

Lemma 4.3. *Let $\mathcal{M}$ be a first-order model and let $(s,h) = \mathsf{T}^{-1}(\mathcal{M},\mathbf{x})$ be a stack-heap model obtained by its inverse translation. Let G be the induced graph of (s,h). Then the following conditions hold:*

1. $\mathcal{M} \models \Delta_n^{\mathrm{WF}} \ \wedge\ \mathsf{reach}^i(\mathbf{h},x,y) \ \Leftrightarrow\ \exists\pi.\ x \overset{\pi}{\rightsquigarrow} y \wedge |\pi| = i$
2. $\mathcal{M} \models \Delta_n^{\mathrm{WF}} \ \wedge\ \mathsf{reach}_{[m,n]}(\mathbf{h},x,y) \ \Leftrightarrow\ \exists\pi.\ x \overset{\pi}{\rightsquigarrow} y \ \wedge\ m \leq |\pi| \leq n$
3. $\mathcal{M} \models \Delta_n^{\mathrm{WF}} \ \wedge\ \mathsf{reachable}^{<i}(\mathbf{h},D,x) \ \Leftrightarrow\ D^{\mathcal{M}} = \{\ell \mid \exists\pi.\ x \overset{\pi}{\rightsquigarrow} \ell \ \wedge |\pi| < i\}$
4. $\mathcal{M} \models \Delta_n^{\mathrm{WF}} \ \wedge\ \mathsf{path}_{[m,n]}(\mathbf{h},D,x,y) \ \Leftrightarrow\ D^{\mathcal{M}} = \{\ell \in \mathsf{dom}(\pi) \mid x \overset{\pi}{\rightsquigarrow} y \wedge m \leq |\pi| \leq n\}$

Proof. Observe that in all cases, the model $\mathcal{M}$ has exactly n locations which is ensured by the formula $\Delta_n^{\mathbf{L}}$.

1. Directly follows from Lemma 4.2.
2. Directly follows from (1) and the fact that the predicate reach enumerates over all possible lengths of paths in the interval $[m,n]$.
3. Directly follows from Lemma 4.2.
4. If there is no path, the claim holds because only the last clause of the predicate path can be satisfied and it guarantees that $D = \emptyset$. If there is a simple path of length i, the path is given as $\pi = \langle x, \mathbf{h}[x], \ldots, \mathbf{h}^i[x]\rangle$. This path satisfies i-th clause. A j-th clause with $j < i$ will not satisfy reachability condition because there is no shorter path. A j-th clause with $j > i$ will not satisfy $y \notin D$. Consequently, only the i-th clause is satisfied which sets D to be set of all locations reachable in less than i steps – i.e., exactly the set $\mathsf{dom}(\pi)$, which concludes the proof.

□

Complexity. Let n be the number of locations. The reachability predicates have the following asymptotic sizes:

- $|\mathsf{reach}^i(\mathbf{h},x,y)| = \mathcal{O}(n)$ because the size of the term $\mathbf{h}^i[x]$ can be up to n.
- $|\mathsf{reach}_{[0,n]}(\mathbf{h},x,y)| = \mathcal{O}(n^2)$ because it consists of $\mathcal{O}(n)$ appearances of $\mathsf{reach}^i(\mathbf{h},x,y)$.
- $|\mathsf{reachable}^{<i}(\mathbf{h},D,x)| = \mathcal{O}(n^2)$ because the set expression can contain up to n terms of the form $\mathbf{h}^j[x]$, each of size up to n.
- $|\mathsf{path}_{[0,n]}(\mathbf{h},D,x,y)| = \mathcal{O}(n^3)$ because it contains $\mathcal{O}(n)$ occurrences of the predicate $\mathsf{reachable}^{\leq i}(\mathbf{h},D,x)$.

In the definition of the translation, we need exactly one reach and one path predicate for each list-segment predicate. The complexity of list-segment translation is therefore $\mathcal{O}(n^3)$ which is asymptotically better than in [17] that needs $\mathcal{O}(n^4)$ space to encode list segments. On the other hand, the encoding of [17] is an instance of a more general encoding, which also works for trees. Our encoding cannot be efficiently generalised for trees because enumeration over all possible paths in branching graphs requires exponential space.

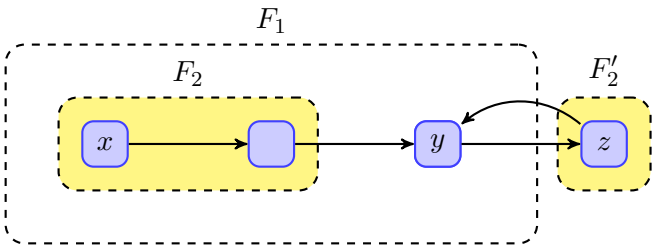


Figure 4.2: An example of a stack-heap model and footprints of sub-formulae of the formula $\varphi \triangleq \mathsf{ls}(x,z) * \big(\mathsf{ls}(x,y) \vee z \mapsto y\big)$ in this model. In particular, the sub-formula $\mathsf{ls}(x,y) \vee z \mapsto y$ has two footprints in the model – F_2 and F_2'.

4.3 Translation of Separating Conjunctions

The translation of the separating conjunction is complicated because its semantics involves a quantification over possible splits of a heap – a second order quantification over disjoint sub-heaps. If the separating conjunction does not lie under a negation, the second-order quantification can be efficiently avoided using Skolemization. Otherwise, one needs to either quantify over arrays or replace the quantification by a finite, but exponential enumeration. The former is possible because there are only finitely many arrays over the finite domain, but according to our experiments, both Z3 and CVC5 give-up on such formulae.

In [28], it was shown that for a formula φ from the fragment of boolean combinations of symbolic heaps and for a fixed model (s,h), there exists for each separating conjunction in φ only one relevant way how it can split the heap h. This allows the translation to remove quantifiers even when the separating conjunction lies under a negation – existential and universal quantification over one element domain are the same thing. This unique way to split the heap h in a model (s,h) for a formula $\psi_1 * \psi_2$ is induced by so-called *footprints* of sub-formulae ψ_1 and ψ_2 in the model (s,h). The footprint of pure atoms is the empty set, no matter what the heap is. Similarly, the footprint of a points-to assertion $x \mapsto y$ is always the singleton set $\{x\}$. In the case of a list segment $\mathsf{ls}(x,y)$, its footprint is still unique w.r.t. fixed model (s,h) – it is the domain of the simple path from x to y in $G[(s,h)]$ if such a path exists. Otherwise, we may take as the unique footprint the empty set. Intuitively, if there is no list segment, we can look at any subset of the model to conclude that there is indeed no list segment. Finally, the footprint of a separating conjunction is the union of the footprints of its operands.

The unique footprint property can be extended for conjunctions and even guarded negations, but it stops working when disjunctions appear under separating conjunctions. To demonstrate this, let us first formally define footprints.

Definition 4.2 (Footprint)**.** *Let φ be a formula and let (s,h) be a stack-heap model. A set $F \subseteq \mathsf{dom}(h)$ is called a footprint of φ in a model (s,h) if $(s, h|_F) \models \varphi$. We collect all such sets in $\mathsf{footprints}_{(s,h)}(\varphi)$.*

In other words, a footprint defines a subset of a model in which the given formula φ can be satisfied. An example is given in Figure 4.2 for the formula $\varphi \triangleq \mathsf{ls}(x,z) * \big(\mathsf{ls}(x,y) \vee z \mapsto y\big)$. The footprint of its sub-formula $\mathsf{ls}(x,z)$ is denoted by F_1. As can be seen, the footprint of the disjunction $\mathsf{ls}(x,y) \vee z \mapsto y$ is not uniquely determined as it can be satisfied in the sub-heaps induced by both F_2 and F_2'. In the case of a negation, the situation is even more

complicated as the formula $\neg\mathsf{emp}$ could be satisfied on a sub-heap induced by an arbitrary non-empty footprint $F \subseteq \mathsf{dom}(h)$.

In [28], the unique footprint of each sub-formula is axiomatized during its translation and used for translation of separating conjunctions. Although footprints are not unique in our logic, we can still use them to efficiently translate separating conjunctions by limiting their quantification to the computed footprints only. If the set of footprints is small, then the formula can be translated with only a small enumeration. Of course, in the presence of negations under separating conjunctions, the translated formula will grow exponentially.

Instead of axiomatizing footprints, we will compute them syntactically – for each sub-formula, we will compute a set of terms representing its possible footprints. Because this set is parametrised by some model (s, h), it cannot be precisely computed during the translation. Therefore, we will compute its over-approximation.

Definition 4.3. *Let φ be a formula and let (s, h) be a model. An over-approximation of the set of all possible footprints of φ in the model (s, h), denoted as $\mathsf{footprints}^{\#}_{(s,h)}(\varphi)$, is inductively defined as follow:*

- $\mathsf{footprints}^{\#}_{(s,h)}(x = y) = \mathsf{footprints}^{\#}_{(s,h)}(x \neq y) = \{\emptyset\}$
- $\mathsf{footprints}^{\#}_{(s,h)}(x \mapsto y) = \{\{s(x)\}\}$
- $\mathsf{footprints}^{\#}_{(s,h)}(\mathsf{ls}(x, y)) = \begin{cases} \{\mathsf{dom}(\pi)\} & \text{if } s(x) \overset{\pi}{\rightsquigarrow} s(y) \\ \{\emptyset\} & \text{if such } \pi \text{ does not exist} \end{cases}$
- $\mathsf{footprints}^{\#}_{(s,h)}(\neg\varphi) = 2^{\mathbf{L}}$
- $\mathsf{footprints}^{\#}_{(s,h)}(\psi_1 \wedge \psi_2) = \begin{cases} \mathsf{footprints}^{\#}_{(s,h)}(\psi_1) & \text{if } |\mathsf{footprints}^{\#}_{(s,h)}(\psi_1)| \leq |\mathsf{footprints}^{\#}_{(s,h)}(\psi_2)| \\ \mathsf{footprints}^{\#}_{(s,h)}(\psi_2) & \text{otherwise} \end{cases}$
- $\mathsf{footprints}^{\#}_{(s,h)}(\psi_1 \vee \psi_2) = \mathsf{footprints}^{\#}_{(s,h)}(\psi_1) \cup \mathsf{footprints}^{\#}_{(s,h)}(\psi_2)$
- $\mathsf{footprints}^{\#}_{(s,h)}(\psi_1 \wedge_{\neg} \psi_2) = \mathsf{footprints}^{\#}_{(s,h)}(\psi_1)$
- $\mathsf{footprints}^{\#}_{(s,h)}(\psi_1 * \psi_2) = \{F_1 \cup F_2 \mid F_1 \in \mathsf{footprints}^{\#}_{(s,h)}(\psi_1) \ \wedge \ F_2 \in \mathsf{footprints}^{\#}_{(s,h)}(\psi_2)\}$
- $\mathsf{footprints}^{\#}_{(s,h)}(\psi_1 \mathrel{-\!\circledast} \psi_2) = 2^{\mathbf{L}}$

Notice that, in the case of a formula that does not contain disjunctions, negations, and septractions, there will be no over-approximation, and the result will be a singleton set – this is an analogy of the unique footprint property from [28]. Observe that, in the case of the conjunction, the precise footprint would be the intersection of the footprints of its operands. Since we cannot evaluate needed equivalence of elements of those sets purely syntactically, we over-approximate intersection by taking its operand with lesser cardinality. In the case of the negation, we cannot compute anything more precise than all subsets of the location domain. In the case of the septraction, we could compute more precise footprints. However, because of our syntactic restriction on the fragment (septractions cannot lie under negations), we, in fact, do not need to compute footprints of septractions. The reason is the following. We need footprints only for translating separating conjunctions that lie under a negation (otherwise we can use Skolemization). Because of the mentioned restrictions,

no septraction can lie under a negated separating conjunction. We will now show that the definition indeed correctly over-approximates all possible footprints.

Lemma 4.4. *Let φ be a formula and let (s, h) be a stack-heap model. It holds that*

$$\mathsf{footprints}_{(s,h)}(\varphi) \subseteq \mathsf{footprints}^{\#}_{(s,h)}(\varphi).$$

Proof. If $(s, h) \not\models \varphi * \mathsf{true}$, then there does not exist $F \subseteq \mathsf{dom}(h)$ such that $(s, h|_F) \models \varphi$, i.e., there are no footprints of φ in (s, h) and the claim therefore trivially holds. Assume that $(s, h) \models \varphi * \mathsf{true}$, we prove the claim by the structural induction on φ.

- *Base cases.* If φ is an equality or a disequality, its only possible footprint is the empty set. Similarly, if φ is a pointer $x \mapsto y$ its only possible footprint is the singleton set $\{x\}$. Finally, if φ is a list-segment predicate $\mathsf{ls}(x, y)$ its footprint in the model (s, h) can be only the domain of the simple path from x to y. By Lemma 4.2, the path is always uniquely determined.

- *Induction steps.* If φ is either a negation or a septraction, then the claim trivially holds. Let $\varphi \triangleq \psi_1 \bowtie \psi_2$ be a binary connective other than the septraction. Let us define following short names:

$$\begin{aligned} \mathcal{F} &= \mathsf{footprints}_{(s,h)}(\varphi) & \mathcal{F}^{\#} &= \mathsf{footprints}^{\#}_{(s,h)}(\varphi) \\ \mathcal{F}_1 &= \mathsf{footprints}_{(s,h)}(\psi_1) & \mathcal{F}_1^{\#} &= \mathsf{footprints}^{\#}_{(s,h)}(\psi_1) \\ \mathcal{F}_2 &= \mathsf{footprints}_{(s,h)}(\psi_2) & \mathcal{F}_2^{\#} &= \mathsf{footprints}^{\#}_{(s,h)}(\psi_2) \end{aligned}$$

  From the induction hypothesis, we have that $\mathcal{F}_i \subseteq \mathcal{F}_i^{\#}$ for $i = 1, 2$. If φ is a conjunction, then both ψ_1 and ψ_2 need to be satisfied in (s, h). By the definition of footprint, it holds that $\mathcal{F} = \mathcal{F}_1 \cap \mathcal{F}_2$. From induction hypothesis we have that $\mathcal{F}_1 \cap \mathcal{F}_2 \subseteq \mathcal{F}_i^{\#}$ for $i = 1, 2$. Similarly, if $\varphi \triangleq \psi_1 \wedge_{\neg} \psi_2$, then only ψ_1 is satisfied in (s, h) and therefore $\mathcal{F} = \mathcal{F}_1$. Thus, $\mathcal{F} \subseteq \mathcal{F}_1^{\#}$ by induction hypothesis. If φ is a disjunction, it holds that $\mathcal{F} \subseteq \mathcal{F}_1 \cup \mathcal{F}_2 \subseteq \mathcal{F}_1^{\#} \cup \mathcal{F}_2^{\#} = \mathcal{F}^{\#}$. Finally, if φ is a separating conjunction, it can be satisfied only in a heap which is a disjoint union of sub-heaps induced by footprints $F_1 \in \mathcal{F}_1$ and $F_2 \in \mathcal{F}_2$. The set $\mathcal{F}^{\#}$ over-approximate this set by taking unions of all footprints even if they are not disjoint.

□

Finally, we can provide a simplified semantics of the separating conjunction in the way we have already sketched – instead of quantifying over all possible splits of a heap, we will quantify only over splits induced by over-approximated footprints.

Lemma 4.5. *Let $\varphi \triangleq \psi_1 * \psi_2$ be a formula and let (s, h) be a stack-heap model. Further let $\mathcal{F}_1 = \mathsf{footprints}^{\#}_{(s,h)}(\psi_1)$ and let $\mathcal{F}_2 = \mathsf{footprints}^{\#}_{(s,h)}(\psi_2)$. Then $(s, h) \models \varphi$ iff*

$$(s,h) \models \bigvee_{F_1 \in \mathcal{F}_1} \bigvee_{F_2 \in \mathcal{F}_2} (s, h|_{F_1}) \models \psi_1 \wedge (s, h|_{F_2}) \models \psi_2 \wedge h|_{F_1} \uplus^s h|_{F_2} \neq \bot \wedge F_1 \cup F_2 = \mathsf{dom}(h)$$

Proof.

($\Leftarrow$) If there exist sets F_1 and F_2 satisfying the assumption, then it holds that the heaps $h|_{F_1}$ and $h|_{F_2}$ are witnesses of the semantics of the separating conjunction.

($\Rightarrow$) Assume that $(s,h) \models \varphi$. Then there exists h_1 and h_2 such that $(s,h_1) \models \psi_1$, $(s,h_2) \models \psi_2$, $h_1 \uplus^s h_2 \neq \bot$ and $h_1 \uplus^s h_2 = h$. It clearly holds that $\mathsf{dom}(h_1)$ is a footprint of ψ_1 since $(s,h_1) \models \psi_1$ and analogically, $\mathsf{dom}(h_2)$ is a footprint of ψ_2. Therefore, we can apply Lemma 4.4 to conclude that $\mathsf{dom}(h_1) \in \mathcal{F}_1$ and $\mathsf{dom}(h_2) \in \mathcal{F}_2$. Thus, the statement we want to show holds for $F_1 = \mathsf{dom}(h_1)$ and $F_2 = \mathsf{dom}(h_2)$.

□

Later on, to define strong-disjointness of two heaps, we will also need a predicate $\mathsf{locations}(\mathbf{h}, D, L)$ which intuitively states that the set L contains locations of the heap function obtained by translation of the array $\mathbf{h}$ restricted to the set D. This predicate is defined using a predicate $\mathsf{image}(\mathbf{h}, D, I)$ which states that I is the image of the translated heap. The predicates are defined as follow:

$$\mathsf{image}(\mathbf{h}, D, I) \triangleq \bigwedge_{\ell \in \mathbf{L}} \mathbf{h}[\ell] \in I \leftrightarrow \ell \in D$$

$$\mathsf{locations}(\mathbf{h}, D, L) \triangleq \mathsf{image}(\mathbf{h}, D, I) \wedge L = D \cup I$$

Lemma 4.6. *Let $\mathcal{M}$ be a first-order model and let $(s,h) = \mathsf{T}^{-1}(\mathcal{M}, \mathbf{x})$ be a stack-heap model obtained by its inverse translation. Then the following conditions hold:*

1. $\mathcal{M} \models \Delta_n^{\mathrm{WF}} \wedge \mathsf{image}(\mathbf{h}, D, I) \Leftrightarrow I = \mathsf{img}(h)$
2. $\mathcal{M} \models \Delta_n^{\mathrm{WF}} \wedge \mathsf{locations}(\mathbf{h}, D, L) \Leftrightarrow L = \mathsf{locs}(h)$

Proof. By the definition of the model translation, the set D is always interpreted as $\mathsf{dom}(h')$. Then, both claims follows directly from the definition of $\mathsf{img}(h')$ and $\mathsf{locs}(h')$, respectively. □

4.4 Translation of Septractions

A septraction $\psi_1 \mathrel{-\!\!\circledast} \psi_2$ is satisfied by a model (s,h) if there exists a disjoint extension h_1 of the heap h such that the extension satisfies the left-hand side (i.e., $(s,h_1) \models \psi_1$) and their composition satisfies the right-hand side (i.e., $(s, h \uplus^s h_1) \models \psi_2$). Its translation is even more complicated than in the case of the separating conjunction. This is because it does not quantify over sets only but over whole heaps. We avoid this problem by restricting our fragment and forbid septractions to appear under negations (both under classical negations and in the negated branches of guarded negations). Then we can avoid the quantification using Skolemization.

There is still another complication even in this simplified fragment. It is not sufficient to use a single heap symbol when searching for a model of a septraction. As an example, let us consider the formula $\varphi \triangleq x \mapsto x * (x \mapsto \mathsf{nil} \mathrel{-\!\!\circledast} x \mapsto \mathsf{nil})$. The septraction inside the formula can be clearly satisfied at the empty heap only using the extension $h_1 = \{s(x) \mapsto s(\mathsf{nil})\}$. The whole formula can then be satisfied by a self-pointer $h = \{s(x) \mapsto s(x)\}$. Observe that $h(x)$ and $h_1(x)$ differs because x cannot be equal to nil. Therefore, we need to introduce a fresh heap for each septraction to find its model.

For the needs of our translation, we will look at the septraction from a different point of view. Instead of using a top-down approach saying that *the heap is a model if it can be extended*, we will use a bottom-up approach which says that *the heap which is a model can be obtained as a difference of a model of the right- and of a model of the left-hand*

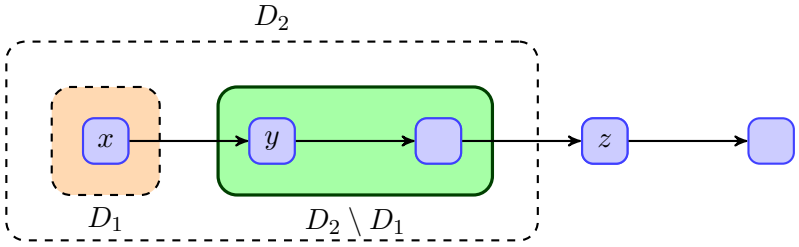


Figure 4.3: An example of a witness heap of the formula $\varphi \triangleq x \mapsto y \mathrel{-\!\circledast} \mathsf{ls}(x, z)$. The dashed boxes denote its sub-heaps induced by D_1 and D_2 satisfying the left- and right-hand sides of φ, respectively. The green solid box denotes their difference induced by $D_2 \setminus D_1$ which is a model of φ.

side. More precisely, let $\varphi \triangleq \psi_1 \mathrel{-\!\circledast} \psi_2$ be a formula. If there exist a heap h' and sets D_1 and D_2 such that $(s, h'|_{D_1}) \models \psi_1$, $(s, h'|_{D_2}) \models \psi_2$ and $D_1 \subseteq D_2$ we can construct a model of the formula φ as $(s, h'|_{D_2 \setminus D_1})$. We will call the heap h' that meets the aforementioned conditions a *witness heap* of φ.

Definition 4.4 (Witness heap)**.** *Let $\varphi \triangleq \psi_1 \mathrel{-\!\circledast} \psi_2$ and let s be a stack. Further, let h' be a heap and let $D_1, D_2 \subseteq \mathsf{dom}(h)$. We say that the heap h' is a witness heap of the septraction φ w.r.t. the stack s and sets D_1, D_2 if the following conditions hold:*

1. $D_1 \subseteq D_2$
2. $(s, h'|_{D_1}) \models \psi_1$
3. $(s, h'|_{D_2}) \models \psi_2$
4. $h'|_{D_1} \uplus^s h'|_{D_2 \setminus D_1} \neq \bot$

An example of a formula and its witness heap is given in Figure 4.3. We will now show that the existence of a witness heap is equivalent to the semantics of the septraction.

Lemma 4.7. *Let $\varphi \triangleq \psi_1 \mathrel{-\!\circledast} \psi_2$ be a formula and let (s, h) be a model. Then $(s, h) \models \varphi$ iff there exists a heap h' and sets D_1, D_2 such that h' is a witness heap of φ w.r.t. the stack s and sets D_1, D_2; it holds that $\mathsf{dom}(h) = D_2 \setminus D_1$, and $\forall \in D_2 \setminus D_1.\ h'(\ell) = h(\ell)$.*

Proof.

($\Rightarrow$) Assume that $(s, h) \models \varphi$. By the semantics of the septraction, there exists a heap h_1 such that $h \uplus^s h_1 \neq \bot$, $(s, h_1) \models \psi_1$ and $(s, h \uplus^s h_1) \models \psi_2$. Let $h' = h \uplus^s h_1$. Then h' is a witness heap of φ w.r.t. the stack s and sets $D_1 = \mathsf{dom}(h_1)$, $D_2 = \mathsf{dom}(h \uplus^s h_1)$. Moreover, it holds that $\mathsf{dom}(h) = D_2 \setminus D_1$ and for all $\ell \in D_2 \setminus D_1$, it holds that $h'(\ell) = h(\ell)$ because h' is defined using h on $D_2 \setminus D_1$.

($\Leftarrow$) Assume that h' is a witness heap of φ w.r.t. the stack s and sets D_1, D_2. Let $h_1 = h'|_{D_1}$. Then, $D_1 \subseteq D_2$, $(s, h_1) \models \psi_1$, $(s, h'|_{D_2}) \models \psi_2$ and $h_1 \uplus^s h'|_{D_2 \setminus D_1} \neq \bot$. From the assumptions that $\mathsf{dom}(h) = D_2 \setminus D_1$ and $\forall \in D_2 \setminus D_1.\ h'(\ell) = h(\ell)$, we have that $(s, h'|_{D_2}) = (s, h_1 \uplus^s h'|_{D_2 \setminus D_1}) = (s, h_1 \uplus^s h)$. Thus, $(s, h) \models \varphi$.

□

4.5 Translation to SMT

Now, we can put all the ingredients together and define the translation function $\mathsf{T}(\varphi, \mathbf{x})$ using fresh symbols:

$$\begin{aligned}\mathsf{T}(\varphi, \mathbf{x}) \triangleq\ & \textbf{let } n = \mathsf{bound}(\varphi, \mathbf{x}) \textbf{ in} \\ & \textbf{let } (\widetilde{\varphi}, \mathcal{A}, \mathcal{F}) = \mathsf{T}^{\mathbf{x}}_n(\varphi, \mathbf{h}, D) \text{ for fresh symbols } \mathbf{h} \text{ and } D \textbf{ in} \\ & \Delta^{\mathrm{WF}}_n \wedge \mathcal{A} \wedge \widetilde{\varphi}\end{aligned}$$

The definition relies on an auxiliary function $\mathsf{T}^{\mathbf{x}}_n(\varphi, \mathbf{h}, D)$ that performs the actual recursive translation. This function is called with two fixed parameters – the set of variables $\mathbf{x}$ and the location bound n; and three another parameters – a formula φ to be translated and symbols $\mathbf{h}$ and D which will be used for the encoding of its heap. Those symbols may change during the translation. For example, a translation of a septraction will use a fresh heap to translate its operands (i.e., to find its witness heap).

The function $\mathsf{T}^{\mathbf{x}}_n(\varphi, \mathbf{h}, D)$ produces a triple $(\widetilde{\varphi}, \mathcal{A}, \mathcal{F})$. The first component is called the *semantics* and it represents constraints on the stack and heap imposed by the formula φ expressed in FOL over arrays and sets. Those constraints may use auxiliary symbols introduced during the translation. The second component $\mathcal{A}$ called *axioms* defines the intended meaning of those auxiliary symbols. The reason why those components are kept separate is that while the semantics can be modified based on the boolean structure of the input formula (e.g., negated), the axioms are always collected in their positive form using conjunctions. The last component $\mathcal{F}$ is called *footprints* and it is a set of location set terms. The meaning of this component is to represent the set $\mathsf{footprints}^{\#}_{(s,h)}(\varphi)$. Observe that, in the top-level definition of the translation $\mathsf{T}(\varphi, \mathbf{x})$, $\mathcal{F}$ is not used, it is only necessary to translate separating conjunctions. In the final formula, the semantics $\widetilde{\varphi}$ and axioms $\mathcal{A}$ are joined in a conjunction together with the well-formedness constraint Δ^{WF}_n.

Translation of atomic formulae. Let φ be an atomic formula and let F be a fresh set symbol. The translation of φ is defined as $\mathsf{T}^{\mathbf{x}}_n(\varphi, \mathbf{h}, D) = (\widetilde{\varphi}, \mathcal{A}, \mathcal{F})$ where the individual components are defined as:

$x = y$:	$\widetilde{\varphi} \triangleq x = y \wedge D = \emptyset$	$\mathcal{A} \triangleq \mathsf{true}$	$\mathcal{F} \triangleq \{\emptyset\}$
$x \neq y$:	$\widetilde{\varphi} \triangleq x \neq y \wedge D = \emptyset$	$\mathcal{A} \triangleq \mathsf{true}$	$\mathcal{F} \triangleq \{\emptyset\}$
$x \mapsto y$:	$\widetilde{\varphi} \triangleq \mathbf{h}[x] = y \wedge D = \{x\}$	$\mathcal{A} \triangleq \mathsf{true}$	$\mathcal{F} \triangleq \{\{x\}\}$
$\mathsf{ls}(x, y)$:	$\widetilde{\varphi} \triangleq \mathsf{reach}_{[0,n]}(\mathbf{h}, x, y) \wedge D = F$	$\mathcal{A} \triangleq \mathsf{path}_{[0,n]}(\mathbf{h}, F, x, y)$	$\mathcal{F} \triangleq \{F\}$

The translation of atomic formulae is quite straightforward. The only interesting case is the list-segment predicate. Here, we use an axiom to ensure that the fresh symbol F is always interpreted as the domain of a simple path from x to y. This symbol is then used as the only footprint term and also as the expected domain of the list segment in the translation of semantics.

Translation of boolean connectives. Let φ be a boolean connective – either $\varphi \triangleq \neg\psi_1$ or $\varphi \triangleq \psi_1 \bowtie \psi_2$ where $\bowtie \in \{\wedge, \wedge_\neg, \vee\}$. We introduce short names for the translations of its operands

$$(\widetilde{\psi}_1, \mathcal{A}_1, \mathcal{F}_1) \triangleq \mathsf{T}^{\mathbf{x}}_n(\psi_1, \mathbf{h}, D) \qquad (\widetilde{\psi}_2, \mathcal{A}_2, \mathcal{F}_2) \triangleq \mathsf{T}^{\mathsf{x}}_n(\psi_2, \mathbf{h}, D)$$

and define the translation of φ as $\mathsf{T}^{\mathbf{x}}_n(\varphi, \mathbf{h}, D) = (\widetilde{\varphi}, \mathcal{A}, \mathcal{F})$:

$$\begin{array}{lllll}
\neg\psi_1: & \widetilde{\varphi} \triangleq \neg\widetilde{\psi}_1 & \mathcal{A} \triangleq \mathcal{A}_1 & \mathcal{F} \triangleq 2^{\mathbf{L}} \\
\psi_1 \wedge \psi_2: & \widetilde{\varphi} \triangleq \widetilde{\psi}_1 \wedge \widetilde{\psi}_2 & \mathcal{A} \triangleq \mathcal{A}_1 \wedge \mathcal{A}_2 & \mathcal{F} \triangleq \begin{cases} \mathcal{F}_1 & \text{if } |\mathcal{F}_1| \leq |\mathcal{F}_2| \\ \mathcal{F}_2 & \text{if } |\mathcal{F}_2| < |\mathcal{F}_1| \end{cases} \\
\psi_1 \wedge_\neg \psi_2: & \widetilde{\varphi} \triangleq \widetilde{\psi}_1 \wedge \neg\widetilde{\psi}_2 & \mathcal{A} \triangleq \mathcal{A}_1 \wedge \mathcal{A}_2 & \mathcal{F} \triangleq \mathcal{F}_1 \\
\psi_1 \vee \psi_2: & \widetilde{\varphi} \triangleq \widetilde{\psi}_1 \vee \widetilde{\psi}_2 & \mathcal{A} \triangleq \mathcal{A}_1 \wedge \mathcal{A}_2 & \mathcal{F} \triangleq \mathcal{F}_1 \cup \mathcal{F}_2
\end{array}$$

The translation of boolean connectives is again straightforward. The translation of the semantics directly captures the original semantics of the input formula. The axioms are always collected using conjunction and no new ones are introduced. The fooptrints directly reflect the inductive definition of the set $\mathsf{footprints}^{\#}_{(s,h)}(\varphi)$.

Observe that the operands of each boolean connective are always translated using the same array $\mathbf{h}$ and the same set D, which will be no longer true for spatial connectives discussed below.

Translation of the separating conjunction. Let $\varphi \triangleq \psi_1 * \psi_2$ and let D_1, D_2 be fresh location set symbols. We introduce short names for the translations of its operands

$$(\widetilde{\psi}_1, \mathcal{A}_1, \mathcal{F}_1) \triangleq \mathsf{T}^{\mathbf{x}}_n(\psi_1, \mathbf{h}, D_1) \qquad (\widetilde{\psi}_2, \mathcal{A}_2, \mathcal{F}_2) \triangleq \mathsf{T}^{\mathsf{x}}_n(\psi_2, \mathbf{h}, D_2)$$

and define the translation of φ as $\mathsf{T}^{\mathbf{x}}_n(\varphi, \mathbf{h}, D) = (\widetilde{\varphi}, \mathcal{A}, \mathcal{F})$:

$$\widetilde{\varphi} \triangleq \bigvee_{F_1 \in \mathcal{F}_1} \bigvee_{F_2 \in \mathcal{F}_2} \widetilde{\psi}_1[F_1/D_1] \wedge \widetilde{\psi}_2[F_2/D_2] \wedge F_1 \cap F_2 = \emptyset \wedge L^{F_1} \cap L^{F_2} \subseteq \mathbf{x} \wedge D = F_1 \cup F_2$$

$$\mathcal{A} \triangleq \mathcal{A}_1 \wedge \mathcal{A}_2 \wedge \bigwedge_{F_1 \in \mathcal{F}_1} \mathsf{locations}(L^{F_1}, F_1, \mathbf{h}) \wedge \bigwedge_{F_2 \in \mathcal{F}_2} \mathsf{locations}(L^{F_2}, F_2, \mathbf{h})$$

$$\mathcal{F} \triangleq \{F_1 \cup F_2 \mid F_1 \in \mathcal{F}_1, F_2 \in \mathcal{F}_2\}$$

Here, we use fresh symbols D_1 and D_2 to represent a split of the heap h. We enumerate over all possible splits using a disjunction over pairs of fooptrints from the set $\mathcal{F}_1 \times \mathcal{F}_2$. Each clause of this enumeration is created by substituting D_i in the translation of the semantics by the footprint F_i, and adding additional requirements that footprints F_1 and F_2 are strongly-disjoint and their union yields D. To express strong-disjointness, we introduce a fresh symbol L^{F_i} for each footprint F_i and add an axiom that ensures that L^{F_i} will be interpreted as the set of locations of the heap represented by $\mathbf{h}$ and F_i.

If φ does not lie under a negation, we can use Skolemization to translate its semantics without any enumeration using fresh symbols L_1 and L_2 to represent heap locations:

$$\widetilde{\varphi} \triangleq \widetilde{\psi}_1 \wedge \widetilde{\psi}_2 \wedge D_1 \cap D_2 = \emptyset \wedge L_1 \cap L_2 \subseteq \mathbf{x} \wedge D = D_1 \cup D_2$$

$$\mathcal{A} \triangleq \mathcal{A}_1 \wedge \mathcal{A}_2 \wedge \mathsf{locations}(L_1, D_1, \mathbf{h}) \wedge \mathsf{locations}(L_2, D_2, \mathbf{h})$$

Translation of the septraction. For septractions, we can always use Skolemization because they never lie under negations. This is ensured by the definition of our fragment. Let $\varphi \triangleq \psi_1 \mathrel{-\!\!\circledast} \psi_2$, let D_1 and D_2 be fresh set symbol, and let $\mathbf{h}'$ be a fresh array symbol. Further, let L_1 and L_2 be fresh set symbols used to represent heap locations. We introduce short names for the translations of its operands

$$(\widetilde{\psi}_1, \mathcal{A}_1, \mathcal{F}_1) \triangleq \mathsf{T}^{\mathbf{x}}_n(\psi_1, \mathbf{h}', D_1) \qquad (\widetilde{\psi}_2, \mathcal{A}_2, \mathcal{F}_2) \triangleq \mathsf{T}^{\mathbf{x}}_n(\psi_2, \mathbf{h}', D_2)$$

and define the translation of φ as $\mathsf{T}^{\mathbf{x}}_n(\varphi, \mathbf{h}, D) = (\widetilde{\varphi}, \mathcal{A}, \mathcal{F})$:

$$\begin{aligned}
\widetilde{\varphi} &\triangleq \widetilde{\psi}_1 \wedge \widetilde{\psi}_2 \wedge D_1 \subseteq D_2 \wedge L_1 \cap L_2 \subseteq \mathbf{x} \wedge \bigwedge_{\ell \in D} \mathbf{h}[\ell] = \mathbf{h}'[\ell] \wedge D = D_2 \setminus D_1 \\
\mathcal{A} &\triangleq \mathcal{A}_1 \wedge \mathcal{A}_2 \wedge \mathsf{locations}(L_1, D_1, \mathbf{h}') \wedge \mathsf{locations}(L_2, D, \mathbf{h}') \\
\mathcal{F} &\triangleq 2^{\mathbf{L}}
\end{aligned}$$

Observe that the definition of $\widetilde{\varphi}$ is based on the definition of a witness heap (Definition 4.4).

Complexity. The time complexity of the translation is dominated by computing the sets of footprints of possibly exponential size w.r.t. the number of locations and therefore also w.r.t. the number of variables. The size of the translated formula can also be up to exponential because of the enumeration caused by translation of separating conjunctions. In the worst case, our decision procedure runs in NEXP because the decision procedure for the used theory runs in NP.

We will now define a fragment SSL^E that we can translate more effectively, i.e., without footprint enumeration and obtain a translated formula of at most polynomial size. Let us first consider some straight-forward optimisations. Observe that the footprints are needed only for translating separating conjunctions. Therefore, we do not need to compute them if we are not under separating conjunction, or if we are just under separating conjunctions that can be translated using Skolemization. In order to do this, the translation function simply has additional flags used to determine whether it can perform Skolemization and whether it should compute footprints or not. We will now define the SSL^E fragment.

Definition 4.5 (SSL^E fragment)**.** *An SSL formula φ is in SSL^E iff at least one of the following conditions holds.*

- *φ does not contain negations, disjunctions, and septraction under separating conjunctions.*
- *φ does not contain spatial connectives under negations.*

Lemma 4.8. *Let $\varphi \in SSL^E$. Then $\mathsf{T}(\varphi, \mathbf{x})$ has a polynomial size w.r.t. $|\varphi| + |\mathbf{x}|$. The decision procedure runs in NP for SSL^E.*

Proof. We will show that we do not need to enumerate over footprints when translating φ. By case distinction over definition of the SSL^E fragment:

- If φ does not contain negations, disjunctions, and septractions under separating conjunctions, then the set $\mathsf{footprints}^{\#}_{(s,h)}(\psi)$ always has at most one element for each subformula ψ of φ, or ψ lies in the part of formula where $\mathsf{footprints}^{\#}_{(s,h)}(\psi)$ will not be needed and the translation will therefore not compute it.

- If φ does not contain any spatial connectives under a negation, we can translate all of them using Skolemization.

To finish the proof, observe that the size of the translated formula is now dominated by the translation of list-segment predicates which is polynomial w.r.t. the number of locations. Since the number of locations is linear w.r.t. the size of the formula φ, we have that the translated formula $\mathsf{T}(\varphi, \mathbf{x})$ has at most polynomial size. The whole decision procedure then runs in NP. □

Observe that SSL^E subsumes the positive fragment as defined in [25] but not the positive fragment as defined in [24] (and also in this work) where one can also use guarded negations in positive formulae. Whether formulae with arbitrary appearance of guarded negations can be effectively translated or not remains an open question for the future work.

Example 4.1. To demonstrate the translation on a simple formula, let us consider the entailment $x \mapsto y * y \mapsto z \models \mathsf{ls}(x, z)$ that can be reduced to unsatisfiability of the formula $\varphi \triangleq (x \mapsto y * y \mapsto z) \wedge_{\neg} \mathsf{ls}(x, z)$. Notice that the entailment does not hold because its left-hand side can be satisfied by a cycle that is not a list segment. All components of the translation are shown in Figure 4.4. While the location bound is $\mathsf{bound}(\varphi, \{x, y, z\}) = 2 \cdot |\{x, y, z\}| + 1 = 7$, the translation uses the optimal bound 3. This optimal bound can be computed based on the structure of φ as shown in Section 5.2.

In the top-left corner, the figure shows the AST of φ and assigns a unique identifier to each of its sub-formulae. Those identifiers are used to index components of the translation. The most interesting case among the translations of the semantics is the separating conjunction ❸. Because the separating conjunction does not lie under a negation, it can be translated using Skolemization. The translation creates two fresh set symbols D_1 and D_2 which are used to translate operands ❶ and ❷, respectively. Moreover, we do not have to add the constraint that locations shared by sub-heaps induced by D_1 and D_2 are covered by $\mathbf{x}$ because φ is positive. The only interesting axiom is created for the list-segment predicate ❹ and it defines F to be the path from x to z on the heap represented by $\mathbf{h}$.

The definition of the reachability predicates uses another optimisation. While the location bound is 3, it also counts with nil that cannot be allocated. Therefore, the maximum bound for reachability can be set to 2. Observe that only the second clause of the path predicate representing the empty simple path is satisfied in the model in the top-right corner. Therefore it holds that $F = \emptyset$. Because D has to be equal to $\{x, y\}$, it holds that $D \neq F$, and, consequently, $\widetilde{\varphi}_4$ is not satisfied. Then, $\widetilde{\varphi}_5$ is satisfied.

The figure also shows how footprints would be computed. However, they are not needed in this case because the only separating conjunction is translated using Skolemization.

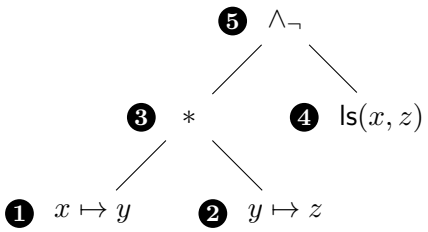
The translated formula can be satisfied by the following first-order model $\mathcal{M}$ over the domain $\mathbf{L} = \{0, 1, 2\}$:

$$x^{\mathcal{M}} = 1, y^{\mathcal{M}} = 2, z^{\mathcal{M}} = 1, \mathsf{nil}^{\mathcal{M}} = 0$$
$$\mathbf{h}^{\mathcal{M}} = \mathsf{K}(0)\langle 1 \lhd 2\rangle\langle 2 \lhd 1\rangle$$
$$D^{\mathcal{M}} = \{1, 2\}$$

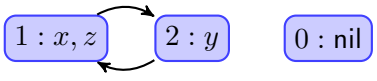
The model will also interpret other components (D_1, D_2, F) but those are not needed to construct a stack-heap $(s, h) = \mathsf{T}^{-1}(\mathcal{M}, \{x, y, z, \mathsf{nil}\})$ of the input SSL formula:

$$s = \{x \mapsto 1, y \mapsto 2, z \mapsto 1, \mathsf{nil} \mapsto 0\}$$
$$h = \{1 \mapsto 2, 2 \mapsto 1\}$$

Input formula:



Possible stack-heap model:



Semantics:

$$\widetilde{\varphi}_1 \triangleq \mathbf{h}[x] = y \wedge D_1 = \{x\}$$
$$\widetilde{\varphi}_2 \triangleq \mathbf{h}[y] = z \wedge D_2 = \{y\}$$
$$\widetilde{\varphi}_3 \triangleq \widetilde{\varphi}_1 \wedge \widetilde{\varphi}_2 \wedge D_1 \cap D_2 = \emptyset \wedge D = D_1 \cup D_2$$
$$\widetilde{\varphi}_4 \triangleq \mathsf{reach}_{[0,2]}(\mathbf{h}, x, z) \wedge D = F$$
$$\widetilde{\varphi}_5 \triangleq \widetilde{\varphi}_3 \wedge \neg\widetilde{\varphi}_4$$

Axioms:

$$\mathcal{A}_1 \triangleq \mathsf{true}$$
$$\mathcal{A}_2 \triangleq \mathsf{true}$$
$$\mathcal{A}_3 \triangleq \mathcal{A}_1 \wedge \mathcal{A}_2$$
$$\mathcal{A}_4 \triangleq \mathsf{path}_{[0,2]}(\mathbf{h}, F, x, z)$$
$$\mathcal{A}_5 \triangleq \mathcal{A}_3 \wedge \mathcal{A}_4$$

Auxiliary predicates:

$$\mathsf{reach}_{[0,2]}(\mathbf{h}, x, z) \triangleq x = z \ \vee\ \mathbf{h}[x] = z \ \vee\ \mathbf{h}^2[x] = z$$

$$\begin{aligned}\mathsf{path}_{[0,2]}(\mathbf{h}, F, x, z) \triangleq\ & \big(\neg\mathsf{reach}_{[0,2]}(\mathbf{h}, x, z) \ \wedge\ F = \emptyset\big) && \text{(no path)}\\ \vee\ & \big(x = z \ \wedge\ F = \emptyset \ \wedge\ z \notin F\big) && \text{(simple path of length 0)}\\ \vee\ & \big(\mathbf{h}[x] = z \ \wedge\ F = \{x\} \ \wedge\ z \notin F\big) && \text{(simple path of length 1)}\\ \vee\ & \big(\mathbf{h}^2[x] = z \ \wedge\ F = \{x, \mathbf{h}[x]\} \ \wedge\ z \notin F\big) && \text{(simple path of length 2)}\end{aligned}$$

Translated formula:

$$\mathsf{T}(\varphi, \{x, y, z\}) \triangleq \Delta_3^{\mathrm{WF}} \ \wedge\ \mathcal{A}_5 \ \wedge\ \widetilde{\varphi}_5$$

Footprints (only for illustration):

$$\mathcal{F}_1 \triangleq \{\{x\}\} \qquad \mathcal{F}_2 \triangleq \{\{y\}\} \qquad \mathcal{F}_3 \triangleq \{\{x, y\}\}$$
$$\mathcal{F}_4 \triangleq \{F\} \qquad \mathcal{F}_5 \triangleq \{\{x, y\}\}$$

Figure 4.4: An example of the translation for the formula $\varphi \triangleq (x \mapsto y * y \mapsto z) \wedge_\neg \mathsf{ls}(x, z)$. The translation uses the optimal location bound $n = 3$. Each component is indexed by the id of its corresponding sub-formula. Those ids are assigned in the AST of φ in the top-left corner. The bottom part shows how footprints would be computed. This is just for illustration because the only separating conjunction is translated using Skolemization – by introducing fresh symbols D_1 and D_2, which are implicitly existentially quantified.

4.6 Proof of the Correctness

This section is devoted to the proof of the correctness of the proposed translation. Its correctness is summarised by the following theorem.

Theorem 4.1 (Translation correctness)**.** *An SSL formula φ is satisfiable over variables $\mathbf{x}$ iff its translation $\mathsf{T}(\varphi, \mathbf{x})$ is satisfiable. Moreover, if $\mathcal{M} \models \mathsf{T}(\varphi, \mathbf{x})$, then $\mathsf{T}^{-1}(\mathcal{M}, \mathbf{x}) \models \varphi$.*

In other words, the theorem states that the input and its translation are equisatisfiable. Moreover, the inverse translation of a first-order model always yields a stack-heap model of the original formula. The high-level idea of the proof is the following. We first establish a correspondence between stack-heap models and first-order models, and then show that an input formula is satisfied by some stack-heap model (s, h) iff its translation is satisfied by a first-order model $\mathcal{M}$ that corresponds to (s, h). To prove this for spatial connectives, we will have to define an operation of composition of two models and prove that it mimics the strongly-disjoint union of two heaps. To finish the proof, we will also show that $\mathsf{T}^{-1}(\mathcal{M}, \mathbf{x})$ corresponds to $\mathcal{M}$.

In the remainder of this chapter, we fix an SSL formula φ, a set of variables $\mathbf{x}$, and their location bound $n = \mathsf{bound}(\varphi, \mathbf{x})$.

4.6.1 SMT Models

In this section, we introduce several notations related to first-order models. We first define a *model of SMT encoding* (*SMT model* for short) – a model that satisfies the top-level constraints given by the formula Δ_n^{WF}.

Definition 4.6 (SMT model)**.** *Let $\mathcal{M}$ be a first-order model. We say that $\mathcal{M}$ is a model of SMT encoding (SMT model for short) w.r.t. φ and $\mathbf{x}$ if $\mathcal{M} \models \Delta_n^{\mathrm{WF}}$.*

In particular, all SMT models w.r.t. fixed φ and $\mathbf{x}$ have the same domain $\mathbf{L}$ of cardinality n defined by the formula Δ_n^{WF}. We will now formalise the correspondence of stack-heap and SMT models.

Definition 4.7 (Corresponding models)**.** *Let (s, h) be a stack-heap model and let $\mathcal{M}$ be an SMT model. The model $\mathcal{M}$ corresponds to (s, h), written as $\mathcal{M} \sim (s, h)$, if the following conditions hold:*

1. $\mathsf{dom}(s) = \mathbf{x}$,
2. $\forall x \in \mathbf{x}.\ s(x) = x^{\mathcal{M}}$,
3. $\mathsf{dom}(h) = D^{\mathcal{M}}$,
4. $\forall \ell \in D^{\mathcal{M}}.\ h(\ell) = \mathbf{h}[\ell]^{\mathcal{M}}$.

Lemma 4.9. *Let $\mathcal{M}$ be an SMT model. There exists the unique stack-heap model such that $\mathcal{M} \sim (s, h)$. Moreover, it holds that $(s, h) = \mathsf{T}^{-1}(\mathcal{M}, \mathbf{x})$.*

Proof. The uniqueness of (s, h) follows from the fact that each of its components is uniquely determined in the definition of the correspondence. Directly from Definition 4.1, we have that $\mathcal{M} \sim \mathsf{T}^{-1}(\mathcal{M}, \mathbf{x})$. Thus, $(s, h) = \mathsf{T}^{-1}(\mathcal{M}, \mathbf{x})$. □

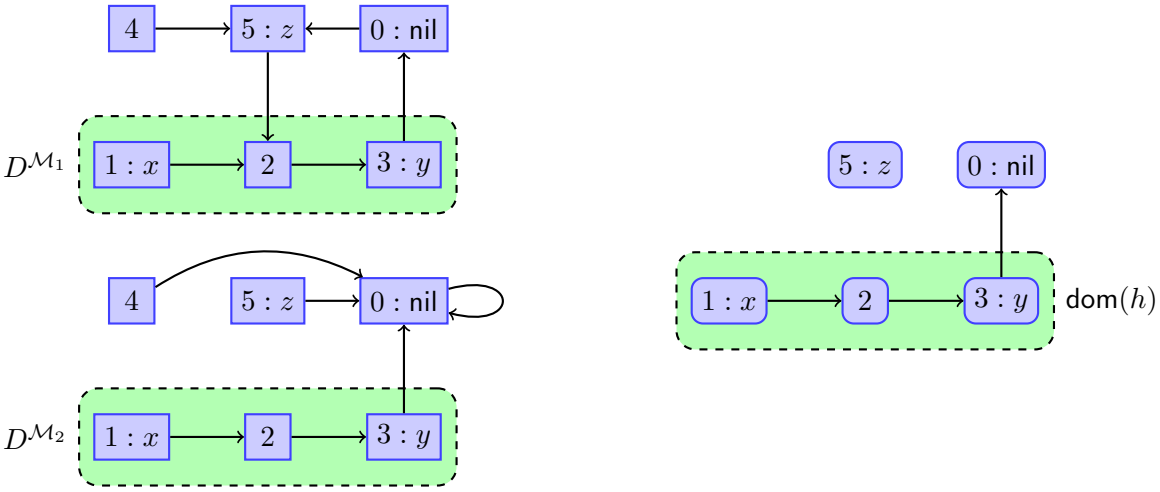


Figure 4.5: An example of SMT models $\mathcal{M}_1$ and $\mathcal{M}_2$ and a stack-heap model (s, h) that corresponds to both of them.

The converse of the previous lemma does not hold because for a stack-heap model (s, h), we have multiple corresponding SMT models – two models may differ in their interpretation of the array $\mathbf{h}$ outside of their common interpretation of the heap domain D. This situation is demonstrated in Figure 4.5 that depicts a graphic representation of SMT models $\mathcal{M}_1$ and $\mathcal{M}_2$ (on the left-hand side) that both correspond to the same stack-heap model (on the right-hand side). Based on this observation, we define an equivalence relation such that $\mathcal{M}_1$ will be equivalent with $\mathcal{M}_2$.

Definition 4.8 (Equivalent SMT models)**.** *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be SMT models. Model $\mathcal{M}_1$ is equivalent with $\mathcal{M}_2$, denoted as $\mathcal{M}_1 \equiv \mathcal{M}_2$, if the following conditions hold:*

1. $\forall x \in \mathbf{x}.\ x^{\mathcal{M}_1} = x^{\mathcal{M}_2}$,
2. $D^{\mathcal{M}_1} = D^{\mathcal{M}_2}$,
3. $\forall \ell \in D^{\mathcal{M}_1}.\ \mathbf{h}^{\mathcal{M}_1}[\ell] = \mathbf{h}^{\mathcal{M}_2}[\ell]$.

Lemma 4.10. *Relation $\equiv$ on SMT models is an equivalence relation. Moreover, it holds that $\mathcal{M}_1 \equiv \mathcal{M}_2$ iff for all stack-heap models (s, h), it holds that $\mathcal{M}_1 \sim (s, h) \Leftrightarrow \mathcal{M}_2 \sim (s, h)$.*

Proof. Both claims follow directly from the definition of equivalent models and from the definition of model correspondence. □

We would like to further work with equivalence classes of $\equiv$ on SMT models. In order to do this, we need to ensure that formulae created during the translation cannot distinguish equivalent models. In other words, this means that all formulae respect our encoding of partial functions. For example, the formula $\widetilde{\varphi} \triangleq D = \emptyset \wedge \mathbf{h}[x] = y$ does not respect this encoding because it constraints value of the partial function represented by $\mathbf{h}$ outside of its domain D. We will call formulae that respect this property *well-defined*.

Definition 4.9 (Well-defined formula)**.** *Let $\widetilde{\varphi}$ be a first-order formula. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be SMT models such that $\mathcal{M}_1 \equiv \mathcal{M}_2$. Formula $\widetilde{\varphi}$ is well-defined if $\mathcal{M}_1 \models \widetilde{\varphi} \Leftrightarrow \mathcal{M}_2 \models \widetilde{\varphi}$.*

4.6.2 Composition of SMT Models

We will now define when two SMT models are *compatible* w.r.t. the set of variables $\mathbf{x}$. Further, we will define a *composition* of compatible SMT models. Intuitively, this operation will mimic the operator $\uplus^s$ in the domain of SMT models. We will later need lemmas about properties of composition to prove the correctness for the cases of spatial connectives.

To define compatibility of two models, we define the *image* of the array $\mathbf{h}$ w.r.t. some set $X \subseteq \mathsf{dom}(\mathbf{h})$, as $\mathsf{arr\_img}(\mathbf{h}, X) = \{y \mid \exists x \in X.\, \mathbf{h}[x] = y\}$.

Definition 4.10 (Compatible models)**.** *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two SMT models. Further, let $I_i = \mathsf{arr\_img}(\mathbf{h}^{\mathcal{M}_i}, D^{\mathcal{M}_i})$ for $i \in \{1, 2\}$. SMT models $\mathcal{M}_1$ and $\mathcal{M}_2$ are $\mathbf{x}$-compatible if the following conditions hold:*

1. $\forall x \in \mathbf{x}.\ x^{\mathcal{M}_1} = x^{\mathcal{M}_2}$
2. $D^{\mathcal{M}_1} \cap D^{\mathcal{M}_2} = \emptyset$
3. $(D^{\mathcal{M}_1} \cup I_1) \cap (D^{\mathcal{M}_2} \cup I_2) \subseteq \mathbf{x}^{\mathcal{M}_1}$

Intuitively, two models are compatible if (1) they interpret the stack in the same way, (2) their interpretation of heap domains are disjoint, and (3) all locations common in their interpretations of heaps are among interpretations of variables. The next step is to define how compatible models can be composed.

Definition 4.11 (Model composition)**.** *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be SMT models. Their composition $\mathcal{M}_1 \oplus^{\mathbf{x}} \mathcal{M}_2$ is defined as $\langle \mathbf{L}^{\mathcal{M}_1}, \langle x^{\mathcal{M}_1} \rangle_{x \in \mathbf{x}}, \mathbf{h}^{\mathcal{M}_1} \boxplus \mathbf{h}^{\mathcal{M}_2}, D^{\mathcal{M}_1} \cup D^{\mathcal{M}_2} \rangle$ if $\mathcal{M}_1$ and $\mathcal{M}_2$ are $\mathbf{x}$-compatible and undefined otherwise. The composition of arrays, $\boxplus$, is defined as:*

$$\mathbf{h}^{\mathcal{M}_1} \boxplus \mathbf{h}^{\mathcal{M}_2} = \begin{cases} \mathbf{h}[\ell]^{\mathcal{M}_1} & \text{if } \ell \in D^{\mathcal{M}_1}, \\ \mathbf{h}[\ell]^{\mathcal{M}_2} & \text{if } \ell \in D^{\mathcal{M}_2}, \\ \mathsf{nil}^{\mathcal{M}_1} & \text{otherwise.} \end{cases}$$

The composition has the same domain $\mathbf{L}$ as both of its operands (this is ensured by the fact that both operands are SMT models w.r.t. the same fixed φ and $\mathbf{x}$). The composition also interprets all variables in the same way as its operands because the operands are compatible. The composition of arrays $\boxplus$ mimics disjoint union of two partial functions. Notice that $\boxplus$ is well-defined because heap domains $D^{\mathcal{M}_1}$ and $D^{\mathcal{M}_2}$ of compatible models are disjoint. The following lemma shows that the model composition precisely captures the strongly-disjoint union of two heaps.

Lemma 4.11. *Let $(s, h_1) \sim \mathcal{M}_1$ and $(s, h_2) \sim \mathcal{M}_2$ be two pairs of corresponding models. Then the following properties hold:*

1. $h_1 \uplus^s h_2 = \bot \ \Leftrightarrow\ \mathcal{M}_1 \oplus^{\mathbf{x}} \mathcal{M}_2 = \bot$
2. $(s, h_1 \uplus^s h_2) \sim \mathcal{M}_1 \oplus^{\mathbf{x}} \mathcal{M}_2$

Proof.

1. Because $\mathcal{M}_1$ and $\mathcal{M}_2$ correspond to stack-heap models with the same stack, we know that each symbol $x \in \mathbf{x}$ is interpreted in the same way in both models. Consequently, their composition can be undefined iff at least one of conditions (2) or (3) from the definition of compatibility is not satisfied. If the condition (2) is not satisfied, then $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) \neq \emptyset$ and vice versa. If the condition (3) is not satisfied, then $\mathsf{locs}(h_1) \cap \mathsf{locs}(h_2) \not\subseteq s(\mathbf{x})$ and vice versa.

2. Directly follows from (1) and the definition of the composition.

□

We will now prove two key lemmas that we will later need to prove the correctness of the translation of spatial connectives.

Lemma 4.12 (Extension by a compatible model)**.** *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be $\mathbf{x}$-compatible SMT models. Let ψ be a well-defined formula s.t. $D \notin \mathsf{vars}(\psi)$. Then $\mathcal{M}_1 \models \psi$ iff $\mathcal{M}_1 \oplus^{\mathbf{x}} \mathcal{M}_2 \models \psi$.*

Proof. Let $\mathcal{M}' = \mathcal{M}_1 \oplus^{\mathbf{x}} \mathcal{M}_2$. We will show that the interpretations of all terms in the formula ψ are the same in both models $\mathcal{M}_1$ and $\mathcal{M}'$. Then also all predicates and subformulae of ψ have the same boolean values in both models and consequently ψ is either satisfied in both models, or falsified in both models. We have to consider two sorts of terms:

(a) *Location terms.* Each location term t is of the form $\mathbf{h}^i[x_j]$ where $i \in \mathbb{N}$ and $x_j \in \mathbf{x}$ is a location variable. We show the statement by the induction over i. If $i = 0$, then t is a location variable x_j which is, by the definition of the compatibility, interpreted in the same way in both $\mathcal{M}_1$ and $\mathcal{M}_2$ and consequently also in their composition. Let $t = \mathbf{h}^{i+1}[x_j]$, and let $t' = \mathbf{h}^i[x_j]$. By the induction hypothesis, the term t' is interpreted as the same location ℓ in both models. Let us consider following cases for ℓ:

- If $\ell \in D_1^{\mathcal{M}_1}$, then the interpretation of $\mathbf{h}[\ell]$ in $\mathcal{M}_1$ is the same as in $\mathcal{M}'$ by the definition of the composition.
- If $\ell \notin D_1^{\mathcal{M}_1} \wedge \ell \notin D^{\mathcal{M}_2}$, then $\mathbf{h}[\ell]$ may be interpreted differently in those models, but there exist $\mathcal{M}_1' \equiv \mathcal{M}_1$ that interprets $\mathbf{h}[\ell]$ in the same way as the model $\mathcal{M}_2$. It holds that $\mathcal{M}_1 \oplus^{\mathbf{x}} \mathcal{M}_2 \equiv \mathcal{M}_1' \oplus^{\mathbf{x}} \mathcal{M}_2$.
- If $\ell \notin D^{\mathcal{M}_1} \wedge \ell \in D^{\mathcal{M}_2}$, then we can replace $\mathcal{M}_1$ by its equivalent model $\mathcal{M}_1'$ that interprets $\mathbf{h}[\ell]$ in the same way as $\mathcal{M}_1 \oplus \mathcal{M}_2$. Again, $\mathcal{M}_1 \oplus^{\mathbf{x}} \mathcal{M}_2 \equiv \mathcal{M}_1' \oplus^{\mathbf{x}} \mathcal{M}_2$.

(b) *Location set terms.* Let t be a location term such that t. We know that ψ does not contain the symbol D. The term t is either a constant, i.e., a possibly empty enumeration of locations, or an application of a set operation to a tuple of set terms. We will prove the statement by induction over the structure of t. If t is an enumeration of constants, then the statement holds, because all of its elements (location terms) are interpreted in the same way by (a). The induction step is trivial because set operations will yield the same result in both models.

□

Lemma 4.13. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be $\mathbf{x}$-compatible SMT models. Let ψ_1 and ψ_2 be well-defined formulae such that $D \notin \psi_1$ and $D \notin \psi_2$. Then the following statements are equivalent:*

1. $\mathcal{M}_1 \models \psi_1 \wedge \mathcal{M}_2 \models \psi_2$
2. $\mathcal{M}_1 \oplus^{\mathbf{x}} \mathcal{M}_2 \models \psi_1 \wedge \psi_2$

Proof. By Lemma 4.12 we have

$$\mathcal{M}_1 \models \psi_1 \;\Leftrightarrow\; \mathcal{M}_1 \oplus^{\mathbf{x}} \mathcal{M}_2 \models \psi_1,$$
$$\mathcal{M}_2 \models \psi_2 \;\Leftrightarrow\; \mathcal{M}_2 \oplus^{\mathbf{x}} \mathcal{M}_1 \models \psi_2.$$

The claim then follows from the commutativity of the composition. □

4.6.3 Translation Invariants

To prove Theorem 4.1, we will show that the recursive translation function $\mathsf{T}^{\mathbf{x}}_n(\psi, \mathbf{h}, D)$ satisfies several invariants. Let $\mathcal{M}$ be an SMT model and let (s, h) be its corresponding stack-heap model. Let ψ be a sub-formula of φ, for its translation $(\widetilde{\psi}, \mathcal{A}, \mathcal{F}) = \mathsf{T}^{\mathbf{x}}_n(\psi, \mathbf{h}, D)$, the following statements hold:

(I1) **Well-definedness**. Formula $\widetilde{\psi}$ is well-defined according to Definition 4.9.

(I2) **Skolemization**. If ψ does not lie under a negation or in a branch negated by a guarded negation in φ, then $\widetilde{\psi}$ does not lie under a negation or an universal quantifier in $\widetilde{\varphi}$.

(I3) **Consistency of the axioms**. The axioms $\mathcal{A}$ and the well-formedness constraint Δ^{WF}_n are consistent, i.e., there exists a model $\mathcal{M}'$ such that $\mathcal{M}' \models \mathcal{A} \wedge \Delta^{\mathrm{WF}}_n$. This invariant ensures that top-level constraints created by the translation of the formula ψ are always satisfiable.

(I4) **Correctness of the footprints**. The set $\mathcal{F}$ over-approximates the set of all possible footprints of ψ in (s, h). More precisely, we will show that $\mathcal{F}^{\mathcal{M}} = \mathsf{footprints}^{\#}_{(s,h)}(\psi)$.

(I5) **Correctness of the translation**. The translation of the formula ψ is correct. More precisely, it holds that $(s, h) \models \psi$ iff $\mathcal{M} \models \widetilde{\psi}$.

The first invariant ensures the well-definedness of all formulae that is needed to prove other invariants. The second invariant guarantees that the translation will not introduce any negation or universal quantifier over an existentially quantified symbol, for which there was no negation over this symbol in the original SSL formula. Consequently, we can perform the Skolemization and replace it by a constant symbol. The third invariant merely requires that there is no inconsistency in auxiliary definitions introduced during the translation. The fourth invariant makes sure that set $\mathcal{F}$ correctly captures all footprints. Finally, the last invariant states the correctness of the translation. Theorem 4.1 follows almost directly from the last invariant applied to the whole input formula φ.

Lemma 4.14 (Invariant I1)**.** *Let ψ be an SSL formula and let $(\widetilde{\psi}, \mathcal{A}, \mathcal{F}) = \mathsf{T}^{\mathbf{x}}_n(\psi, \mathbf{h}, D)$ be its translation. The formula $\widetilde{\psi}$ is well-defined.*

Proof. Let $\mathcal{M} \models \widetilde{\psi}$ and let $\ell \notin D^{\mathcal{M}}$. Let $\mathcal{M}'$ be an SMT model that interprets all terms except $\mathbf{h}[\ell]$ as $\mathcal{M}$. In order to show that $\widetilde{\psi}$ is well-defined, we need to show that $\mathcal{M}' \models \widetilde{\psi}$. We will proceed by induction on the structure of the original SSL formulae ψ.

If ψ is a pure atom, than the claim holds because $\widetilde{\psi}$ does not restrict the mapping of $\mathbf{h}$ at all. If $\psi \triangleq x \mapsto y$, then $\widetilde{\psi}$ does not restrict the mapping of $\mathbf{h}$ for locations other than $x^{\mathcal{M}}$ which is in $D^{\mathcal{M}}$. If $\psi \triangleq \mathsf{ls}(x, y)$, then set D is interpreted using the predicate path $\mathsf{path}(\mathbf{h}, D, x, y)$. As shown in the proof of Lemma 4.3, exactly one clause of this predicate is satisfied in $\mathcal{M}$. If the i-th clause is satisfied, than $\widetilde{\psi}$ restricts only locations $\mathbf{h}^j[\ell]^{\mathcal{M}}$ such that $j < i$. Since all such locations are in $D^{\mathcal{M}}$ by Lemma 4.3, $\widetilde{\psi}$ is well-defined.

The claim directly follows from the inductive hypothesis for all boolean connectives and also for the separating conjunction because they do not impose any additional restrictions on the array $\mathbf{h}$. Finally, if ψ is a septraction, then it restricts $\mathbf{h}$ only at locations in $D^{\mathcal{M}}$ by the definition of its translation which asserts that $\bigwedge_{\ell \in D} \mathbf{h}[\ell] = \mathbf{h}'[\ell]$. Thus, $\widetilde{\psi}$ is well-defined. □

Lemma 4.15 (Invariant I2)**.** *Let ψ be an SSL formula and let $(\widetilde{\psi}, \mathcal{A}, \mathcal{F}) = \mathsf{T}^{\mathbf{x}}_n(\psi, \mathbf{h}, D)$ be its translation. If ψ does not lie under a negation or in a branch negated by a guarded negation in φ, then $\widetilde{\psi}$ does not lie under a negation or an universal quantifier in $\widetilde{\varphi}$.*

Proof. It can be easily verified that the translation never introduces universal quantifiers and uses only those negations that were already present in the formula φ. □

Lemma 4.16 (Invariant I3)**.** *Let ψ be an SSL formula and let $(\widetilde{\psi}, \mathcal{A}, \mathcal{F}) = \mathsf{T}^{\mathbf{x}}_n(\psi, \mathbf{h}, D)$ be its translation. The formula $\mathcal{A} \wedge \Delta^{\mathrm{WF}}_n$ is satisfiable.*

Proof. The formula Δ^{WF}_n is always satisfiable by a model $\mathcal{M}$ with domain $\mathbf{L} = \{\ell_1, ..., \ell_n\}$ and such that $\mathsf{nil} \notin D$. Each path axiom is satisfiable in isolation by Lemma 4.3. Similarly, each $\mathsf{location}$ axiom is satisfiable in isolation by Lemma 4.6. All those axioms are combined using conjunctions in all cases of the translation. From the definition of the translation, it follows that each axiom speaks about a fresh symbol. Consequently, the conjunction of satisfiable axioms is also satisfiable. □

Lemma 4.17 (Invariant I4)**.** *Let ψ be an SSL formula and let $(\widetilde{\psi}, \mathcal{A}, \mathcal{F}) = \mathsf{T}^{\mathbf{x}}_n(\psi, \mathbf{h}, D)$ be its translation. Then $\mathcal{F}^{\mathcal{M}} = \mathsf{footprints}^{\#}_{(s,h)}(\psi)$.*

Proof. By induction on the structure of ψ. The case of the list-segment predicate is ensured by Lemma 4.3. Other cases are trivial because their definitions of the set $\mathcal{F}$ directly copy the inductive definition of the set $\mathsf{footprints}^{\#}_{(s,h)}(\psi)$. □

Lemma 4.18 (Invariant I5)**.** *Let ψ be a formula and let $(\widetilde{\psi}, \mathcal{A}, \mathcal{F}) = \mathsf{T}^{\mathbf{x}}_n(\psi, \mathbf{h}, D)$ be its translation. Then $(s, h) \models \psi$ iff $\mathcal{M} \models \widetilde{\psi}$.*

Proof. By structural induction on ψ.

Atomic formulae.

- $\psi \triangleq x = y$:

$$\begin{aligned}
(s,h) \models x = y &\Leftrightarrow s(x) = s(y) \wedge \mathsf{dom}(h) = \emptyset && \text{(SSL semantics)}\\
&\Leftrightarrow x^{\mathcal{M}} = y^{\mathcal{M}} \wedge D^{\mathcal{M}} = \emptyset && \text{(model correspondence)}\\
&\Leftrightarrow \mathcal{M} \models x = y \wedge D = \emptyset && \text{(FOL semantics)}\\
&\Leftrightarrow \mathcal{M} \models \widetilde{\psi} && \text{(translation)}
\end{aligned}$$

- $\psi \triangleq x \neq y$:

$$\begin{aligned}
(s,h) \models x \neq y &\Leftrightarrow s(x) \neq s(y) \wedge \mathsf{dom}(h) = \emptyset && \text{(SSL semantics)}\\
&\Leftrightarrow x^{\mathcal{M}} \neq y^{\mathcal{M}} \wedge D^{\mathcal{M}} = \emptyset && \text{(model correspondence)}\\
&\Leftrightarrow \mathcal{M} \models x \neq y \wedge D = \emptyset && \text{(FOL semantics)}\\
&\Leftrightarrow \mathcal{M} \models \widetilde{\psi} && \text{(translation)}
\end{aligned}$$

- $\psi \triangleq x \mapsto y$:

$$\begin{aligned}
(s,h) \models x \mapsto y &\Leftrightarrow h(s(x)) = y \wedge \mathsf{dom}(h) = \{s(x)\} && \text{(SSL semantics)}\\
&\Leftrightarrow \mathbf{h}[x]^{\mathcal{M}} = y^{\mathcal{M}} \wedge D^{\mathcal{M}} = \{x^{\mathcal{M}}\} && \text{(model correspondence)}\\
&\Leftrightarrow \mathcal{M} \models \mathbf{h}[x] = y \wedge D = \{x\} && \text{(FOL semantics)}\\
&\Leftrightarrow \mathcal{M} \models \widetilde{\psi} && \text{(translation)}
\end{aligned}$$

- $\psi \triangleq \mathsf{ls}(x,y)$:

$$\begin{aligned}
(s,h) \models \mathsf{ls}(x,y) &\Leftrightarrow \exists \pi.\ s(x) \overset{\pi}{\rightsquigarrow} s(y) \wedge \mathsf{dom}(h) = \mathsf{dom}(\pi) && \text{(Lemma 4.1)}\\
&\Leftrightarrow \mathcal{M} \models \mathsf{reach}_{[0,n]}(\mathbf{h}, x, y) \wedge \mathsf{path}_{[0,n]}(\mathbf{h}, D, x, y) && \text{(Lemma 4.3)}\\
&\Leftrightarrow \mathcal{M} \models \widetilde{\psi} && \text{(translation)}
\end{aligned}$$

Inductive steps for boolean connectives. Let ψ be either a negation $\psi \triangleq \neg\psi_1$ or a binary formula $\psi \triangleq \psi_1 \bowtie \psi_2$ where $\bowtie \in \{\wedge, \wedge_\neg, \vee\}$. We introduce short names for the results of translation of an operand ψ_i:

$$(\widetilde{\psi}_1, \mathcal{A}_1, \mathcal{F}_1) \triangleq \mathsf{T}^{\mathsf{x}}_n(\psi_1, \mathbf{h}, D) \qquad (\widetilde{\psi}_2, \mathcal{A}_2, \mathcal{F}_2) \triangleq \mathsf{T}^{\mathsf{x}}_n(\psi_2, \mathbf{h}, D)$$

- $\psi \triangleq \neg\psi_1$:

$$\begin{aligned}
(s,h) \models \neg\psi &\Leftrightarrow (s,h) \not\models \psi_1 && \text{(SSL semantics)}\\
&\Leftrightarrow \mathcal{M} \not\models \widetilde{\psi}_1 && \text{(induction hypothesis)}\\
&\Leftrightarrow \mathcal{M} \models \neg\widetilde{\psi} && \text{(translation)}
\end{aligned}$$

- $\psi \triangleq \psi_1 \wedge \psi_2$:

$$\begin{aligned}
(s,h) \models \psi_1 \wedge \psi_2 &\Leftrightarrow (s,h) \models \psi_1 \wedge (s,h) \models \psi_2 && \text{(SSL semantics)}\\
&\Leftrightarrow \mathcal{M} \models \widetilde{\psi}_1 \wedge \mathcal{M} \models \widetilde{\psi}_2 && \text{(induction hypothesis)}\\
&\Leftrightarrow \mathcal{M} \models \widetilde{\psi} && \text{(translation)}
\end{aligned}$$

- $\psi \triangleq \psi_1 \wedge_\neg \psi_2$:

$$\begin{aligned}
(s,h) \models \psi_1 \wedge_\neg \psi_2 &\Leftrightarrow (s,h) \models \psi_1 \wedge (s,h) \not\models \psi_2 && \text{(SSL semantics)}\\
&\Leftrightarrow \mathcal{M} \models \widetilde{\psi}_1 \wedge \mathcal{M} \not\models \widetilde{\psi}_2 && \text{(induction hypothesis)}\\
&\Leftrightarrow \mathcal{M} \models \widetilde{\psi} && \text{(translation)}
\end{aligned}$$

- $\psi \triangleq \psi_1 \vee \psi_2$:

$$\begin{aligned}
(s,h) \models \psi_1 \vee \psi_2 &\Leftrightarrow (s,h) \models \psi_1 \vee (s,h) \models \psi_2 && \text{(SSL semantics)}\\
&\Leftrightarrow \mathcal{M} \models \widetilde{\psi}_1 \vee \mathcal{M} \models \widetilde{\psi}_2 && \text{(induction hypothesis)}\\
&\Leftrightarrow \mathcal{M} \models \widetilde{\psi} && \text{(translation)}
\end{aligned}$$

Inductive step for the separating conjunction. Let $\psi \triangleq \psi_1 * \psi_2$. We introduce short names for the results of the translation of its operands using fresh set symbols D_1 and D_2:

$$(\widetilde{\psi}_1, \mathcal{A}_1, \mathcal{F}_1) \triangleq \mathsf{T}^{\mathbf{x}}_n(\psi_1, \mathbf{h}, D_1) \qquad (\widetilde{\psi}_2, \mathcal{A}_2, \mathcal{F}_2) \triangleq \mathsf{T}^{\mathsf{x}}_n(\psi_2, \mathbf{h}, D_2)$$

Let $(s, h) \models \psi$, by the definition of SSL semantics this is equivalent to:

$$\exists h_1, h_2.\ (s, h_1) \models \psi_1 \wedge (s, h_2) \models \psi_2 \wedge h_1 \uplus^s h_2 \neq \bot \wedge h_1 \uplus^s h_2 = h.$$

From the invariant I3, we have that $\mathcal{F}_i = \mathsf{footprints}^{\#}_{(s,h)}(\psi_i)$ for $i = 1, 2$. Then we can apply Lemma 4.5 to obtain an equivalent statement:

$$\bigvee_{F_1 \in \mathcal{F}_1} \bigvee_{F_2 \in \mathcal{F}_2} (s, h|_{F_1}) \models \psi_1 \wedge (s, h|_{F_2}) \models \psi_2 \wedge h|_{F_1} \uplus^s h|_{F_2} \neq \bot \wedge F_1 \cup F_2 = \mathsf{dom}(h).$$

After applying induction hypotheses for ψ_1 and ψ_2, Lemma 4.11, and the definition of the model correspondence, we obtain equivalent claim:

$$\bigvee_{F_1 \in \mathcal{F}_1} \bigvee_{F_2 \in \mathcal{F}_2} (\mathcal{M}_1 \models \widetilde{\psi}_1) \wedge (\mathcal{M}_2 \models \widetilde{\psi}_2) \wedge \mathcal{M}_1 \oplus^{\mathbf{x}} \mathcal{M}_2 \neq \bot \wedge F_1 \cup F_2 = D.$$

Formulae $\widetilde{\psi}_1$ and $\widetilde{\psi}_2$ are by invariant I1 well-defined. They also do not contain the symbol D because they were translated using fresh symbols D_1 and D_2, respectively. Therefore, we can apply Lemma 4.13 to obtain an equivalent formulation:

$$\bigvee_{F_1 \in \mathcal{F}_1} \bigvee_{F_2 \in \mathcal{F}_2} (\mathcal{M}_1 \oplus^{\mathbf{x}} \mathcal{M}_2 \models \widetilde{\psi}_1 \wedge \widetilde{\psi}_2) \wedge \mathcal{M}_1 \oplus^{\mathbf{x}} \mathcal{M}_2 \neq \bot \wedge F_1 \cup F_2 = D.$$

From Lemma 4.6 and the definition of model compatibility, this is an equivalent formulation for:

$$\bigvee_{F_1 \in \mathcal{F}_1} \bigvee_{F_2 \in \mathcal{F}_2} (\mathcal{M}_1 \oplus^{\mathbf{x}} \mathcal{M}_2 \models \widetilde{\psi}_1 \wedge \widetilde{\psi}_2 \wedge D_1 \cap D_2 = \emptyset \wedge L^{D_1} \cap L^{D_2} \subseteq \mathbf{x}) \wedge F_1 \cup F_2 = D.$$

Finally, this is equivalent to:

$$\mathcal{M}_1 \oplus^{\mathbf{x}} \mathcal{M}_2 \models \bigvee_{F_1 \in \mathcal{F}_1} \bigvee_{F_2 \in \mathcal{F}_2} \Big(\widetilde{\psi}_1\ [F_1/D_1] \wedge \widetilde{\psi}_2\ [F_2/D_2] \wedge F_1 \cap F_2 = \emptyset \\ \wedge L^{F_1} \cap L^{F_2} \subseteq \mathbf{x} \wedge F_1 \cup F_2 = D \Big),$$

which is, from the definition of the translation, equivalent to $\mathcal{M}_1 \oplus^{\mathbf{x}} \mathcal{M}_2 \models \widetilde{\psi}$. The case of the translation using Skolemization is proved analogically using invariant I2 to show that Skolemization can be indeed used.

Inductive step for the septraction. Let $\psi \triangleq \psi_1 \mathbin{-\!\circledast} \psi_2$. We introduce short names for the results of the translation of its operands using fresh symbols $\mathbf{h}'$, D_1 and D_2:

$$(\widetilde{\psi}_1, \mathcal{A}_1, \mathcal{F}_1) \triangleq \mathsf{T}^{\mathbf{x}}_n(\psi_1, \mathbf{h}', D_1) \qquad (\widetilde{\psi}_2, \mathcal{A}_2, \mathcal{F}_2) \triangleq \mathsf{T}^{\mathsf{x}}_n(\psi_2, \mathbf{h}', D_2)$$

Let $(s, h) \models \psi$. By Lemma 4.7, this is equivalent to the existence of a witness heap h' w.r.t. the stack s and sets D_1, D_2 such that $\mathsf{dom}(h) = D_2 \setminus D_1$ and $\forall \ell \in \mathsf{dom}(h).h(\ell) = h'(\ell)$. By the definition of the fragment, ψ cannot lie under a negation. Using the invariant I2, we can perform Skolemization to remove existential quantifiers:

$$\begin{aligned}&(s, h_1|_{D_1}) \models \psi_1 \wedge (s, h_1|_{D_2}) \models \psi_2 \wedge D_1 \subseteq D_2 \wedge \mathsf{dom}(h) = D_2 \setminus D_1 \\ &\wedge h_1|D_1 \uplus^s h'|_{D_2 \setminus D_1} \neq \bot \wedge \forall \ell \in \mathsf{dom}(h).h(\ell) = h'(\ell).\end{aligned}$$

After applying the induction hypotheses and using model correspondence, we obtain:

$$\begin{aligned}&(\mathcal{M}_1 \models \widetilde{\psi_1}) \wedge (\mathcal{M}_1 \oplus^x \mathcal{M}_2 \models \widetilde{\psi}_2) \wedge D_1 \subseteq D_2 \wedge D = D_2 \setminus D_1 \\ &\wedge \mathcal{M}_1 \oplus^x \mathcal{M}_2 \neq \bot \wedge \forall \ell \in D.\mathbf{h}[\ell] = \mathbf{h}'[\ell].\end{aligned}$$

From Lemma 4.6 and the definition of model compatibility, we can rewrite this as:

$$\begin{aligned}&(\mathcal{M}_1 \models \widetilde{\psi_1}) \wedge (\mathcal{M}_1 \oplus^x \mathcal{M}_2 \models \widetilde{\psi}_2) \wedge D_1 \subseteq D_2 \wedge D = D_2 \setminus D_1 \\ &\wedge L_1 \cap L_2 \subseteq \mathbf{x} \wedge \forall \ell \in D.\mathbf{h}[\ell] = \mathbf{h}'[\ell].\end{aligned}$$

Finally, formulae $\widetilde{\psi}_1$ is by invariant I1 well-defined. It also do not contain the symbol D because it was translated using fresh symbol D_1. Therefore, we can apply Lemma 4.12 to obtain an equivalent formulation:

$$\mathcal{M}_1 \oplus^x \mathcal{M}_2 \models \widetilde{\psi}_1 \wedge \widetilde{\psi}_2 \wedge D_1 \subseteq D_2 \wedge D = D_2 \setminus D_1 \wedge L_1 \cap L_2 \subseteq \mathbf{x} \wedge \forall \ell \in D.\mathbf{h}[\ell] = \mathbf{h}'[\ell],$$

which is, from the definition of the translation, equivalent to $\mathcal{M}_1 \oplus^x \mathcal{M}_2 \models \widetilde{\psi}$. This concludes the proof.

□

Finally, we can prove Theorem 4.1 as a corollary of invariants and previously proved lemmas.

Proof of theorem 4.1. From invariants I3 and I5, we have that the SSL formula φ is satisfiable over variables $\mathbf{x}$ iff $\mathsf{T}(\varphi, \mathbf{x})$ is satisfiable. Moreover, from Lemma 4.9, we have that $(s, h) = \mathsf{T}^{-1}(\mathcal{M}, \mathbf{x})$. □

Chapter 5

Optimisations

In this chapter, we describe several original optimisations of the decision procedure proposed in Section 4. In the first part, we focus on proving general tighter bounds for symbolic heaps. Besides bounds on the number of locations in a model, we will also introduce an idea of list-length bounds that will allow us to decrease the size of the encoding of list-segment predicates. Then, we will show how tighter bounds can be computed for general formulae based on their structure. A simple method for computing tighter locations bounds was sketched already in [18], but we propose a more detailed and precise approach. Moreover, [18] does not consider bounds on lengths of list segments at all.

5.1 Tighter Bounds for Symbolic Heaps

Recall that, according to our definition, a formula φ is a symbolic heap if it is of the form $\varphi \triangleq * \psi_i$ where all ψ_i are atomic formulae. While the symbolic heap fragment is one of the most simplest forms of separation logic, it is frequently used in verification tools. It therefore makes sense to propose optimisations for its encoding even though it is just a small subset of SSL. In previously proposed approaches based on a small-model property, optimised bounds for symbolic heaps were not considered [17, 18].

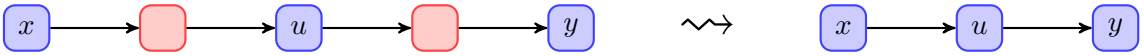
First, we will show that each satisfiable symbolic heap has a model where all locations are named – this improves the location bound to $|\mathsf{vars}(\varphi)|$ for this fragment. To prove this, we will use a reduction of sub-heaps similar to the reduction of chunks from the proofs of small-model properties in Theorem 3.4 and Theorem 3.5.

Lemma 5.1. *Let φ be a symbolic heap and let $(s, h) \models \varphi$ be its model. Then there exists a heap h' such that $(s, h') \models \varphi$ and h' does not contain any anonymous locations, i.e., $\mathsf{locs}(h') \subseteq \mathsf{img}(s)$.*

Proof. We will show how a heap h' can be constructed from the heap h. Let $\varphi \triangleq *_{1 \leq i \leq n} \psi_i$. By the semantics of SSL and the fact that each symbolic heap is a positive formula, we can decompose the heap h into disjoint sub-heaps $h_1, \ldots, h_n$ such that $h = h_1 + \cdots + h_n$ and, for all $1 \leq i \leq n$, it holds that $(s, h_i) \models \psi_i$. We reduce each sub-heap h_i to a sub-heap h'_i by removing all anonymous locations. Formally, we set $\mathsf{dom}(h'_i) = \{\ell \in \mathsf{dom}(h_i) \mid \ell \in \mathsf{img}(s)\}$ and define its mapping as:

$$h'_i(\ell) = h_i^k(\ell) \text{ where } k > 0 \text{ is the minimal natural number such that } h_i^k(\ell) \in \mathsf{img}(s).$$

The reduced sub-heaps are well-defined because the original heap is either empty or a sequence of pointers with a named sink. The named sink guarantees that some number k such that $h_i^k(\ell) \in \mathsf{img}(s)$ always exists for each locations $\ell \in \mathsf{dom}(h'_i)$. Graphically, the reduction of a non-empty sub-heap can be visualised as follow:



Now, we need to show that the reduction preserves satisfiability, i.e., $(s, h'_i) \models \psi_i$ for each i. If ψ_i is a pure atom or a points-to assertion, this trivially holds since $h_i = h'_i$. If $\varphi_i \triangleq \mathsf{ls}(x, y)$, then the sub-heap is modified but remains a sequence of pointers from x to y. It also holds that $h' = h'_1 + \cdots + h'_n \neq \bot$ because the reduction can only remove locations. Thus, $(s, h') \models \varphi$.

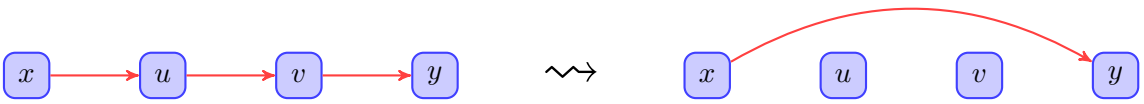
Further, we have that $\mathsf{dom}(h') \subseteq \mathsf{img}(s)$ because, by the definition of the reduction, $\mathsf{dom}(h'_i) \subseteq \mathsf{img}(s)$ for each i. From Lemma 3.2, it follows that all dangling locations are also named, and therefore $\mathsf{locs}(h') \subseteq \mathsf{img}(s)$. □

Our experiments show that decreasing of the location bounds is not always enough to efficiently solve some formulae. We will therefore also compute a *list-length bound* for each predicate $\mathsf{ls}(x, y)$ that occurs in the input formula. The list-length bound is an interval $[m, n]$ such that it is enough to consider paths π such that $m \leq |\pi| \leq n$ only when translating the list-segment predicate. In the translation, the interval is used to parameterise the predicates reach and path used to express the semantics of the given list-segment predicate.

We will now show, that for a symbolic heap φ, it is always sufficient to use the list-length bound $[0, 1]$ for all list-segment predicates in φ. In other words, if φ is satisfiable, we can find a model where each list segment is either empty or a single pointer.

Lemma 5.2. *Let φ be a symbolic heap and let $(s, h) \models \varphi$ be its model. Then there exists h' such that $(s, h') \models \varphi$, and, for each predicate $\psi \triangleq \mathsf{ls}(x, y) \in \mathsf{subformulae}(\varphi)$, it holds that the predicate ψ is satisfied in a sub-heap of size at most one, i.e., it holds that either $s(x) = s(y)$ or $h'(s(x)) = s(y)$.*

Proof. Again, we will show how to construct a heap h' from the heap h. Let $\varphi \triangleq *_{1 \leq i \leq n} \psi_i$. By the semantics of SSL, the heap h can be decomposed into disjoint sub-heaps $h_1, \ldots, h_n$ such that $h = h_1 + \cdots + h_n$, and, for all $1 \leq i \leq n$, it holds that $(s, h_i) \models \psi_i$. Using Lemma 5.1, we can safely assume that all h_i does not contain any anonymous locations. It holds that h_i is either an empty heap, a single pointer, or an acyclic sequence of pointers with a uniquely determined source x and sink y. In the third case, we reduce it to a heap $h'_i = \{x \mapsto y\}$. Graphically, this can be visualised as:



Since the reduction can only decrease domains, we have that $h' = h'_1 + \cdots + h'_n \neq \bot$. We will further show that $(s, h'_i) \models \psi_i$ for each i. The only nontrivial case is $\psi_i \triangleq \mathsf{ls}(x, y)$ because sub-heaps of pure atoms and points-to assertions are not modified. If $s(x) = s(y)$, then $h_i = \emptyset$ and consequently $h_i = h'_i$. If $s(x) \neq s(y)$, then $h'_i(s(x)) = s(y)$ and $\mathsf{dom}(h') = \{x\}$. Thus, $(s, h'_i) \models \psi_i$ for all i, and consequently $(s, h') \models \varphi$. Moreover, for each $\mathsf{ls}(x, y) \in \mathsf{subformulae}(\varphi)$ it holds that either $s(x) = s(y)$ or $h'(s(x)) = s(y)$ by the definition of the reduction. □

Using the previous lemma, we can encode the list-segment predicate $\mathsf{ls}(x, y)$ occurring in a symbolic heap in constant space as:

$$\mathsf{T}_n^x(\mathsf{ls}(x, y), h, D) \;\triangleq\; (x = y \wedge D = \emptyset) \;\vee\; (x \neq y \wedge h[x] = y \wedge D = \{x\})$$

Consequently, if φ is a symbolic heap, its translation $\mathsf{T}(\varphi, \mathbf{x})$ has a linear size.

5.2 Tighter Bounds for General Formulae

In this section, we will describe our original approach for computing more precise location and list-length bounds based on *SL-graphs*. SL-graphs were already used in [11] to design a polynomial decision procedure for symbolic heaps, but we will use them in a slightly different context. For simplicity, we will focus on formulae which do not contain septractions.

Definition 5.1 (SL graph)**.** *Let* $\mathbf{x}$ *be a set of variables. SL-graph over* $\mathbf{x}$ *is a tuple* $G = (\mathbf{x}, \circledcirc_{\mapsto}, \circledcirc_{\rightsquigarrow}, \circledcirc_{=}, \circledcirc_{\neq})$ *where*

- $\circledcirc_{\mapsto} \subseteq \mathbf{x} \times \mathbf{x}$ *defines directed points-to edges,*
- $\circledcirc_{\rightsquigarrow} \subseteq \mathbf{x} \times \mathbf{x}$ *defines directed list-segment edges,*
- $\circledcirc_{=} \subseteq \{\{x, y\} \mid x, y \in \mathbf{x}\}$ *defines undirected equality edges,*
- $\circledcirc_{\neq} \subseteq \{\{x, y\} \mid x, y \in \mathbf{x}\}$ *defines undirected disequality edges.*

Individual relations of G are must-equalities ($\circledcirc_{=}$), must-disequalities ($\circledcirc_{\neq}$), must-pointers ($\circledcirc_{\mapsto}$), and must-list segments ($\circledcirc_{\rightsquigarrow}$). Intuitively, they represent atomic relations between variables, that hold in all models of some formula φ. We also define the set of variables that must be allocated in some formula φ:

$$\mathsf{alloc}(G) = \{x \in \mathbf{x} \mid \exists y \in \mathbf{x}.\; x \circledcirc_{\mapsto} y \vee (x \circledcirc_{\neq} y \wedge x \circledcirc_{\rightsquigarrow} y)\}$$

In other words, variable x is allocated if it is either a source of some must-pointer, or a source of some non-empty must-list segment.

To compute SL-graph of a formula φ w.r.t. variables $\mathbf{x}$, denoted as $G_{\mathbf{x}}[\varphi]$, we define several auxiliary functions:

$$G_1 \sqcap G_2 = \big(\mathbf{x}, \circledcirc_{\mapsto G_1} \cap \circledcirc_{\mapsto G_2}, \circledcirc_{\rightsquigarrow G_1} \cap \circledcirc_{\rightsquigarrow G_2}, (\circledcirc_{= G_1} \cap \circledcirc_{= G_2})^*, \circledcirc_{\neq G_1} \cap \circledcirc_{\neq G_2}\big)$$
$$G_1 \sqcup G_2 = \big(\mathbf{x}, \circledcirc_{\mapsto G_1} \cup \circledcirc_{\mapsto G_2}, \circledcirc_{\rightsquigarrow G_1} \cup \circledcirc_{\rightsquigarrow G_2}, (\circledcirc_{= G_1} \cup \circledcirc_{= G_2})^*, \circledcirc_{\neq G_1} \cup \circledcirc_{\neq G_2}\big)$$
$$G_1 \uplus G_2 = \big(\mathbf{x}, \circledcirc_{\mapsto G_1} \cup \circledcirc_{\mapsto G_2}, \circledcirc_{\rightsquigarrow G_1} \cup \circledcirc_{\rightsquigarrow G_2}, (\circledcirc_{= G_1} \cup \circledcirc_{= G_2})^*, \circledcirc_{\neq G_1} \cup \circledcirc_{\neq G_2} \cup (\mathsf{alloc}(G_1) \times \mathsf{alloc}(G_2))\big)$$

The first two operations perform the intersection and the union of all edges, respectively. The disjoint union of two SL-graphs, $G_1 \uplus G_2$, additionally adds pairs of variables allocated in both models to must-disequalities. Observe that we always take the reflexive and transitive closure of muse-equalities to achieve that $\circledcirc_{=}$ is the equivalence relation.

Definition 5.2 (SL-graph of a formula)**.** *Let φ be a formula and let $\mathbf{x}$ be a set of variables. An SL-graph $G_{\mathbf{x}}[\varphi]$ of φ over $\mathbf{x}$ is defined inductively on the structure of the formula φ as follow:*

- $G_{\mathbf{x}}[x = y] = (\mathbf{x}, \emptyset, \emptyset, \{x \mathrel{\ominus\!\!\!-} y\}, \emptyset)$
- $G_{\mathbf{x}}[x \neq y] = (\mathbf{x}, \emptyset, \emptyset, \emptyset, \{x \mathrel{\not\ominus} y\})$
- $G_{\mathbf{x}}[x \mapsto y] = (\mathbf{x}, \{x \mathrel{\ominus\!\!\!\rightarrow} y\}, \emptyset, \emptyset, \emptyset)$
- $G_{\mathbf{x}}[\mathsf{ls}(x, y)] = (\mathbf{x}, \emptyset, \{x \mathrel{\ominus\!\!\!\rightsquigarrow} y\}, \emptyset, \emptyset)$
- $G_{\mathbf{x}}[\neg\varphi] = (\mathbf{x}, \emptyset, \emptyset, \emptyset, \emptyset)$
- $G_{\mathbf{x}}[\varphi_1 \wedge \psi_2] = G[\varphi_1] \sqcup G[\varphi_2]$
- $G_{\mathbf{x}}[\varphi_1 \wedge_{\neg} \varphi_2] = G[\varphi_1]$
- $G_{\mathbf{x}}[\varphi_1 \vee \varphi_2] = G[\varphi_1] \sqcap G[\varphi_2]$
- $G_{\mathbf{x}}[\varphi_1 * \varphi_2] = G[\varphi_1] \uplus G[\varphi_2]$

Lemma 5.3. *Let φ be an SSL formula Then the following correctness conditions for must-predicates hold:*

- *If $x \mathrel{\ominus\!\!\!-} y$, then $\forall (s, h) \in [\![\varphi]\!]_{\mathbf{x}}.\ (s, h) \models x = y * \mathsf{true}$*
- *If $x \mathrel{\not\ominus} y$, then $\forall (s, h) \in [\![\varphi]\!]_{\mathbf{x}}.\ (s, h) \models x \neq y * \mathsf{true}$*
- *If $x \mathrel{\ominus\!\!\!\rightarrow} y$, then $\forall (s, h) \in [\![\varphi]\!]_{\mathbf{x}}.\ (s, h) \models x \mapsto y * \mathsf{true}$*
- *If $x \mathrel{\ominus\!\!\!\rightsquigarrow} y$, then $\forall (s, h) \in [\![\varphi]\!]_{\mathbf{x}}.\ (s, h) \models \mathsf{ls}(x, y) * \mathsf{true}$*
- $\forall (s, h) \in [\![\varphi]\!]_{\mathbf{x}}.\ \mathsf{alloc}(G) \subseteq \mathsf{dom}(h)$

Proof (sketch). In all the cases, the computation of $G_{\mathbf{x}}[\varphi]$ propagates must-relations from atoms based on the boolean structure of the formula φ. Only in the case of the negation, it sets all must-relations to be empty, based on the semantics of negation in SSL. In the case of the separating conjunction $\psi_1 * \psi_2$, all must-relations of ψ_i for $i = 1, 2$ must also hold in $\psi_1 * \psi_2$. If x must be allocated in ψ_1 and y must be allocated in ψ_2, then $x \neq y$ in all models of $\psi_1 * \psi_2$. □

Let $\mathbf{x}/_{\ominus\!\!\!-}$ be the partition of variables induced by the must-equality relation. We define the number of must-pointers p as $p = |\{x \in \mathbf{x}/_{\ominus\!\!\!-} \mid \exists y \in \mathbf{x}/_{\ominus\!\!\!-}.\ x \mathrel{\ominus\!\!\!\rightarrow} y\}|$. Now we are ready to define the location bound of a formula more precisely. Recall that the proofs of small model properties (Theorem 3.4, Theorem 3.5) assumed the worst-case when all variables are distinct, and that each variable is allocated and gives rise to a chunk of size two. Based on must-equalities and must-pointers, we can relax those assumptions – we do not have to take into account those variables that are surely equivalent to others, and for each must-pointer, we can decrease the bound by one because we know that it will induce a chunk of size exactly one:

$$\mathsf{bound}'(\varphi, \mathbf{x}) = \begin{cases} 2 \cdot |\mathsf{vars}^{+}(\varphi)/_{\ominus\!\!\!-}| - p + 1 & \text{if } \varphi \text{ is positive} \\ 2 \cdot |(\mathbf{x} \setminus \{\mathsf{nil}\})/_{\ominus\!\!\!-}| + \lceil \varphi \rceil - p + 1 & \text{otherwise} \end{cases}$$

Lemma 5.4. *Let φ be a satisfiable formula and let $\mathsf{vars}(\varphi) \subseteq \mathsf{x}$ be set of variables. Then there exists model (s', h') such that $(s', h') \models \varphi$ and $|\mathsf{locs}(h')| \leq \mathsf{bound}'(\varphi, \mathbf{x})$.*

Proof. The proof is analogical to the proofs of Theorem 3.4 and 3.5 with two exceptions. First, there are at most $|\mathsf{vars}^+(\varphi)/_{\circledminus}|$ allocated variables for positive formula, and at most $|(\mathbf{x} \setminus \{\mathsf{nil}\})/_{\circledminus}|$ allocated variables for general formulae. Second, there are at least p chunks which consist of a single pointer and will therefore need just a single location in the worst-case. □

The second use case of SL-graphs is to compute more precise list-length bounds. Let $G = (\mathbf{x}, \circledast, \circledcirc, \circledminus, \circledcirc\!\!\!\!/)$ be an SL-graph of φ over $\mathbf{x}$ and let $G^{\mapsto} = (\mathbf{x}, \circledast)$. Further, let the location bound $n = \mathsf{bound}'(\varphi, \mathbf{x})$.

$$\mathsf{ls\_bound}(\mathsf{ls}(x, y)) = \begin{cases} [0, 0] & \text{if } x \circledminus y, \\ [0, 1] & \text{otherwise if } x \circledast y, \\ [k, l] & \text{otherwise if there exists a simple path } x \overset{\pi}{\rightsquigarrow} y \text{ in } G^{\mapsto} \text{ such} \\ & \text{that } |\pi| = l \text{ and } \pi' \text{ is the maximal prefix of } \pi \text{ of length} \\ & |\pi'| = k \text{ such that for all } v_1, v_2 \in \pi'.\ v_1 \circledcirc\!\!\!\!/ v_2, \\ [0, m] & \text{otherwise if } \mathsf{ls}(x, y) \text{ has the positive polarity in } \varphi \\ & \text{and } m = n - |\mathsf{alloc}(G)| + 1, \\ [0, n] & \text{otherwise.} \end{cases}$$

We will now describe the individual cases in detail:

- If $x \circledminus y$, then the list segment clearly must be empty.
- If $x \circledast y$, then the list segment is either a single pointer, or it is empty if $x = y$.
- The third case is a generalisation of the second case. If there exists a sequence of must-pointers from x to y of length l, then the maximal length of the list segment is l. The minimal length is determined using the maximal prefix of this sequence such that all of its elements are guaranteed to be distinct.
- The fourth case uses the fact that the list segment cannot allocate variables that are allocated by some other sub-formulae. This is, however, applicable only if the predicate $\mathsf{ls}(x, y)$ has a positive polarity, i.e., we know that it must be satisfied. Then we can subtract the number of the surely allocated variables except one. This is because a one of must-allocated variable may origin from them list segment $\mathsf{ls}(x, y)$ itself.

Chapter 6

Implementation

This chapter describes our implementation of the proposed decision procedure in a new solver called Astral (Automation for strong-separation logic). The implementation of the translation and bound computation straightforwardly follows their mathematical definitions. We therefore focus on the implementation of the solver's front-end and the SMT back-end.

6.1 Architecture

Astral is written in the OCaml programming language and it is publicly available[1] under the MIT license. The OCaml language was chosen because it offers a trade-off between performance and high-level abstraction. We divide the architecture of the solver into three parts: (1) the front-end deals with input parsing and preprocessing, (2) the decision procedure implements the translation and its optimisations, and (3) the SMT back-end handles communication with SMT solvers.

6.2 Front-end

The solver accepts as an input a formula in the format specified by the SL-COMP competition [16]. The format extends the SMT-LIB v2 format by adding commands for declaring the type of the heap and for specification of inductive predicates. Currently, Astral does not support arbitrary typing of heaps. Instead, it requires locations to be defined as an uninterpreted sort with the fixed name `Loc` and the heap to be declared with the sort $\texttt{Loc} \to \texttt{Loc}$. To parse the input, we use a generic parser for logical languages implemented in the `Dolmen`[2] library.

An example of the input can be seen in Listing 6.1. The example also shows how we deal with the fact that satisfiability is parametrised by a set of variables $\mathbf{x}$ in SSL. Since the format allows to declare variables that are not used, we set $\mathbf{x}$ to be the set of all declared location variables (plus nil) in the input file.

After parsing, we perform a basic preprocessing of the input. Its main reason is to introduce guarded negations which are not explicitly specified in the input. This is achieved by pushing negations bottom-up as far as possible (i.e., until they reach either the top of the formula or a spatial connective). The process may yield a positive formula even for

[1] https://github.com/TDacik/Astral
[2] https://github.com/Gbury/dolmen

```
;; Declaration of location sort. Currently fixed to this form by Astral.
(declare-sort Loc 0)

;; Declaration of heap sort. Currently fixed to this form by Astral.
(declare-heap (Loc Loc))

;; Declaration of location variables
(declare-const x Loc)
(declare-const y Loc)
(declare-const z Loc)

;; Input formula
(assert (ls x y))

(assert
  (sep
    (not emp)
    (not emp)
    (not emp)
  )
)

(check-sat)
```

Figure 6.1: An example of the input format for the formula $\mathsf{ls}(x, y) \wedge (\neg\mathsf{emp} * \neg\mathsf{emp} * \neg\mathsf{emp})$ over the set of variables $\mathbf{x} = \{x, y, z, \mathsf{nil}\}$. The variable nil is always added implicitly.

inputs where this is not obvious. For example, the formula $\mathsf{ls}(x, y) \wedge (\neg x = y \wedge \neg x \mapsto y)$ will be rewritten to the positive formula $\mathsf{ls}(x, y) \wedge_{\neg} (x = y \vee x \mapsto y)$.

6.3 SMT Back-end

The first version of Astral implemented the translation directly using an OCaml binding for Z3. Later, we have found interesting to try performance of other SMT solvers. This would be more easy, if we could use theories standardised in the SMT-LIB standard – we would use the Z3 OCaml binding to output the translated formula in the SMT-LIB format and then call another solver on the file. Neither the theory of sets or the generalised theory of arrays is, however, not standardised by SMT-LIB.

We have therefore decided to implement a more generic SMT back-end that allows Astral to use multiple SMT solvers. Instead of using the Z3 OCaml binding for representation of the translated formula, we implemented our own inner representation of SMT formulae and models. The inner representation of formulae is then translated to the input language of the selected solver using a back-end for the given solver. If the solver returns `sat` and a model, then the back-end translates the model back into our inner representation.

On the one hand, this solution required to re-implement some features already provided by the Z3 binding (such as substitution of terms). On the other hand, it allowed us to work with higher-level concepts during the translation and let the low-level details of their translation to individual back-ends. Currently, we have two back-ends for concrete solvers and one for their parallel combination:

- **Z3 back-end** – The Z3 solver is the default one and is therefore always installed with Astral. The translation from our inner representation is done using the OCaml binding of Z3. Our experiments show that this backend is faster for formulae including list-segment predicates.

- **cvc5 back-end** – The cvc5 solver is not installed together with Astral and if one wants to use it, it has to be installed in the path. Since it currently does not have an OCaml binding, we translate our internal representation to the SMT-LIB format using the cvc5's syntax for sets, store it to a temporary file, and call cvc5 in another process. After the solver finishes, we have to parse the model from its SMT-LIB representation. This of course bring some additional overhead, but the cvc5 back-end still usually performs better than Z3 backend for formulae which do not contain list-segment predicates.

- **Parallel back-end** – Our experiments show that none of the previously mentioned solvers is strictly better. An obvious solution is therefore to run them both in parallel and wait for the first one which returns a result. We have implemented this approach using `Domainslib`[3] which implements high-level mechanisms for running multiple tasks in parallel running threads.

  Unfortunately, thread-level parallelism is not available in OCaml prior to its version 5.0 because of its usage of a *global runtime lock*[4]. Since OCaml 5.0 is still in its alpha version, some libraries used in Astral are not compatible with it. The parallel back-end therefore could not be merged into the main branch of Astral and experimentally evaluated.

[3] https://github.com/ocaml-multicore/domainslib
[4] https://ocamlverse.github.io/content/parallelism.html

Chapter 7

Experimental Evaluation

This section is devoted to an experimental evaluation of the proposed decision procedure. First, we focus on a comparison with other translation-based decision procedures implemented in the tools Sloth [18] and Grasshopper [28]. We performed experiments on two categories of the international competition SL-COMP [32]. Those categories include manually crafted formulae and also real-life verification conditions generated by verification tools.

Then, we conducted an experimental comparison with the decision procedure implemented in the SMT solver cvc5. Since Astral cannot handle benchmarks used to evaluate cvc5 [30], which frequently contain unguarded negations, we prepared our own benchmarks focused on guarded negations and septractions. Those benchmarks consist of crafted parametric formulae with growing complexity and randomly-generated formulae.

During this chapter, we will use Astral-Z3 and Astral-cvc5 to refer to the the Astral solver running with Z3 and cvc5 back-end, respectively. All experiments were conducted on a machine with 2.5 GHz Intel Core i5-7300HQ processor and 16 GiB RAM, running Ubuntu 18.04. The benchmark consisting of preprocessed formulae from SL-COMP, generated parametric formulae, and randomly-generated formulae is available as a github repository[1]. The repository also contains translations of those formulae to formats used by Sloth and Grasshopper.

7.1 Comparison with Translation-Based Decision Procedures

First, we will compare Astral with other decision procedures based on a translation to SMT. The first of them is Sloth [17], which implements a translation based on a small-model property and was the main inspiration of our approach. The second is Grasshopper, which translates the input formula to an intermediate logic called GRASS, which is later translated to SMT using a partial instantiation of GRASS axioms [28]. Note that Grasshopper is not a solver, rather a verification tool for heap-manipulating programs. To run it as a solver with minimal overhead, we encode an entailment formula $\varphi \models \psi$ as the empty program with the precondition φ and postcondition ψ. Such a program is verified iff the entailment is valid. Similarly, we encode satisfiability of a formula φ as the empty program with the precondition φ and postcondition $\bot$. Such a program is verified iff the formula is unsatisfiable.

[1] https://github.com/TDacik/seplog-bench/

Table 7.1: Experimental results for the category QF_SHLS_SAT.

	Results				Times [s]		
Solver	Correct	Wrong	Timeouts	Winner	Total	Mean	Maximal
Astral-cvc5	110	0	0	100	6.71	0.06	0.13
Astral-Z3	110	0	0	10	31.47	0.28	3.48
Grasshopper	110	0	0	0	161.09	1.46	11.03
Sloth	0	0	110	0	-	-	-

Table 7.2: Experimental results for the subset of the category QF_SHLS_ENTL containing verification conditions. The total and mean time are computed including TOs, maximum time excluding TOs.

	Results				Times [s]		
Solver	Correct	Wrong	Timeouts	Winner	Total	Mean	Maximal
Astral-cvc5	85	0	1	27	75.11	0.87	0.82
Astral-Z3	86	0	0	22	4.67	0.05	0.70
Grasshopper	86	0	0	37	5.37	0.06	1.99
Sloth	62	19	5	0	637.26	20.44	7.41

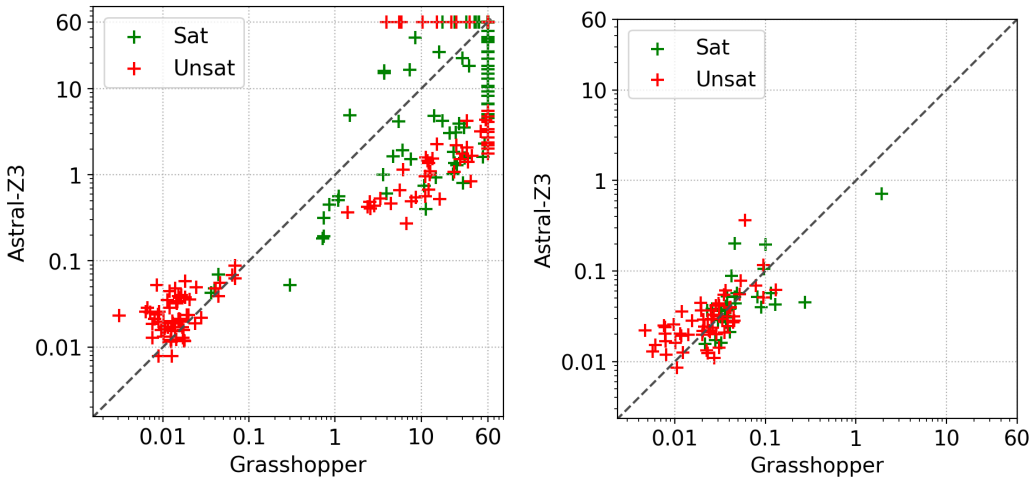
Both Astral and Grasshopper are implemented in OCaml. Sloth is implemented in Python, and the results can be therefore skewed by different speeds of those languages (Ocaml is believed to be faster in general because it is a compiled language). We could measure just the time of calls to an SMT solver, but this would ignore improvements in translation such as the bound computation used in Astral. We therefore decided to measure the overall run time for all solvers. Another source of distortion can be usage of different backend SMT solvers. As for Astral, we used it in modes running Z3 and cvc5. Grasshopper can use both Z3 and cvc4 (an older version of cvc5), but its latest version crashes when Z3 is used. Therefore, we use it only with the cvc4 back-end. Sloth can be run only using Z3.

In the comparison, we focused on the categories QF_SHLS_SAT and QF_SHLS_ENTL of SL-COMP, which stand for satisfiability and entailment in the symbolic heap fragment with lists, respectively. The satisfiability benchmark consists solely of randomly generated formulae. The complexity of those formulae ranges from 10 to 20 variables with an increasing number of atoms. The entailment benchmark contains both crafted formulae and real-life verification conditions. Those verification conditions mostly originate from the tool Smallfoot [7]. The crafted formulae are either randomly generated, or they are created by cloning the previously mentioned verification conditions (note that the cloning is used only to increase the complexity and such formulae do not represent verification problems anymore). The process of generating and cloning is in details described in [27]. Because the difficulty of crafted formulae and verification conditions differ (random formulae contain up to 20 list-segment predicates while verification conditions not more than 5), we consider them as two separate categories in our experiment. We set the timeout of 60 seconds for all experiments in this section.

The results for the category QF_SHLS_SAT are given in Table 7.1. The table shows that Sloth is not able to solve any of the formulae, and both configurations of Astral outperform Grasshopper. Moreover, Astral-cvc5 wins in almost 90 % of all cases.

Table 7.3: Experimental results for crafted formulae from the category QF_SHLS_ENTL. The total and mean time are computed including TOs, maximum time excluding TOs.

	Results				Times [s]		
Solver	Correct	Wrong	Timeouts	Winner	Total	Mean	Maximal
Astral-cvc5	66	0	144	44	8 651	41.19	7.35
Astral-Z3	174	0	36	125	3 072	14.63	57.41
Grasshopper	140	0	70	25	5 480	26.09	52.03
Sloth	68	0	142	0	8 744	41.63	29.98



(a) QF_SHLS_ENTL (random formulae) (b) QF_SHLS_ENTL (verif. conditions)

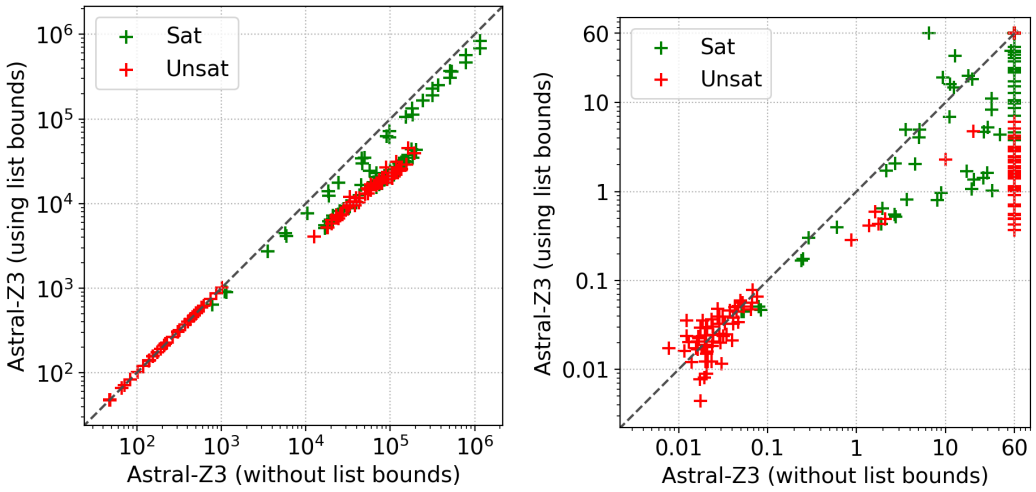
Figure 7.1: A comparison of running times of Astral-Z3 and Grasshopper on entailments in the symbolic heap fragment. Times are in seconds and timeout was set to the 60 seconds. Axes are logarithmic.

Based on our experiments, the significant difference between Astral and Sloth is due to improved bounds proved in Section 5.1.

The results for verification conditions from the category QF_SHLS_ENTL are given in Table 7.2. All formulae were correctly solved by both Astral-Z3 and Grasshopper. While Grasshopper wins in more cases, Astral-Z3 is faster overall. The difference is, however, negligible. This can be also seen in Figure 7.1b. Astral-cvc5 times out in one case, but otherwise solves all formulae under one second. This demonstrates that Astral can effectively solve formulae coming from real-life applications. This is not true for Sloth which times out in five cases even on very simple formulae. Moreover, in 19 cases, it returns „invalid“ for a valid entailment. This seems to be an implementation bug because it manifests even for simple entailments such as $\mathsf{ls}(x, y) \models \mathsf{ls}(x, y)$. We have reported the issue[2], but it was not confirmed at the time of writing this thesis.

Results for crafted formulae from the category QF_SHLS_ENTL are given in Table 7.3. The results suggest that formulae with many list-segment predicates (up to 20) are hard for all translation-based solvers. The best is Astral-Z3 which, however, still timeouts in 36 cases. A detailed comparison of Astral and Grasshopper is given in Figure 7.1a. The figure shows that Grasshopper wins mostly on easy unsatisfiable formulae that are

[2] https://github.com/katelaan/sloth/issues/1



(a) Translated formula size

(b) Running time

Figure 7.2: A comparison of Astral-Z3 running with the list-length bounds computation and without it for crated entailments in the symbolic heap fragment. Times are in seconds and the timeout was set to 60 seconds. Axes are logarithmic.

solved under a tenth of second by both solvers. Astral times out mostly for unsatisfiable formulae, but it is able to solve many satisfiable formulae that Grasshopper cannot solve.

We also compared Astral with Asterix [22] which won the previous edition of SL-COMP in the considered categories. Asterix can solve all instances almost immediately (under 0.006 seconds) and beats Astral in all the cases. This is, however, an expected result because Asterix implements a specialised algorithm for the symbolic heap fragment while Astral targets much more complex logic.

7.2 Evaluation of List-Length Bounds Computation

We believe that the main improvement of the translation implemented in Astral are methods for bound computation. Especially, methods for computing bounds of lengths of list-segment predicates. To verify this hypothesis, we run Astral with and without the list-length bound computation on crafted formulae from the category QF_SHLS_ENTL. Notice that, for satisfiability in the symbolic heap fragment, the list-length bound computation does not help because, in this fragment, we always have the bound $[0, 1]$ for each list-segment predicate by Lemma 5.2.

First, we compare the sizes of translated formulae. We measure the size of a formula as the number of nodes in its AST. The size is measured without any simplification. The results are shown in Figure 7.2a. The size of translated formulae ranges from 100 to 1 million. There are several clusters of formulae which are probably caused by the fact that those formulae are crafted and randomly-generated. For some formulae, there is no difference in size, but there are formulae whose size is more than five times lesser when the list bounds are used.

Figure 7.2b shows that the reduced size has a significant positive impact on the running time. In 65 cases out of 210, it allows us to solve problems which would otherwise timeout. Among of them, there is a lot of unsatisfiable formulae that are now solved under one second.

There are several satisfiable formulae such that the running time is higher although their size is smaller (one of them even timeouts), but the heuristics performs still better for a majority of satisfiable formulae. However, it seems that the list-length bound computation helps more in the case when formula is unsatisfiable. This is natural because it restricts the state space that an SMT solver has to search to declare a formula as unsatisfiable. On the other hand, this could be a consequence of how formulae are generated.

7.3 Comparison with cvc5

In this section, we present an experimental comparison of Astral with the decision procedure for SL implemented in the SMT solver cvc5. This decision procedure targets a fragment that is incomparable with the fragment supported by Astral. On the one hand, cvc5 supports arbitrary magic wands. On the other hand, it does not support list-segment predicates at all. Moreover, in the presence of unguarded negations, there could be a difference between the standard semantics of separation logic used by cvc5 and the strong-separation semantics used by Astral. For the following experiment, we have extended Astral with an option to perform translation in the classical semantics (the translation will not generate constraints that locations shared by sub-heaps are named). We will not prove this claim, but with this modification, Astral should be sound for the considered fragment under the classical semantics of SL.

We first tried Astral on the SL-COMP category QF_BSL_SAT which precisely corresponds to the fragment supported by cvc5, which was also the only participant in this category in the last edition of SL-COMP[3]. Formulae from this benchmark frequently contain a negation under a separating conjunction which itself lies under another negation. Such formulae are extremely hard for Astral because they trigger an extensive enumeration over footprints when separating conjunctions are translated. Consequently, Astral was able to solve only two simplest formulae of the category. In the rest of the experiments, we therefore focused on a fragment that contains negations in a limited form only.

7.3.1 Parametric Formulae

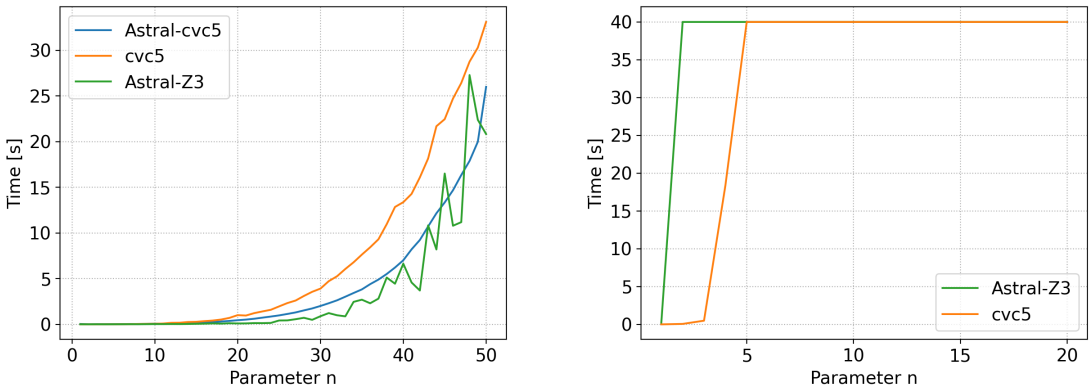
To do a comparison on a fragment that Astral can handle, we prepared several sets of parametric formulae with growing complexity based on a parameter n. Those formulae focus on usage of septractions, and negations under separating conjunctions, i.e., features that are extensions of the previously proposed translation-based procedures. Note that cvc5 does not support septractions directly and we therefore encode them as magic wands. We used the time limit of 40 seconds for all the experiments.

- *Heap size.* The first formula states that the heap can be split into n non-empty sub-heaps, i.e., that the heap has size at least n:

$$\mathsf{size}^{\geq n} \triangleq \underbrace{\neg\mathsf{emp} * \cdots * \neg\mathsf{emp}}_{n \text{ times}}$$

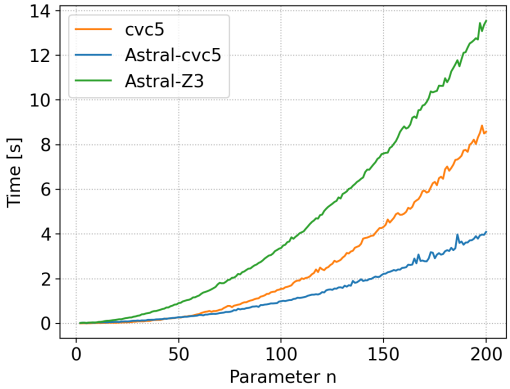
  The formula contains negations under separating conjunctions, but all separating conjunctions can be translated using Skolemization. The results in Figure 7.3a show that Astral can solve such formulae efficiently and even slightly faster than cvc5.

[3] https://www.irif.fr/~sighirea/sl-comp/19/qf_bsl_sat.html

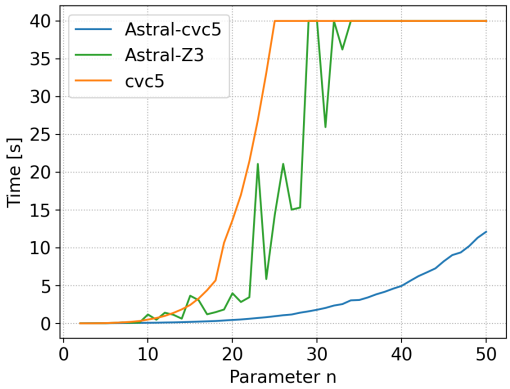


(a) Heap size

(b) Exact heap size



(c) Septractions



(d) Pointers using septractions

Figure 7.3: A comparison of Astral and cvc5 on parametric formulae with complexity growing based on a parameter n. The timeout was set to 40 seconds.

- *Exact heap size.* The second formula states that the heap has size *exactly* n:

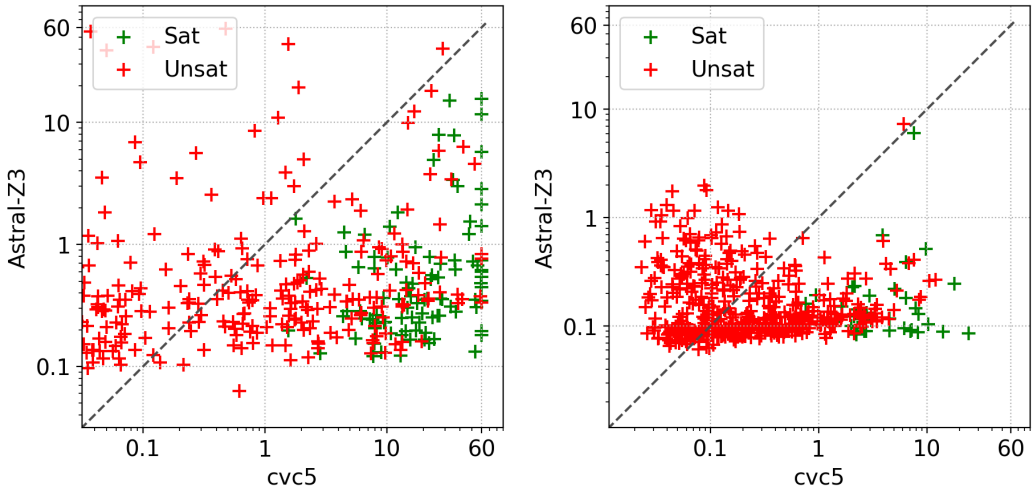
$$\mathsf{size}^{=n} \triangleq \mathsf{size}^{\geq n} \wedge \neg \mathsf{size}^{\geq n+1}$$

  Unlike in the case of the previous formula, separating conjunctions in the sub-formula $\neg \mathsf{size}^{\geq n+1}$ cannot be translated using Skolemization. Figure 7.3b shows that the formula is indeed very hard for all solvers even for very small n. Astral-Z3 is able to solve it for $n = 1$ only (and, for $n = 2$, in 47 seconds, which is slightly above the time limit) and cvc5 for $n = 4$ only. Astral-cvc5 is not shown in the figure because its backend solver always gives-up and returns `unknown`.

- *Septractions.* The third formula uses septractions to express that variables $x_1, \ldots, x_n$ are not allocated:

$$\mathsf{not\_alloc}(x_1, \ldots, x_n) \triangleq \big((x_1 \mapsto \mathsf{nil}) \mathrel{-\!\circledast} \mathsf{true}\big) * \cdots * \big((x_n \mapsto \mathsf{nil}) \mathrel{-\!\circledast} \mathsf{true}\big)$$

  The formula can be trivially satisfied by the empty heap. We use it to benchmark how Astral can deal with septractions combined with negations (the atom true is syntactic sugar for $\mathsf{emp} \vee \neg\mathsf{emp}$). Due to its simplicity, the formula can be quickly



(a) Guarded negations (b) Septractions

Figure 7.4: A comparison of Astral-Z3 and cvc5 on randomly generated formulae. The timeout was set to 60 seconds. Axes are logarithmic.

solved by all solvers even for 200 variables. Astral-cvc5 performs best, and, for $n = 200$, it is two times faster than cvc5.

- *Pointers using septractions.* The last formula expresses that the heap contains a cyclic sequence of pointers using septractions:

$$\mathsf{ptr\_septr}^n \triangleq \big(\mathsf{emp} \multimap\!\circledast\ x_1 \mapsto x_2\big) * \cdots * \big(\mathsf{emp} \multimap\!\circledast\ x_{n-1} \mapsto x_n\big) * \big(\mathsf{emp} \multimap\!\circledast\ x_n \mapsto x_1\big)$$

  The results in Figure 7.3c show that both versions of Astral outperform cvc5. Moreover, Astral-cvc5 is able to solve formulae for $n = 50$ quite fast, while cvc5 runs out of the time already for $n = 25$.

7.3.2 Randomly Generated Formulae

To further compare solvers on problems with less regular structure than in the case of parametric formulae, we prepared two sets of randomly generated formulae. All formulae were generated as random binary balanced trees of depth six over eight variables. Those parameters were selected based on experiments to achieve a reasonable complexity of the generated formulae. Atoms were restricted to points-to assertions only. Pure atoms were not used because cvc5 uses an imprecise semantics for them (they can be satisfied on an arbitrary heap) and Astral uses the precise semantics (they can be satisfied on the empty heap only). Those semantics may be easily converted to each other, but we rather do not use them in this experiment. We use the QCheck tool [4] to generate the formulae. We have generated two sets of 500 formulae. Those sets differs in the allowed connectives:

- *Guarded negations.* This fragment focuses on mixing separating conjunctions with boolean conjunctions, disjunctions and guarded negations. The top-level connective is always a guarded negation (the formulae therefore represent entailments). Note

[4] https://github.com/c-cube/qcheck

that those formulae are not necessary in the fragment SSL^E, i.e., their translation can have an exponential size. This is because separating conjunctions can be negated by guarded negations and footprints are not guaranteed to be unique because of disjunctions. However, the exponential blow-up should not be as significant as in case of unguarded negations.

- *Septractions.* In this set, we added septractions but removed guarded negations. All formulae of the set are therefore in SSL^E because all separating conjunctions can be translated using Skolemization.

We used Astral-Z3 for the comparison. It would be better to use Astral-cvc5 to show that differences are not caused by other back-end technologies, but on many of the randomly generated formulae, Astral-cvc5 gives-up with the `unknown` result. It seems that during the translation, we use some combinations of features that is not supported by cvc5. However, we have not been able to track down what this combination is at the time of writing this thesis. On the other hand, all previous experiments show that Astral-cvc5 is faster than Astral-z3 on formulae without list-segment predicates, and we therefore believe that the comparison is fair.

The results for the first set are shown in Figure 7.4a. Due to the way how the formulae were generated, there are more unsatisfiable formulae. On almost all satisfiable formulae, Astral-Z3 is faster. There are also several satisfiable formulae which cvc5 cannot solve in the limit but Astral-Z3 solves them under one second. The results for the second set are shown in Figure 7.4b. Here, almost all generated formulae are unsatisfiable. Again Astral-Z3 is faster for all satisfiable. In our future work, we would like to more precisely evaluate those experiments. In particular, we would like to run the experiment also with Astral-cvc5 to see whether results are influenced by back-end SMT solver.

When performing experiments, we have found several formulae for which Astral and cvc5 produced different results. It turned out that the problem was with septractions and that incorrect results were produced by cvc5. We prepared a minimal example of the incorrect behaviour and reported it[5]. The problem was in a heuristic that would, e.g., for the septraction $x \mapsto y \mathbin{-\!\circledast} x \mapsto y$ conclude that the pointer $x \mapsto y$ has to be in the model. This is of course not true because the formula can be satisfied by the empty heap only. The problem seems trivial when a septraction is used but it is much more complicated when looking from the perspective of magic wands which are used in cvc5. The issue was fixed, but when we repeated our experiments, we have found that the fix has introduced another unsoundness[6]. Again, the issue was confirmed and fixed.

7.4 Summary and Future Work

Our experiments showed that Astral outperforms existing translation-based decision procedures implemented in the tools Sloth and Grasshopper on the frequently used symbolic heap fragment. In the case of satisfiability for this fragment, our improvements are due to improved bounds proved in Section 5.1. In the case of entailment, we have experimentally evaluated that the improvement is due to the computation of bounds on lengths of list segment predicates. Moreover, Astral is able to efficiently solve all of considered problems that originate from verification tools.

[5] https://github.com/cvc5/cvc5/issues/8659
[6] https://github.com/cvc5/cvc5/issues/8863

The comparison with the CVC5 solver shows that ASTRAL has a problem with formulae containing unguarded negations in such a way that it cannot use Skolemization. However, we expected this because of our way of translating separating conjunctions using an extensive enumeration over footprints and the fact, that we currently do not have heuristics to tackle it. Future work in this direction can focus on trying to reduce possible footprints of negations. This could be done, e.g., based on computation of variables that cannot be allocated by the given negation using SL-graphs. Another possible direction is to develop a method to perform the enumeration over footprints lazily.

Chapter 8

Conclusion

In this thesis, we proposed a decision procedure for strong-separation logic based on a translation to SMT and implemented this decision procedure in a new solver called Astral. The translation is inspired by the previous works, but we have significantly extended the fragment that can be translated. Those extensions include support for negations, limited usage of septractions (and therefore also limited usage of magic wands), and support for mixing of boolean and spatial connectives. We also proposed several original heuristics to decrease size of translated formulae. Our experimental results showed that those heuristics help our decision procedure to outperform other translation-based decision procedures implemented in the tools Sloth and Grasshopper. The comparison with the decision procedure implemented in the prominent SMT solver cvc5 on its own benchmark showed that Astral cannot handle some classes of formulae containing negations yet. On the other hand, experiments on parametric and randomly generated formulae suggest that Astral can efficiently handle formulae containing septractions or negations in the so-called guarded form. On formulae containing guarded negations, it even significantly outperforms the cvc5 solver. Moreover, based on those experiments, we found and reported several incorrect results produced by cvc5 for formulae containing magic wands. Those turned to be results of incorrect heuristics and were fixed based on our reports.

Future work. There are many possible directions for the future work. First of them is to design an efficient methods to deal with formulae which contain unguarded negations, e.g., by using lazy enumeration when translating separating conjunctions. Another interesting research direction is to extend expressivity of SSL. While trees and data constraints were already studied in [24], another extensions such as user-defined inductive predicates or quantifiers were not yet studied in the context of SSL. Finally, we would like to also study how SSL can be used in automated program verification. In this direction, we would like to focus on the so-called bi-abductive analysis [10].

Bibliography

[1] Appel, A. W., Dockins, R., Hobor, A., Beringer, L., Dodds, J. et al. *Program Logics for Certified Compilers*. USA: Cambridge University Press, 2014. ISBN 110704801X.

[2] Bansal, K., Barrett, C., Reynolds, A. and Tinelli, C. A New Decision Procedure for Finite Sets and Cardinality Constraints in SMT. In: *IJCAR*. 2017.

[3] Barbosa, H., Barrett, C. W., Brain, M., Kremer, G., Lachnitt, H. et al. Cvc5: A Versatile and Industrial-Strength SMT Solver. In: *TACAS*. 2022.

[4] Barrett, C., Fontaine, P. and Tinelli, C. *The SMT-LIB Standard: Version 2.6* [`www.SMT-LIB.org`]. 2021.

[5] Batz, K., Fesefeldt, I., Jansen, M., Katoen, J.-P., Kessler, F. et al. Foundations for Entailment Checking in Quantitative Separation Logic. In: Sergey, I., ed. *Programming Languages and Systems*. Cham: Springer International Publishing, 2022.

[6] Berdine, J., Calcagno, C. and O'Hearn, P. W. A Decidable Fragment of Separation Logic. In: *FSTTCS*. 2004.

[7] Berdine, J., Calcagno, C. and O'Hearn, P. W. Symbolic Execution with Separation Logic. In:. Berlin, Heidelberg: Springer-Verlag, 2005. APLAS'05.

[8] Bradley, A. R. and Manna, Z. *The Calculus of Computation: Decision Procedures with Applications to Verification*. 1stth ed. Springer Publishing Company, Incorporated, 2010. ISBN 3642093477.

[9] Brotherston, J., Gorogiannis, N. and Petersen, R. L. A Generic Cyclic Theorem Prover. In: *APLAS*. 2012.

[10] Calcagno, C., Distefano, D., O'Hearn, P. W. and Yang, H. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM*. New York, NY, USA: Association for Computing Machinery. 2011.

[11] Cook, B., Haase, C., Ouaknine, J., Parkinson, M. and Worrell, J. Tractable Reasoning in a Fragment of Separation Logic. In: *Proceedings of the 22nd International Conference on Concurrency Theory*. Berlin, Heidelberg: Springer-Verlag, 2011. CONCUR'11.

[12] Demri, S., Lozes, É. and Mansutti, A. The Effects of Adding Reachability Predicates in Propositional Separation Logic. In: Baier, C. and Lago, U. D.,

ed. *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018*. Springer, 2018.

[13] Echenim, M., Iosif, R. and Peltier, N. The Bernays-Schönfinkel-Ramsey Class of Separation Logic with Uninterpreted Predicates. *ACM Transactions on Computational Logic*. 2019, vol. 21.

[14] Enea, C., Lengál, O., Sighireanu, M. and Vojnar, T. Compositional Entailment Checking for a Fragment of Separation Logic. USA: Kluwer Academic Publishers. dec 2017, vol. 51, no. 3, p. 575–607. ISSN 0925-9856.

[15] Iosif, R., Rogalewicz, A. and Vojnar, T. *Deciding Entailments in Inductive Separation Logic with Tree Automata*. 2014.

[16] Iosif, R., Serban, C., Reynolds, A. and Sighireanu, M. Encoding Separation Logic in SMT-LIB v2.5. In:. 2018.

[17] Katelaan, J., Jovanovic, D. and Weissenbacher, G. A Separation Logic with Data: Small Models and Automation. In: *IJCAR*. 2018.

[18] Katelaan, J., Jovanovic, D. and Georg, W. Sloth: Separation Logic and Theories via Small Models. In: *Informal proceedings of the First Workshop on Automated Deduction for Separation Logics (ADSL)*. 2018.

[19] Katelaan, J., Matheja, C., Noll, T. and Zuleger, F. Harrsh: A Tool for Unied Reasoning about Symbolic-Heap Separation Logic. In: Barthe, G., Korovin, K., Schulz, S., Suda, M., Sutcliffe, G. et al., ed. *LPAR-22 Workshop and Short Paper Proceedings*. 2018, vol. 9. Kalpa Publications in Computing.

[20] Moura, L. de and Bjørner, N. Generalized, efficient array decision procedures. In: *2009 Formal Methods in Computer-Aided Design*. 2009, p. 45–52.

[21] Moura, L. M. de and Bjørner, N. S. Z3: An Efficient SMT Solver. In: *TACAS*. 2008.

[22] Navarro Pérez, J. A. and Rybalchenko, A. Separation Logic Modulo Theories. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. march 2013, vol. 8301.

[23] O'Hearn, P. W. Resources, Concurrency and Local Reasoning. *Theor. Comput. Sci.* 2004, vol. 375.

[24] Pagel, J. *Decision Procedures for Separation Logic: Beyond Symbolic Heaps*. Dissertation.

[25] Pagel, J. and Zuleger, F. Strong-Separation Logic. In:. March 2021, p. 664–692. ISBN 978-3-030-72018-6.

[26] Parkinson, M. J. The Next 700 Separation Logics - (Invited Paper). In: *VSTTE*. 2010.

[27] Pérez, J. A. N. and Rybalchenko, A. Separation logic + superposition calculus = heap theorem prover. In: *PLDI '11*. 2011.

[28] Piskac, R., Wies, T. and Zufferey, D. Automating Separation Logic Using SMT. In: Sharygina, N. and Veith, H., ed. *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 773–789.

[29] Piskac, R., Wies, T. and Zufferey, D. Automating Separation Logic with Trees and Data. In: *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Berlin, Heidelberg: Springer-Verlag, 2014, p. 711–728. ISBN 9783319088662.

[30] Reynolds, A., Iosif, R. and King, T. A Decision Procedure for Separation Logic in SMT. In: *ATVA*. 2016.

[31] Reynolds, J. Separation logic: A logic for shared mutable data structures. In:. February 2002, p. 55– 74. ISBN 0-7695-1483-9.

[32] Sighireanu, M., Navarro Pérez, J. A., Rybalchenko, A., Gorogiannis, N., Iosif, R. et al. SL-COMP: Competition of Solvers for Separation Logic. In:. 2019.

[33] Ta, Q.-T., Le, T. C., Khoo, S.-C. and Chin, W.-N. Automated Lemma Synthesis in Symbolic-Heap Separation Logic. *Proc. ACM Program. Lang.* New York, NY, USA: Association for Computing Machinery. 2017, vol. 2, POPL.

Appendix A

Contents of the Attached Medium

The attached memory medium contains the following:

```
/
├── Astral/ ...  source code of Astral
├── tex/ ...  source codes of this thesis
├── xdacik00.pdf ...  this thesis in PDF
└── seplog_bench/ ...  formulae used for experiments
```

Appendix B

Installation and Usage

Source code of Astral can be found on the attached medium or online at https://github.com/TDacik/Astral. The solver can be installed via opam package manager by cloning the repository and running:

```
$ opam install
```

By default, Astral is installed with the Z3 solver. To use Astral with cvc5 backend, it has to be installed manually and present in the path. After Astral is installed, it can be run by the following command:

```
$ astral [options] formula.smt2
```

The most common options are:

- `--debug` ... Store debug information such as translated formula in `.smt2` format or SMT models in `astral_debug` directory.
- `--backend=<cvc5|z3>` ... Select backend SMT solver.
- `--loc-bound=<n>` ... Force location bound to be n (potentially unsound).
- `--no-list-bounds` ... Do not use optimised translation of list-segment predicates
- `--semantics=<weak|strong>` ... Default is `strong`. When option `weak` is used, result can be unsound for formulae with negations.