Automata in Decision Procedures and Performance Analysis

Tomáš Fiedor

October 31, 2019

Abstract

This thesis focuses on improving the state of the art of automata-based formal analysis and verification techniques for systems with an infinite state space.

In the first part of the thesis, we develop two efficient decision procedures for the WS1S logic, both of them exploiting the correspondence between formulae of WS1S logic and finite automata. We start by proposing a novel antichain-based decision procedure which is, however, limited to formulae in the prenex normal form. Later, we generalize the approach to arbitrary formulae by defining the so-called language terms and constructing an on-the-fly procedure dealing with the terms using lazy techniques. In order to achieve an efficient implementation, we propose numerous optimizations (some of these optimization are not limited to our approaches only). We evaluated both our methods with other recent state-of-the art tools. The achieved results are encouraging and show we can extend the usage of WS1S to wider classes of formulae.

The second part of the thesis focuses on resource bounds analysis of heap-manipulating programs. We propose a new class of shape norms based on lengths of paths between distinct points in the heap, which we derive automatically from the analysed program. For this class of norms, we introduce a calculus capable of precisely inferring changes of the analysed norms and use it to generate a corresponding integer representation of an input program followed by dedicated state-of-the art resource bounds analysis. We implemented our approach over the shape analysis based on forest-automata, implemented in the Forester tool, and using a well-established resource bounds analyser, implemented in the Loopus tool. In our experimental evaluation, we show that we indeed obtained a powerful analyser that is able to handle some showcase examples that were never analysed fully automatically before.

Keywords

Antichains, amortized complexity, binary decision diagrams, finite automata, forest automata, formal verification, heap-manipulating programs, monadic logic, nondeterminism, resource bounds analysis, second-order logic, shape analysis, shape norms, static analysis, tree automata, ws1s.

Abstrakt

Tato práce se věnuje vylepšení současného stavu formalní analýzy a verifikace založené na automatech a zaměřené na systémy s nekonečnými stavovými prostory.

V první části se práce zabývá dvěma rozhodovacími procedurami pro logiku WS1S, které jsou založené na korespondenci mezi formulemi logiky WS1S a konečnými automaty. První metoda je založena na tzv. antiřetězcích, ale, je limitována pouze na formule v prenexním normálním tvaru. Následně je tento přístup zobecněn na libovolné formule, jsou zavedeny tzv. jazykové termy a na jejich základě je navržena nová procedura, která pracuje za běhu a zpracovává tyto termy "líným" způsobem. Abychom získali efektivní rozhodovací proceduru, je dále navržena sada optimalizací (přičemž některé nejsou limitovány pouze pro naše přístupy). Obě metody jsou srovnány s ostatními nástroji implementujícími různé známé rozhodovací procedury. Získané výsledky jsou povzbuzující a ukazují, že použitelnost logiky WS1S je možno rozšířit na širší třídu formulí.

V druhé části se práce zabývá analýzou mezí zdrojů programů manipulujících s haldou. Je zde navržena nová třída tzv. tvarových norem založených na délkách cest mezi význačnými místy na haldě, které jsou automaticky odvozovány z analyzovaného programu. Na základě této třídy norem je dále navržen kalkul, který je schopen přesně odvodit změny odvozených normů a použít je k vygenerování odpovídající celočíselné reprezentace vstupního programu, která je následně využita pro následovanou dedikovanou analýzou mezí zdrojů. Tato metoda byla implementována nad analýzou tvaru založenou na tzv. lesních automatech, implementovanou v nástroji Forester, a dále byl použit dobře zavedený analyzátor mezí zdrojů, implementovaný v nástroji Loopus. V experimentální evaluaci bylo ukázáno, že je opravdu takto získán silný analyzátor, který je schopen odvodit meze programů, které ještě nikdy plně automatizovaně odvozené nebyly.

Klíčová slova

Analýza mezí zdrojů, analýza tvaru, antiřetězce, amortizovaná složitost, binární rozhodovací diagramy, formální analyza, konečné automaty, logika druhého řádu, lesní automaty, monadická logika, programy manipulující s haldou, nedeterminismus, statická analýza, stromové automaty, tvarové normy, ws1s.

Citace

Tomáš Fiedor, Automata in Decision Procedures and Performance Analysis, disertační práce, Brno, FIT VUT v Brně, 2019

Automata in Decision Procedures and Performance Analysis

Prohlášení

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně pod vedením prof. Ing. Tomáše Vojnara, Ph.D. a doc. Mgr. Adama Rogalewicze, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Tomáš Fiedor
23. října 2019

Acknowledgements

First of all, I would like to thank both of my supervisors Tomáš Vojnar and Adam Rogalewicz for all their support, help and contributions. Further, I would like to thank all of my co-authors, Lukáš Holík, Petr Janků, Ondřej Lengál, Moritz Sinn and Florian Zuleger for their collaboration on implementation, experimental evaluation or discussions, but most of all, for their great writing. I would like to thank the whole VeriFIT group, for making the doctorate fun, in particular, I would like to thank Martin Hruška for all the fun, Zdeněk Letko for being great mentor during the first year of my PhD studies, and Hanka Pluháčková for all the help with statistics and her great illustrations.

I would like to thank all of my students, which I have supervised, especially Jiří Pavela and Michal Kotoun. Both of them did an outstanding work on projects, in which I have contributed. Moreover, I would like to thank Jan Zelený, Viktor Malík and Red Hat Brno for providing me the scholarship and, but most of all, for showing an interest in my research.

I would like to thank all of my good friends, Maro, Bobo, Špendla, Maslík, Jana, Lenka, and Feši, for binge drinking, playing board games, various vacations and parties. At last, I would like to thank my family for being supportive, especially my brother Honza, who helped me with both technical and non-technical stuff during my studies.

The work presented in this thesis was supported by the Czech Ministry of Education, Youth, and Sports (under the project LTE118019), the Czech Science Foundation (under the projects GP13-37876P, GA14-11384S, GA16-17538S, GJ16-24707Y, and GA17-12465S), the internal BUT FIT projects FIT-S-14-2486 and FIT-S-17-4014, and the ECSEL Joint Undertaking (under the AQUAS and Arrowhead H2020 projects).

Contents

1. Introduction

For almost a century, computer technology has been a necessary part of our lives: mankind exploits it in everyday life, in medicine, in transportation or in heavy industry. Naturally, computer programs should be error-free, since any error can have either little (such as bad user experience), medium (such as cash loss) or even severe consequences (such as accidents or crashes). To prevent these errors before-hand, one can try to use formal analysis and verification, however, these techniques still face a great challenge: complex systems leads to an analysis of infinite state space, and, in many cases, even to infeasibility. While, for many classes of programs we can prove or analyse many properties (including safety, termination or atomicity of programs), every year the list keeps growing with many new properties to be checked and many new characteristics of programs to be analysed leading to brand new challenges.

For instance recently, developers have been more frequently demanding tools that would help them understand the performance of their code. In some cases, they even need to verify that their programs stay within the expected resource bounds (i.e. bounds on the expected consumption of computational time, memory, disk space, energy, etc.) or at least obtain a reasonable estimate of the program performance. In their software, *performance-related issues* are common and lead to a poor user experience or a waste of computational resources, as is documented by many recent studies [JSS+12, NJT13]. These studies claim that the root cause of all of these issues is that developers do not understand the performance of their programs enough. But there are many other important factors involved in such widespread: insufficient performance regression testing, small test workloads or the fast development frequently breaking the codebase.

Unlike in the case of functional bugs, a large percentage of performance bugs is usually discovered through code reasoning or profiling, and not through the majority of users reporting negative effects of the bugs or through regular automated checks. Performance degradations are subtle, and they tend to manifest only with considerably big workloads. In the end they are missed by the frequent regression testing and noticed only by individual users in individual cases. So, naturally, techniques to help developers reason about the performance, better test oracles or better profiling techniques are needed in order to discover these kinds of bugs early in the process. Obviously, we have to extend the developer's everyday toolbox with efficient automated performance analyses and automated detection of performance bugs.

Although some research of automated performance analysis has already been done, the currently known techniques are still far from being satisfactory. This is especially true when the analysed code works not only with simple data types such as integers, but employs complex dynamic data structures based on pointers such as lists, trees, and their various combinations or extensions.

Such data structures are commonly used in complex system code like operating system kernels, compilers, database engines, browsers and even embedded systems, whose poor performance can significantly impact the user experience. It is a well-known fact that dynamic data structures are hard to develop and can contain intricate errors which, in addition, only manifest under certain circumstances and are thus difficult to track down. Moreover, in performance critical applications, developers use even more advanced data structures such as, e.g. red-black trees, priority heaps, or lock-free linked lists, as well as various advanced programming techniques like, e.g. pointer arithmetic or block operations for performance speed up. For programs based on such techniques, even safety verification is still a challenge and works on their automated performance analysis are extremely rare.

In theory we can divide performance analysis into two main approaches — static and dynamic analysis. For an input program, the first one allows us to infer theoretically proven resource bounds; on the other hand, the latter collects performance records from one or more program runs, possibly extrapolates these data and then only estimates resource bounds, without any theoretical proof. But while both approaches have their advantages, e.g. in terms of the speed or precision, and the right time to be used, they share many challenges that must be overcome to apply them in everyday development.

One of these challenges is choosing a suitable formal theory to describe the program invariants. We need a theory that allows for a scalable analysis to be implemented on top of it and that is, at the same time, expressive enough to be able to reason about properties of advanced structures, especially their shapes or resource bounds. Commonly, researches use logics due to their great expressive power. However, with such power comes the price: great complexity of the associated decision problems, with some logics even being undecidable. In order to achieve an efficient analysis, one then has to improve the state of the art or use dedicated theories such as, e.g. separation logic [Rey02], three-valued logic [SRW02], or weak monadic second order logic with one successor (WS1S) [Büc59].

The latter, WS1S, has lots of applications not only for reasoning over the data structures [MPQ11, MQ11, ZHW$^+$14]. It is still a decidable logic, however, its decision problem lies in the NONELEMENTARY class: it lurks on the borders of decidability. So while many WS1S formulae are decidable in a reasonable time, sometimes its complexity simply strikes back. And then we have to either fight back or give up building on the WS1S at all. But, we hope we could exploit the recent advancements in automata theory, e.g. the antichain principles, to push the usability border of WS1S even further.

Another challenge is how to build such analysers. In particular, in the area of resource bounds analysis, current static resource bounds analyses are so far mostly limited to programs with integer variables only. When pointers are used in the analysed programs the analysers usually return a huge number of false negatives, not knowing the precise targets of the used pointers or the shape of the dynamic data structures being handled. They are forced to work with basic assumptions over the pointer variables, and thus they have to sacrifice soundness or precision of the approach. Programs with pointers are, however, common in practice, so this limitation is rather significant from the point of view of applicability.

Alternatively, we can give up the precision of the static analysis, focus on dynamically captured data and only settle for estimates of the program performance from concrete program runs. While one cannot guarantee how the program under analysis will perform or whether it will trigger any bug, dynamic analysis can still provide a useful insight and can be exploited, e.g. to detect performance changes or to infer statistical models of expected performance, leading to a better program understanding.

In the end, both static and dynamic analyses have their own shortcomings, nevertheless, we, researchers, should also focus our efforts on the developer experience. We should always strive to achieve a high performance bug fix ratio instead of high bug detection ratio since only that shows that our methods are applicable in practice. Every performance analysis should provide at least (i) approximate location where the bug was located, (ii) estimated severity how the performance or functionality is influenced by the bug, and (iii) detection confidence whether the bug was real or spurious. These factors greatly affect whether the developer will confirm the bug the subsequent decision whether it should be fixed. However, most importantly, if these bugs are to be fixed at all, developers have to catch them early in the development process when their mindset is still in the context of the influenced code. This can only be achieved by integrating static and dynamic analysers into the existing development workflows such as the continuous integration.

1.1. Goals of the Thesis

The aim of this thesis is to extend the current state of the art of formal analysis and verification of systems with infinite state space and with the focus on techniques based on automata. In particular, we address this goal in two distinct parts. On one hand, the thesis focuses on developing novel methods based on static analysis and, on the other hand, also on enhancing methods for deciding formal theories, that are currently used in existing methods, to enable analysis and verification of a broader range of programs.

The first goal is enhancing the current methods for deciding selected logics that were applied as an underlying formal theory for, e.g. representation of advanced data structure's invariants. In particular, the focus is put on decision procedures for the weak monadic second order logic of one successor (WS1S), which is the target of the translation of logics of Strand [MPQ11] and UABE [ZHW+14] — logics allowing expression of invariants of advanced data structures and arrays respectively. The current state-of-the-art decision procedures, however, are not efficient enough to decide more complex formulae and so authors of Strand and UABE had to find a workaround in order to apply them properly in the field of program verification. Motivated by this situation, one of the goals is thus to improve the current state of the art in WS1S decision procedures, and to make them more efficient to be usable on more complex formulae such as those of Strand or UABE.

The other goal focuses on static performance analysis for heap-manipulating programs (with the emphasis on resource bounds analysis and automatic complexity analysis). While the current state-of-the-art of the resource bounds analysis of integer programs is already quite advanced, the state of the art of performance analysis of heap-manipulating programs is much less developed. We build on results from the fields of shape analysis [HHR$^+$12, HŠRV13, HHL$^+$15a] and resource bounds analysis of integer programs [ZGSV11, SZV14, SZV17] to develop a sound analysis for verification of resource bounds of programs manipulating with advanced data structures. The key to solving this goal lies in a proper definition of so-called shape norms — numerical measures on data structures, such as the length of list or the number of elements in trees. Hence, the goal is to propose a flexible and powerful class of norms that will allow one to analyse a wide selection of data structures, such as binary trees or even skip-lists.

1.2. An Overview of the Achieved Results

We achieved encouraging results both in the resource bounds analysis and in the decision procedures for underlying theories, showing that we succeeded in all of the proposed goals. In particular, we have proposed two decision procedures for the WS1S logic, with focus on deciding formulae describing structural invariants, and a parametric performance analysis framework for static resource bounds analysis of heap-manipulating programs. Our achievements include development of three tools which we evaluated on various nontrivial examples. In the following, we will briefly describe all our results that will be discussed in detail in the rest of the thesis.

Antichain-Based Decision Procedure for the WS1S Logic. The weak monadic second-order logic of one successor is a concise yet decidable logic for describing regular properties of finite words. It is a suitable theory that one can use to specify invariants of linear structures such as linked-lists. Even though it was proven it has NONELEMENTARY worst-case complexity [Mey72], it has been successfully applied in various fields, including verification of programs with advanced data structures, mostly due to the well-known and an efficient decision procedure implemented in the MONA tool [EKM98].

We had already remarked that there were several attempts in using WS1S to decide properties of singly linked structures [MPQ11, MQ11] and arrays [ZHW$^+$14], where the underlying decision procedure of MONA failed for more complex formulae. The algorithm of MONA is based on the correspondence between formulae and automata — it takes the input formula φ and translates it to the corresponding finite automaton $\mathcal{A}_\varphi$. The problem of the unsatisfiability of the formula φ is then reduced to the problem of the emptiness of the language of the automaton $\mathcal{A}_\varphi$. However, our idea is that always fully constructing the corresponding automaton is not efficient. The NONELEMENTARY complexity of WS1S is mainly caused by quantifier alternation (i.e. alternations of existential and universal quantifications).

The traditional decision procedure converts formulae with quantifier alternations $\forall\exists$ to the form of $\neg\exists\neg\exists$. The existential quantification is then done by removing or altering some of the transitions in the corresponding automata and it may consequently introduce non-determinism in the resulting automata. The subsequent processing of the negation, however, requires a deterministic automaton as the input and so we are forced to perform the determinisation first — a costly process. Hence a long chain of alternations leads to a huge exponential blow-up. So if we explicitly construct the automaton, we have to face the worst-case complexity the complement in every subformula.

Instead, we proposed a technique based on antichain principles of [ACH+10]. We limited ourselves to formulae in the prenex-normal form, i.e. formulae of the form $\varphi = \exists X_m \neg\exists X_{m-1} \ldots \neg\exists X_2 \neg\exists X_1 : \varphi_0$, where φ_0 is a quantifier-free formula. In each step, we process the whole chain of the quantifications with i alternations on-the-fly. So, instead of working with sets of states, we work with sets of sets of ... of states of the automaton which we represent using so-called nested symbolic terms. During the state space exploration, we use generalized antichain-based pruning on all of the levels of the nested structure of symbolic terms and reduce the number of explored states. Hence, we achieve two state space reductions (i) by efficiently representing the whole chain of quantifiers and (ii) by pruning away portions of state space subsumed by other nested symbolic terms. The emptiness check is then reduced to a simple test if the intersection of nested symbolic terms representing initial and final states of $\mathcal{A}_\varphi$ is empty.

We implemented the proposed algorithm in a prototype tool called dWiNA. We have obtained encouraging results, when we were able to beat the Mona tool when processing the last alternation in the formulae obtained from [MQ11] — encodings of various invariants of singly-linked list methods. However, at this stage, we still failed for many formulae, mostly due to limitations of our approach. This work was published as a paper in the proceedings of 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) [FHLV15] and its extended version was published in the Acta Informatica journal [FHLV19].

Lazy Decision Procedure for WS1S Logic While our first proposed decision procedure brought a significant progress, it was still limited to formulae in the prenex-normal form. We generalized our antichain based technique to arbitrary formulae φ by representing the formula as a so-called language term $t_{[\varphi]}$ — a symbolic representation of the explicit automaton $\mathcal{A}_\varphi$ imitating its nested structure of subformulae. This symbolic representation allows us to interleave the phases of the explicit automaton construction and the language emptiness check (and hence the formula unsatisfiability check). Moreover, by generalizing our antichain based reduction techniques we can significantly prune out large portions of the generated state space (i.e. prune out states that are not relevant to the test). Also, by interleaving the construction and the check we can avoid building large portions of the automata explicitly (i.e. skip portions of state space that are not necessary to prove the emptiness).

Besides this generalized decision procedure, we also proposed series of heuristics and optimizations. The most notable is the usage of the so-called anti-prenexing to push quantifiers deeper in the syntax tree of the formula to minimize the extent of the state space explosion. Moreover, the main advantage of our proposed procedure is that it can be effectively combined with the explicit procedure of the Mona tool and exploit its many optimizations. We explored various combinations of these two procedures and finally proposed a heuristic which yields efficient results in practice — we use the explicit procedure of Mona (exploiting its efficient minimization of automata) on quantifier free subformulae and our on-the-fly procedure on the unprocessed rest.Some of these optimizations are not limited only to our decision procedure and can be used by other decision procedures to enhance their efficiency as well.

We extended our prototype dWiNA to a tool called Gaston and evaluated it on series of experiments again with focus on formulae describing invariants of data structures [MQ11, ZHW$^+$14]. Our tool Gaston was able to outperform both our previous approach and other recent or state-of-the-art approaches including the Mona tool. We believe that our efficient implementation opens new possibilities of using the WS1S logic to express invariants of more complex data structures. This work was published as a paper in the proceedings of 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) [FHJ$^+$17].

Parametric Framework for Resource Bounds Analysis The approach of [SZV17] presents a well-established solution for resource bounds analysis. Indeed, their implementation was able to obtain precise bounds for about 45% of loops of an extensive benchmark consisting of thousands of functions. However, it fails for many programs that we ran it on; mostly due to a missing shape analysis of the advanced data structures stored on the heap. Consequently, authors show that even with an unsound shape analysis they are able to increase the number of computed bounds to an astounding 71%. So even just using imprecise shape analysis shows a great potential.

We proposed a parametric framework combining shape analysers with resource bounds analysers to extend the state of the art in performance analysis to handle significantly more heap manipulating programs. We build on foundations of [HHL$^+$15a] implemented within the Forester tool [HHL$^+$15b] — a powerful shape analyser capable of handling complex data structures such as skip-lists — and on foundations of [SZV17] implemented within the Loopus tool [SZ10] — a light-weight resource bounds analyser capable of amortized reasoning. In brief, our approach is based on identifying so-called shape-norms — a numerical measure on data structures, such as lengths of lists or numbers of elements in trees. In our case, we define shape norms as lengths of paths described by regular expressions between two distinct points in the heap such as cells pointed by certain pointer variables or containing certain data. We derive the set of norms suitable for analysing a given program directly from the input heap-manipulating program and transform the original program to a corresponding integer representation by inferring changes of norms for each pointer instruction. Finally, we use the underlying bounds analyser to compute the resource bounds. This transformation is sound: the bounds on the generated integer program imply the bounds on the original heap-manipulating program.

We thus build on a well-established approach which takes the input heap-manipulating program and transforms it into the output integer program driven by the shape norms, which was already applied in many other works [MTLT10, AAG$^+$12, FG17]. However, while other definitions of norms are usually restricted only to a certain class of data structures such as linked lists, our proposed class of norms is expressive enough to cope with a wide variety of data structures such as singly-linked lists, binary search trees or even skip-lists. Moreover, while the current state-of-the-art approaches either are limited to an initial fixed set of norms or are dependent on user-defined predicates, we already mentioned that we derive candidate norms directly from the input program, and so provide a fully automated approach. Also, we introduce several heuristics for reduction of the set of the tracked norms. Hence, while other approaches needlessly track many irrelevant norms, we keep the resulting integer program minimal and include only those norms really necessary to compute the resource bounds.

Our approach was implemented in a tool called Ranger [FHR$^+$18b] that is an extension of the Forester tool [HHL$^+$15a]. We compared Ranger with latest related works [AAG$^+$08, AFHG15] on a series of examples either containing advanced data structures (such as sorts of linked lists or binary trees) or requiring amortized reasoning to infer precise resource bounds. To the best of our knowledge, we were the first to infer precise resource bounds for a showcase example of [Atk11] and for 2-level skiplists. These results show we have accomplished the second goal of the thesis. This work was published as a paper in the proceedings of 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI) [FHR$^+$18a].

1.3. Plan of the Thesis

The thesis is structured into two parts. The first, Part I, focuses on enhancing the decision procedures for the WS1S logic and comprises three chapters. Chapter 2 introduces the necessary theory such as the WS1S logic and its applications with focus on its usage as a formal theory for expressing invariants of data structures. Chapters 3 and 4 propose novel decision procedures, in particular the antichain based procedure and the on-the-fly decision procedure, respectively. Part II focuses on using static analysis for performance analysis, in particular on resource bounds analysis of heap-manipulating programs. We introduce this field with Chapter 5 by defining new class of shape numerical measures and a novel resource bounds analyser for heap-manipulating programs. Finally, we summarize our research and achievements in Chapter 6.

Part I.

Decision Procedures for WS1S

2. Preliminaries

Weak monadic second-order logic of one successor (WS1S) is a powerful language for reasoning about regular properties of finite words. It has, indeed, found numerous applications, ranging from software and hardware verification through controller synthesis to, e.g. computational linguistics or verification of parametric systems. Some of its more recent applications include verification of pointer programs and deciding related logics [MPQ11, MQ11, IRŠ13, CDNQ12, ZKR08] as well as model synthesis from regular specifications [HJK10]. At last, one can also apply WS1S as a formal theory to describe invariants of various linear data structures, such as singly-linked lists or arrays, in particular, we can list several recent works based on translation to WS1S logic that focus on verification of linear data structures [MPQ11, MQ11] or arrays [ZHW$^+$14].

Most of these successful applications were possible due to the well-known Mona tool [EKM98], which implements classical automata-based decision procedures for WS1S and WS2S logics (a generalization of WS1S to finite binary trees). However, the worst-case complexity of WS1S is NONELEMENTARY [Mey72], and, despite many optimizations implemented in Mona and other tools, the complexity sometimes simply strikes back. Authors that tries to translate their problems to WS1S are then forced to either find workarounds to circumvent this complexity blowup, such as in [MQ11], or they must often restrict the extent of the input of their approach and give up translating to WS1S altogether [WMK11].

However, for a logic as expressive as WS1S any further advancements in its decision procedures could improve its practical applicability as well as open new applications, e.g. for performance or resource bounds analysis. Especially if one increases their performance (at least) for cases common in practice (e.g. as in the case of SAT solving). Therefore, one of the goals of this thesis is to enhance the current state of the art of WS1S decision procedures. In particular, we will focus on deciding formulae describing properties and invariants of linear data structures (such as linked lists [MPQ11, MQ11] or arrays [ZHW$^+$14]) and aim at improving the decision time, hopefully, allowing these procedures to be more widely used in practice. Our improvements of the existing decision procedures are presented in Chapters 3 and 4. In this chapter, we introduce the WS1S logic and present the classical decision procedure.

2.1. The WS1S Logic

We first introduce the notion of *weak monadic second-order logic of one successor*. However, we will present only its minimal syntax; we refer the interested reader for the full standard syntax and a more thorough introduction to, e.g. Section 3.3 in [CDG$^+$08]. Though later we will present how to express more complex formulae on several examples.

Syntax. WS1S is a monadic second-order logic over the universe of discourse $\mathbb{N}_0$. This means WS1S supports second-order *variables*, usually denoted using upper-case letters $X, Y, \ldots$, that range over finite subsets of $\mathbb{N}_0$, e.g. $X = \{0, 3, 42\}$. Given X and Y are variables, we can defined WS1S atomic formulae as follows:

(i) $X \subseteq Y$, i.e. the standard set inclusion,

(ii) $\mathrm{Sing}(X)$, i.e. the singleton predicate,

(iii) $X = \{0\}$, i.e. X is a singleton containing 0, and

(iv) $X = Y + 1$, i.e. $X = \{x\}$ and $Y = \{y\}$ are singletons and x is the successor of y, i.e. $x = y + 1$.

More complex formulae can be built using the classical logical connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation), and the quantifier $\exists \mathbb{X}$ (existential quantification) where $\mathbb{X}$ is a finite set of variables (we write $\exists X$ if $\mathbb{X}$ is a singleton $\{X\}$). Naturally, we can extend the syntax with more atomic formulae and logical connectives, but the above are sufficient to obtain the full expressive power of the WS1S logic. However, one has to note that if we extend our procedures to support richer syntax with more atomic formulae and more logical connectives we can significantly enhance the speed of their implementations.

Further we will consider second-order variables only; a first-order variable x can be expressed using a second-order variable X restricted with the constraint $\mathrm{Sing}(X)$. Hence, we can express the existential and universal first-order quantification as $\exists x : \varphi \overset{\mathrm{def}}{=} \exists X : \mathrm{Sing}(X) \wedge \varphi$ and $\forall x : \varphi \overset{\mathrm{def}}{\iff} \forall X : \mathrm{Sing}(X) \Rightarrow \varphi$ respectively. Atomic first-order formulae of the form $x = y + 1$ can be substituted with formulae of the form $X = Y + 1$ interpreted as $X = Y + 1 \overset{\mathrm{def}}{\iff} \exists x, y : X = \{x\} \wedge Y = \{y\} \wedge x = y + 1$, and atomic formulae of the form $x = \epsilon$ are substituted with formulae of the form $X = \epsilon$ interpreted as $X = \epsilon \overset{\mathrm{def}}{\iff} \exists x : X = \{x\} \wedge x = \epsilon$, where ϵ is empty symbol. We can express the implication, equivalence and universal quantification as $\varphi \Rightarrow \psi \overset{\mathrm{def}}{\iff} \neg\varphi \vee \psi$, $\varphi \Leftrightarrow \psi \overset{\mathrm{def}}{\iff} \varphi \Rightarrow \psi \wedge \psi \Rightarrow \varphi$ and $\forall X : \varphi \overset{\mathrm{def}}{\iff} \neg \exists X : \neg\varphi$ respectively.

Semantics. A *model* of a WS1S formula $\varphi(\mathcal{X})$ with the set of free variables $\mathcal{X}$ is an assignment $\rho : \mathcal{X} \to 2^{\mathbb{N}_0}$ of the free variables $\mathcal{X}$ of φ to finite subsets of $\mathbb{N}_0$ for which the formula is *satisfied*, written $\rho \models \varphi$. Satisfaction of atomic formulae is defined as follows: (i) $\rho \models X \subseteq Y$ iff $\rho(X) \subseteq \rho(Y)$, (ii) $\rho \models \mathrm{Sing}(X)$ iff $\rho(X)$ is a singleton set, (iii) $\rho \models X = \{0\}$ iff $\rho(X) = \{0\}$, and (iv) $\rho \models X = Y + 1$ iff $\rho(X) = \{x\}, \rho(Y) = \{y\}$, and $x = y + 1$.

Satisfaction of formulae formed using logical connectives is defined as usual. A formula φ is *valid*, written $\models \varphi$, iff all assignments of its free variables to finite subsets of $\mathbb{N}_0$ are its models, and *satisfiable* if it has a model. Otherwise it is *unsatisfiable*. Observe the limitation to *finite* subsets of $\mathbb{N}_0$ (related to the adjective *weak* in the name of the logic); a WS1S formula can indeed only have finite models (although there may be infinitely many of them). W.l.o g, we will assume that each variable in a formula is quantified at most once.

Example 2.1. *In the following we will show how one can express more complex WS1S formulae using the minimal syntax and previously defined helper predicates. Equation 2.1 expresses the membership of first-order variable* x *in second-order set* Y *using only the minimal WS1S syntax. Its semantics and the main idea behind the translation are straightforward.*

$$x \in Y \overset{def}{\Leftrightarrow} \mathrm{Sing}(X) \land X \subseteq Y \tag{2.1}$$

But obviously, we can express more properties in a similar sense to extend the minimal syntax. However, note that some translations can be complex, such as Equation 2.2 expressing that first-order variable x is lesser or equal than first-order variable y.

$$x \leq y \overset{def}{\Leftrightarrow} \forall X : \Big(y \in X \land \underbrace{(\forall z \exists w : (w = z + 1 \land w \in X) \Rightarrow z \in X)}_{\psi}\Big) \Rightarrow x \in X \tag{2.2}$$

Observe the subformula ψ that restricts X *sets to only those that contains all numbers lesser than some* k*, such as* X $= \{1, 2, 3, 4, 5\}$ *for* k $= 5$ *(lets call this property prefix-closed). Given* x *and* y *are singleton sets, e.g.* x $= \{2\}$ *and* y $= \{4\}$ *and* y *is subset of every set* X *that is prefix-closed, then* x *has to be subset of* X *as well. And since this holds for all prefix-closed sets that contains* y*, then it means that* x *is lesser or equal than* y*.*

We conclude, that while these user defined predicates can be useful on the syntactical level, the direct translation to the minimal syntax (sometimes called the *flattening*) is a costly process. Observe that Equation 2.2 introduces one quantifier alternation as well as two additional variables, increasing the complexity by at least an order of magnitude. However, in implementation we can avoid this increase in complexity e.g. by exploiting the correspondence between finite automata and WS1S formulae and define a concrete atomic automaton for the predicate $x \leq y$[1].

[1]The correspondence between WS1S formulae and finite automata will be elaborated in Section 2.2, where we will define concrete atomic automata for corresponding atomic formulae and logical connectives.

2.2. Finite Automata

Both the classical decision procedure as well as our methods are based on finite automata (FA). So first, we will provide a brief preliminaries on automata theory. In WS1S decision procedures we use FAs to represent WS1S formulae and/or their various subformulae. In particular, we usually (e.g. in the case of the MONA tool) represent the entire given WS1S formula and all of its subformulae by corresponding finite automata accepting the language of their satisfying models (their encodings). But, in our methods, we use FAs to represent only some of the subformulae of given WS1S formula, in particular, only the atomic ones.

Let $\mathbb{X}$ be a set of variables. A *symbol* τ over $\mathbb{X}$ is a mapping of all variables in $\mathbb{X}$ to either 0 or 1, e.g. $\tau = \{X_1 \mapsto 0, X_2 \mapsto 1\}$ for $\mathbb{X} = \{X_1, X_2\}$. An *alphabet* over $\mathbb{X}$ is the set of all symbols over $\mathbb{X}$, denoted as $\Sigma_{\mathbb{X}}$. For any $\mathbb{X}$ (even empty) we use $\bar{0}$ to denote the symbol which maps all variables from $\mathbb{X}$ to 0, $\bar{0} \in \Sigma_{\mathbb{X}}$, the so-called *zero symbol*.

A (non-deterministic) *finite* (word) *automaton* (abbreviated as FA in the following) over a set of variables $\mathbb{X}$ and an alphabet $\Sigma_{\mathbb{X}}$ is a quadruple $\mathcal{A} = (Q, \delta, I, F)$ where Q is a finite set of states, $I \subseteq Q$ is a set of *initial* states, $F \subseteq Q$ is a set of *final* states, and $\delta \subseteq Q \times \Sigma_{\mathbb{X}} \times Q$ is a set of transitions of the form (p, τ, q) where $p, q \in Q$ and $\tau \in \Sigma_{\mathbb{X}}$. We use $p \xrightarrow{\tau} q \in \delta$ to denote that $(p, \tau, q) \in \delta$. Note that for an FA $\mathcal{A}$ over $\mathbb{X} = \emptyset$, $\mathcal{A}$ is a unary FA with the alphabet $\Sigma_{\mathbb{X}} = \{\bar{0}\}$.

A *run* r of $\mathcal{A}$ over a word $w = \tau_1\tau_2 \ldots \tau_n \in \Sigma_{\mathbb{X}}^*$ from the state $p \in Q$ to the state $s \in Q$ is a sequence of states $r = q_0 q_1 \ldots q_n \in Q^+$ such that $q_0 = p$, $q_n = s$ and for all $1 \leq i \leq n$ there is a transition $q_{i-1} \xrightarrow{\tau_i} q_i$ in δ. If $s \in F$, we say that r is an *accepting run*. We write $p \overset{w}{\Longrightarrow} s$ to denote that there exists a run from the state p to the state s over the word w. The *language* accepted by a state q is defined by $\mathcal{L}_{\mathcal{A}}(q) = \{w \mid q \overset{w}{\Longrightarrow} q_f, q_f \in F\}$; the language of a set of states $S \subseteq Q$ is defined as $\mathcal{L}_{\mathcal{A}}(S) = \bigcup_{q \in S} \mathcal{L}_{\mathcal{A}}(q)$. When it is clear which FA $\mathcal{A}$ we refer to, we only write $\mathcal{L}(q)$ or $\mathcal{L}(S)$. The language of $\mathcal{A}$ is then defined as $\mathcal{L}(\mathcal{A}) = \mathcal{L}_{\mathcal{A}}(I)$. We say that the state q accepts w and that the automaton $\mathcal{A}$ accepts w to express that $w \in \mathcal{L}_{\mathcal{A}}(q)$ and $w \in \mathcal{L}(\mathcal{A})$ respectively. We call a language $\mathcal{L} \subseteq \Sigma_{\mathbb{X}}^*$ *universal* iff $\mathcal{L} = \Sigma_{\mathbb{X}}^*$.

For a set of states $S \subseteq Q$, we define

$$\begin{aligned}
\mathit{post}[\delta,\tau](S) &= \bigcup_{s \in S} \{t \mid s \xrightarrow{\tau} t \in \delta\}, \\
\mathit{pre}[\delta,\tau](S) &= \bigcup_{s \in S} \{t \mid t \xrightarrow{\tau} s \in \delta\}, \text{ and} \\
\mathit{cpre}[\delta,\tau](S) &= \{t \mid \mathit{post}[\delta,\tau](\{t\}) \subseteq S\}.
\end{aligned}$$

For *post*, *pre* and *cpre* relations, we will omit the transition relation δ if it is clear from the context. The *post* and *pre* relations require no further comment; the *cpre* relation contains only those states that can reach only states of S and no other states (i.e. the so-called *controlled predecessors*).

The *complement* of an automaton $\mathcal{A}$ is the automaton $\mathcal{A}_{\mathcal{C}} = (2^Q, \delta_{\mathcal{C}}, \{I\}, \downarrow\{Q \setminus F\})$ where $\delta_{\mathcal{C}} = \left\{ P \xrightarrow{\tau} post_{[\delta,\tau]}(P) \;\middle|\; P \subseteq Q \right\}$, and $\downarrow\{Q \setminus F\}$ is the set of all subsets of Q that do not contain any final state of $\mathcal{A}$; this corresponds to the standard procedure that first determinizes $\mathcal{A}$ by the subset construction and then swaps its sets of final and non-final states. The language of $\mathcal{A}_{\mathcal{C}}$ is the complement of the language of $\mathcal{A}$, i.e. $\mathcal{L}(\mathcal{A}_{\mathcal{C}}) = \overline{\mathcal{L}(\mathcal{A})}$.

For a set of variables $\mathbb{X}$ and a variable X, the *projection* of X from $\mathbb{X}$, denoted as $\pi_X(\mathbb{X})$, is the set $\mathbb{X} \setminus \{X\}$. For a symbol τ, the projection of X from τ, denoted $\pi_X(\tau)$, is obtained from τ by restricting τ to the domain $\pi_X(\mathbb{X})$[2]. For a transition relation δ, the projection of X from δ, denoted as $\pi_X(\delta)$, is the transition relation $\left\{ p \xrightarrow{\pi_X(\tau)} q \mid p \xrightarrow{\tau} q \in \delta \right\}$.

A *word* over a finite alphabet Σ is a finite sequence $w = a_1 \cdots a_n$, for $n \geq 0$, of symbols from Σ. Its i-th symbol a_i is denoted by $w[i]$. For $n = 0$, the word is the empty word ϵ. A language L is a set of words over Σ. We use the standard language operators of concatenation $L.L'$ and iteration L^*. The (right) quotient of a language L w.r.t the language L' is the language $L - L' = \{u \mid \exists v \in L' : uv \in L\}$. We abuse the notation and write $L - w$ to denote $L - \{w\}$, for a word $w \in \Sigma^*$.

2.3. Deciding WS1S with Finite Automata

The classical decision procedure for WS1S logic [Büc59] (as described in Section 3.3 of [CDG+08]) is based on an automata-logic correspondence and decides the validity (or un/satisfiability) of a WS1S formula $\varphi(X_1, \ldots, X_n)$ by constructing the FA $\mathcal{A}_\varphi$ over the set of variables $\{X_1, \ldots, X_n\}$ which accepts language of all encodings of the models of φ. This automaton is built in a bottom-up manner, according to the syntactic structure of φ, starting with predefined atomic automata for literals and applying a corresponding automata operation for every logical connective and quantifier $(\wedge, \vee, \neg, \exists)$. Hence, for every sub-formula ψ of φ, the procedure will compute the automaton $\mathcal{A}_\psi$ such that $\mathcal{L}(\mathcal{A}_\psi)$ represents exactly all models of ψ, terminating with the resulting automaton $\mathcal{A}_\varphi$.

Models as words. The alphabet of $\mathcal{A}_\varphi$ consists of all symbols over the set $\mathbb{X} = \{X_1, \ldots, X_n\}$ of free variables of φ (for $a, b \in \{0, 1\}$ and $\mathbb{X} = \{X_1, X_2\}$, we use $\begin{smallmatrix} X_1: & a \\ X_2: & b \end{smallmatrix}$ to denote the symbol $\{X_1 \mapsto a, X_2 \mapsto b\}$). A word w from the language of $\mathcal{A}_\varphi$ is a sequence of these symbols, e.g. $\begin{smallmatrix} X_1: & \epsilon \\ X_2: & \epsilon \end{smallmatrix}$, $\begin{smallmatrix} X_1: & 011 \\ X_2: & 101 \end{smallmatrix}$, or $\begin{smallmatrix} X_1: & 01100 \\ X_2: & 10100 \end{smallmatrix}$. We denote the i-th symbol of w as $w[i]$, for $i \in \mathbb{N}_0$. An assignment (model) $\rho : \mathbb{X} \to 2^{\mathbb{N}_0}$ mapping free variables $\mathbb{X}$ of φ to subsets of $\mathbb{N}_0$ is encoded into a word w_ρ of symbols over $\mathbb{X}$ in the following way: w_ρ contains 1 in the j-th position of the row for X_i iff $j \in X_i$ in ρ. Formally, for every $i \in \mathbb{N}_0$ and $X_j \in \mathbb{X}$, if $i \in \rho(X_j)$, then $w_\rho[i]$ maps $X_j \mapsto 1$. On the other hand, if $i \notin \rho(X_j)$, then either $w_\rho[i]$ maps $X_j \mapsto 0$, or the length of w is smaller than or equal to i.

[2]Note there are several ways how to restrict the symbol to the domain — either by removing the track corresponding to the variable from the transitions or pump the transition relation by so-called *don't cares*, i.e. the track will contain both 0 or 1.

Notice that there exist an infinite number of encodings of ρ. The shortest one is w_ρ^s of the length $n+1$, where n is either the largest number appearing in any of the sets that is assigned to a variable of $\mathbb{X}$ in ρ, or -1 when all these sets are empty. The rest of the encodings are all those corresponding to w_ρ^s extended with an arbitrary number of $\bar{0}$ symbols appended to its end.

For example, $\begin{smallmatrix} X_1: & 0 \\ X_2: & 1 \end{smallmatrix}, \begin{smallmatrix} X_1: & 00 \\ X_2: & 10 \end{smallmatrix}, \begin{smallmatrix} X_1: & 000 \\ X_2: & 100 \end{smallmatrix}, \begin{smallmatrix} X_1: & 000\ldots0 \\ X_2: & 100\ldots0 \end{smallmatrix}$ are all valid encodings of the assignment $\rho = \{X_1 \mapsto \emptyset, X_2 \mapsto \{0\}\}$. For the soundness of the decision procedure, it is important that $\mathcal{A}_\varphi$ always accepts either all encodings of ρ or none of them. Later we will see this issue as being crucial when processing selected logical connectives. We use $\mathcal{L}(\varphi) \subseteq \Sigma_{\mathcal{X}}^*$ to denote the language of all encodings of a formula φ's models, where $\mathcal{X}$ are the free variables of φ.

For two sets $\mathcal{X}$ and $\mathcal{Y}$ of variables and any two symbols $\tau_1, \tau_2 \in \Sigma_{\mathcal{X}}$, we write $\tau_1 \sim_{\mathcal{Y}} \tau_2$ iff $\forall X \in \mathcal{X} \setminus \mathcal{Y} : \tau_1(X) = \tau_2(X)$, i.e. the two symbols differ (at most) in the values of variables in $\mathcal{Y}$. The relation $\sim_{\mathcal{Y}}$ is generalized to words such that $w_1 \sim_{\mathcal{Y}} w_2$ iff $|w_1| = |w_2|$ and $\forall 1 \leq i \leq |w_1| : w_1[i] \sim_{\mathcal{Y}} w_2[i]$. For a language $L \subseteq \Sigma_{\mathcal{X}}^*$, we define $\pi_{\mathcal{Y}}(L)$ as the language of words w that are $\sim_{\mathcal{Y}}$-equivalent with some word $w' \in L$. Seen from the point of view of encodings of sets of assignments, $\pi_{\mathcal{Y}}(L)$ encodes all assignments that may differ from those encoded by L (only) in the values of variables from $\mathcal{Y}$. If $\mathcal{Y}$ is disjoint with the free variables of φ, then $\pi_{\mathcal{Y}}(\mathcal{L}(\varphi))$ corresponds to the so-called *cylindrification* of $\mathcal{L}(\varphi)$, and if it is their subset, then $\pi_{\mathcal{Y}}(\mathcal{L}(\varphi))$ corresponds to the so-called *projection* [CDG+08]. We use π_Y to denote $\pi_{\{Y\}}$ for a variable Y.

Automata-logic connection. Now consider formulae over the set of variables $\mathbb{V}$. Let *free*(φ) be the set of free variables of φ, and let $\mathcal{L}^{\mathbb{V}}(\varphi) = \pi_{\mathbb{V} \setminus \mathit{free}(\varphi)}(\mathcal{L}(\varphi))$ be the language $\mathcal{L}(\varphi)$ cylindrified w.r.t those variables of $\mathbb{V}$ that are not free in φ. Let φ and ψ be formulae and assume that $\mathcal{L}^{\mathbb{V}}(\varphi)$ and $\mathcal{L}^{\mathbb{V}}(\psi)$ are languages of encodings of their models cylindrified w.r.t $\mathbb{V}$. Languages of formulae obtained from φ and ψ using logical connectives are defined by equations (2.3) to (2.6). Equations (2.3)-(2.5) above are straightforward: logical connectives translate to the corresponding set operators over the universe of encodings of assignments of variables in $\mathbb{V}$.

$$\mathcal{L}^{\mathbb{V}}(\varphi \vee \psi) = \mathcal{L}^{\mathbb{V}}(\varphi) \cup \mathcal{L}^{\mathbb{V}}(\psi) \tag{2.3}$$

$$\mathcal{L}^{\mathbb{V}}(\varphi \wedge \psi) = \mathcal{L}^{\mathbb{V}}(\varphi) \cap \mathcal{L}^{\mathbb{V}}(\psi) \tag{2.4}$$

$$\mathcal{L}^{\mathbb{V}}(\neg\varphi) = \Sigma_{\mathbb{V}}^* \setminus \mathcal{L}^{\mathbb{V}}(\varphi) \tag{2.5}$$

$$\mathcal{L}^{\mathbb{V}}(\exists \mathcal{X} : \varphi) = \pi_{\mathcal{X}}(\mathcal{L}^{\mathbb{V}}(\varphi)) - \bar{0}^* \tag{2.6}$$

Existential quantification $\exists \mathcal{X} : \varphi$ translates into a composition of two language transformations: *projection* and *saturation*. First, $\pi_{\mathcal{X}}$ makes the valuations of variables of $\mathcal{X}$ arbitrary, which intuitively corresponds to forgetting everything about values of variables in $\mathcal{X}$ (notice that this is a different use of $\pi_{\mathcal{X}}$ than the cylindrification since here variables of $\mathcal{X}$ *are* free variables of φ). The second step is removing suffixes of $\bar{0}$'s from the model encodings. This process is necessary since $\pi_{\mathcal{X}}(\mathcal{L}^{\mathbb{V}}(\varphi))$ might be missing some encodings of models of $\exists \mathcal{X} : \varphi$, as we outlined already.

For example, suppose that $\mathbb{V} = \{X, Y\}$ and the only model of φ is $\{X \mapsto \{0\}, Y \mapsto \{1\}\}$, yielding $\mathcal{L}^{\mathbb{V}}(\varphi) = \begin{array}{r} X: \\ Y: \end{array} \begin{array}{c} 10 \\ 01 \end{array} \begin{bmatrix} 0 \\ 0 \end{bmatrix}^*$. Then $\pi_Y(\mathcal{L}^{\mathbb{V}}(\varphi)) = \begin{array}{r} X: \\ Y: \end{array} \begin{array}{c} 10 \\ ?? \end{array} \begin{bmatrix} 0 \\ ? \end{bmatrix}^*$ does not contain the shortest encoding $\begin{array}{r} X: \\ Y: \end{array} \begin{array}{c} 1 \\ ? \end{array}$ (where each '?' denotes an arbitrary value; sometimes we refer to this value as a *don't care*) of the only model $\{X \mapsto \{0\}\}$ of $\exists Y : \varphi$. It only contains its variants with at least one $\bar{0}$ appended to it. This generally happens for models of φ where the largest number in the value of the variable Y being eliminated is larger than the maximum number found in the values of the free variables of $\exists Y : \varphi$. The role of the $-\bar{0}^*$ quotient is to include the missing encodings of models with a smaller number of trailing $\bar{0}$'s into the language; we call this process *saturation*.

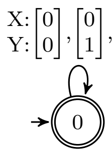
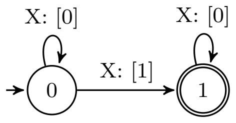
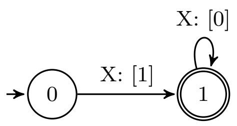
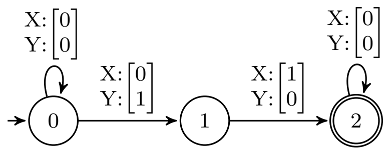
Formally, the automaton $\mathcal{A}_{\exists X : \varphi} = (Q, \pi_X(\delta), I, F^\sharp)$ is obtained from automaton $\mathcal{A}'_\varphi = (Q, \pi_X(\delta), I, F)$ by computing $F^\sharp$ from F using the fixpoint computation $F^\sharp = \mu Z \,.\, F \cup pre_{[\pi_X(\delta), \bar{0}]}(Z)$. Intuitively, the least fixpoint denotes the set of states backward-reachable from F following transitions of $\pi_X(\delta)$ labelled by $\bar{0}$.

Classical Decision Procedure. The standard approach to decide unsatisfiability of a WS1S formula φ with the set of variables $\mathbb{V}$ is to construct an automaton $\mathcal{A}_\varphi$ accepting $\mathcal{L}^{\mathbb{V}}(\varphi)$ and then check emptiness of its language. The construction starts with simple predefined automata $\mathcal{A}_\psi$ for φ's atomic formulae ψ (see Fig. 4.2 for examples of automata for selected atomic formulae and e.g. [CDG+08] for more details) accepting cylindrified languages $\mathcal{L}^{\mathbb{V}}(\psi)$ of models of ψ. These are simple regular languages. The construction then continues by inductively constructing automata $\mathcal{A}_{\varphi'}$ accepting languages $\mathcal{L}^{\mathbb{V}}(\varphi')$ of models for all other sub-formulae φ' of φ, using equations (2.3)–(2.6) above. The language operators used in the rules are implemented using standard automata-theoretic constructions (see [CDG+08]).

The procedure returns an automaton $\mathcal{A}_\varphi$ that accepts exactly all encodings of the models of φ. This means that the language of $\mathcal{A}_\varphi$ is (i) universal iff φ is valid, (ii) non-universal iff φ is invalid, (iii) empty iff φ is unsatisfiable, and (iv) non-empty iff φ is satisfiable. Notice that in the particular case of *ground* formulae (i.e. formulae without free variables), the language of $\mathcal{A}_\varphi$ is either $\mathcal{L}(\mathcal{A}_\varphi) = \{\bar{0}\}^*$ in the case φ is valid, or $\mathcal{L}(\mathcal{A}_\varphi) = \emptyset$ in the case φ is unsatisfiable.

Example 2.2. *In Table 2.1 we show definitions of selected atomic automata for the minimal syntax of the WS1S logic. For each atomic automaton we also show an example of a satisfying assignment and its encoding in the language of the corresponding automata.*

Table 2.1.: Examples of atomic DFA corresponding to atomic formulae of WS1S minimal syntax, listed with example satisfying models and their encodings in the language of automata.

Formula	$X \subseteq Y$	$Sing(X)$
Automaton		
Model	$\begin{array}{l} X \mapsto \{\ \ 2,\ \ 4\} \\ Y \mapsto \{1,2,3,4\} \end{array} \models X \subseteq Y$	$X \mapsto \{2\} \models Sing(X)$
Encoding	$\begin{array}{l} \text{X:} \\ \text{Y:} \end{array} \begin{bmatrix}0\\0\end{bmatrix} \begin{bmatrix}0\\1\end{bmatrix} \begin{bmatrix}1\\1\end{bmatrix} \begin{bmatrix}0\\1\end{bmatrix} \begin{bmatrix}1\\1\end{bmatrix} \begin{bmatrix}0\\0\end{bmatrix}^* \subseteq \mathcal{L}(\mathcal{A}_{X \subseteq Y})$	$X : [0][0][1][0]^* \subseteq \mathcal{L}(\mathcal{A}_{\text{Sing}(X)})$
Formula	$X = \{0\}$	$X = Y + 1$
Automaton		
Model	$X \mapsto \{0\} \models X = \{0\}$	$\begin{array}{l} X \mapsto \{2\} \\ Y \mapsto \{1\} \end{array} \models X = Y + 1$
Encoding	$X : [1][0]^* \subseteq \mathcal{L}(\mathcal{A}_{X=\{0\}})$	$\begin{array}{l} \text{X:} \\ \text{Y:} \end{array} \begin{bmatrix}0\\0\end{bmatrix} \begin{bmatrix}0\\1\end{bmatrix} \begin{bmatrix}1\\0\end{bmatrix} \begin{bmatrix}0\\0\end{bmatrix}^* \subseteq \mathcal{L}(\mathcal{A}_{X=Y+1})$

2.3.1. Implementations and Other Decision Procedures

We already mentioned that Mona [EKM98] is the usual tool of choice for deciding WS1S formulae. The efficiency of Mona stems from many optimizations, both higher-level (such as automata minimization, the efficient encoding of first-order variables used in models, or the use of BDDs to encode the transition relation of automata) as well as lower-level (e.g. optimizations of hash tables, etc.) [KMS02, Kla99]. Apart from Mona, there are other related tools based on the explicit automata procedure as presented in Section 2.3, such as jMosel [TWMS06] for a related logic M2L(Str)[3], which implements several distinct optimizations (such as second-order value numbering [MST10]) that allow it to outperform Mona on some benchmarks (Mona also provides an M2L(Str) interface on top of the WS1S decision procedure and so these two tools can be meaningfully compared), or the procedure using symbolic finite automata implemented within the `Automata` library of D'Antoni *et al.* in [DV14]. They both implement optimizations that allow them to outperform Mona on some benchmarks, however, none of them provides a compelling evidence of being consistently more efficient.

Recently, a couple of logic-based approaches for deciding WS1S appeared. Ganzow and Kaiser [GK10] developed a new decision procedure for the weak monadic second-order logic on inductive structures, within their tool Toss, which is even more general than WSkS[4]. Such approach completely avoids using automata; instead, it is based on Shelah's composition method. Their Toss tool is quite promising as it outperforms Mona on some of the benchmarks, however, it since it supports only minimal syntax we cannot perform a meaningful comparison on benchmarks used in practice. On the other hand, Traytel [Tra15] uses the classical decision procedure, recast in the framework of coalgebras. The work focuses on testing the equivalence of a pair of formulae, which is performed by finding a bisimulation between derivatives of the formulae. While it is shown that it can outperform Mona on some simple artificial examples, the implementation is not yet optimized enough and is easily outperformed by the rest of the tools on other benchmarks.

[3]M2L(Str) is a monadic second order logic on strings which differs from WS1S by restricting the quantification by some constant bound k. In WS1S one can interpret M2L(Str) by restricting variables with additional constraints and by introducing new variable \$, which models the bound k of the universe of discourse.

[4]Weak monadic logic with k successors. It is natural to observe that WS1S ($k = 1$) and WS2S ($k = 2$) are specialization of this generic logic. Note that, it has been proven, that fixing $k = 2$ is enough to obtain the whole expressive power of WSkS.

2.4. Invariants of Data Structures as WS1S Formulae

In the beginning of the chapter, we have listed the many applications of WS1S. We have mentioned that several authors tried to use it as an underlying theory for defining structural invariants of programs or algorithms. In particular, we have listed its two recent applications: the STRAND [MPQ11, MQ11] and UABE [ZHW$^+$14] logics used for defining invariants of common linked lists and arrays respectively. Authors of STRAND chose WS1S as a target of the translation of their higher-level logic, however, they soon got hit by the complexity blow-up while deciding more complex formulae [MPQ11]. So, as their follow-up work, they simplified the underlying decision procedure to circumvent this explosion in order to apply their method on a broader range of data structures and examples [MQ11]. The simplification had encouraging results, yet there is still lots of room for improvements and, moreover, these formulae can serve as a baseline benchmark to compare new decision procedures. On the other hand, while the authors of UABE had some successful initial application of their verification method, they failed for more complex examples, and, in the end, they gave up on extending or simplifying the UABE logic. Still, its rich expressiveness shows potential for possible application in practice.

We will use formulae of STRAND and UABE as one of the benchmarks to experimentally evaluate our decision procedures. So, in the following, we will briefly introduce both of these logics. We will focus mainly on the translation of these higher-order logics to WS1S and the structure of the resulting WS1S formulae. For more details about both of these logics refer to original papers.

2.4.1. STRAND: STRucture ANd Data Logic

The STRAND logic, which was introduced by Parlato et.al. in [MPQ11], combines reasoning over structural and data components of common dynamic data structures — a combination necessary to express invariants used for verification of many complex heap-manipulating programs. Each STRAND formula φ comprises both structural constraints, specified by a complete monadic second order logic over the heap-structure (which is decided by the underlying MSO-solver) and data constraints, specified by a data-logic over the the data fields of the elements or nodes (which is decided by the underlying SMT solver). Such a combination is powerful enough to express invariants over many data structures including nested lists, cyclic or doubly-linked lists. Note that in this thesis, we limit ourselves to the linear fragment of the STRAND logic only, and hence WS1S will be used as the underlying MSO logic for specifying structural constraints in the benchmarks we use. In order to specify invariants for tree structures, such as, e.g. threaded trees, one would have to use the WS2S logic instead. Finally, we remark that the STRAND logic also has its own limitations as one cannot express, e.g. constraints over the lengths of a list of nodes.

The idea behind encoding structural constraints of data structures in the WS1S builds on the notion of regular class of trees that serve as a skeleton or a backbone[5]. This class can be used as a source over which one can then define the concrete data structures (such as e.g. list between two concrete variables). The precise subset of nodes and edges of the backbone tree corresponds to the concrete set of nodes and edges of some data structure defined in a program. Thus, the structure on the heap can be defined as follows: the locations on heap (memory cells, variables, etc.) can be mapped to backbone nodes, and selector and field pointers can be mapped to backbone edges. So, in summary, the structural part of STRAND is directly encoded in WS1S as a formula specifying subset of nodes and edges of a backbone tree. The underlying decision procedure then interprets the resulting structural formulae on this tree-backbone.

Example 2.3. *We adapt the example of [MPQ11] in order to show how to encode some basic structural properties in WS1S. We will assume the following helper predicates, in particular, predicates* $disjoint(S_1, S_2) \overset{\text{def}}{\iff} \neg\exists z : (z \in S_1 \land z \in S_2)$ *(to denote that sets* S_1 *and* S_2 *are disjoint) and* $x \rightarrow^* y$ *(to denote that* $x = y$ *or* x *precedes* y *in the singly-linked list). We can then define a structural formula denoting that two lists between nodes* $head_1$ *and* $tail_1$ *and nodes* $head_2$ *and* $tail_2$ *are disjoint as follows:*

$$\begin{aligned}\exists S_1, S_2 :&disjoint(S_1, S_2) \land head_1 \in S_1 \land tail_1 \in S_1 \land head_2 \in S_2 \land tail_2 \in S_2\\ &\land head_1 \rightarrow^* tail_1 \land head_2 \rightarrow^* tail_2\\ &\land \big[\forall z_1 : (head_1 \rightarrow^* z_1 \land z_1 \rightarrow^* tail_1) \Rightarrow z_1 \in S_1\big]\\ &\land \big[\forall z_2 : (head_2 \rightarrow^* z_2 \land z_2 \rightarrow^* tail_2) \Rightarrow z_2 \in S_2\big]\end{aligned}$$

Authors of STRAND proposed a program verification method using their logic. Their method is based on inferring Hoare tripples of the STRAND formulae corresponding to pre-conditions, post-conditions and loop-invariants (supplied by the user) for selected program locations (e.g. loop headers) and the subsequent check of this tripple. The structural part of the decision procedure verifies that the number of minimal models is finite and determines a bound on the size of these minimal models. Finally The data part verifies that none of the constraints on data fields were violated. The main idea is that if the formula is satisfiable, then it is also satisfiable by a *data extension* of some minimal model.

We use the set of selected STRAND formulae as one of the benchmarks to evaluate both methods we proposed for deciding WS1S. In particular, we adapt formulae of [MQ11] which encode pre-conditions, post-conditions, and loop-invariants of various operations over singly-linked structures (such as bubblesort, insertion of an element into a singly-linked list, etc.). These formulae were used to evaluate a revised decision procedure introduced in [MQ11]. In this revision, the authors weakened the requirements of the structural part of the STRAND formulae yielding a more efficient decision procedure.

[5]You may be familiar with the notion of *graph types* [KS93] proposed by some of the authors of the MONA tool. These graph types build on a similar notion of tree backbone, but the logic of STRAND is more generic and subsumes the notion of graph types. The main difference is in the definition of backbone tree edges as graph types define its edges using regular expressions. For more information regarding the differences between STRAND and graph types refer to [MPQ11]

As we remarked, this was due to their failed previous attempt where deciding the structural part proved to be the bottleneck of the procedure, because of the non-elementary blowup. Note that we could not use the original benchmark of [MPQ11] since it contains so-called zero-order (boolean) variables, which are currently not efficiently supported by any of our tools. For an exact example of the used formulae from this benchmark, we refer the reader to [FHJ+16].

2.4.2. UABE: Unbounded Arrays with Bounded Elements

Another recent application of WS1S as an underlying theory for encoding and checking structural invariants is a decidable first-order unbounded array theory of bounded elements [ZHW+14] abbreviated as UABE. While the constraint of boundness may seem too restrictive, the notion of bounded elements actually accurately abstracts real programs. The value of each element in a concrete array defined in a program is always bounded by its data type, such as by an integer. However, note that, from a theoretical perspective, there is also a consequence if one focuses on unbounded elements — the theory becomes undecidable. The authors further propose a decision procedure for UABE, which is based on the translation to the WS1S logic. So, in the following, we will focus on the notion of this translation only.

The UABE logic is a single dimensional array theory, which is divided into three subtheories each describing an individual aspect of arrays — the actual array theory $\mathcal{T}_{\mathbb{A}}$, the index theory $\mathcal{T}_{\mathbb{N}}$ and the element theory $\mathcal{T}_{\mathbb{Z}_n}$. As we remarked in the beginning, the array theory $\mathcal{T}_{\mathbb{A}}$ describes arrays of unbounded lengths. This theory defines, besides common array predicates (such as array read, array write or comparison of arrays), also an additional attribute of its size[6]. The index theory $\mathcal{T}_{\mathbb{N}}$ allows one to define constraints over array indexes whose values are defined over sort of $\mathbb{N}$, with the signature (such as adding constant to index or comparison of indexes). Although one can extend this signature to allow e.g. addition of two variable indexes, authors actually show that such extension leads to undecidability. At last, the element theory $\mathcal{T}_{\mathbb{Z}_n}$, is defined over arbitrary bounded numerical sort (in particular, the authors focus on non-negative numbers) with the signature of comparison and addition. For element theory, it is important to point out its parameter n, i.e. the bound of the sort. This bound can be seen e.g. as a number of bits available to store these elements in a program, i.e. $\mathbb{Z}_n = \{0, 1, \ldots, 2^n - 1\}$.

The minimal syntax of UABE consists of atomic formulae defined over these three listed theories (arrays, indexes and elements). The more complex formulae can be constructed using usual logical connectives — negation, conjunction and existential quantification. Similarly to WS1S, UABE syntax can be extended with more logical connectives or user predicates wlog as was demonstrated in Section 2.1.

[6]Note that while this may seem unnatural in theory, in many programming languages array types do contain an attribute of array length, such as in the Java language.

The semantics of UABE formulae is defined in standard way and will not be discussed in this subsection, we will only remark several interesting points. Reading from an array with an index out-of-bounds has an undefined result; this reflects the memory access with an invalid address. Likewise addition of two elements of $\mathbb{Z}_n$ sort with result exceeding the constant 2^n has an undefined result; this reflects the arithmetic overflow. However, including such undefined results in the theory has a terrible consequence for verification usage — formulae will be unsatisfiable, when some sanity conditions are violated by these undefined values. Naturally, during verification, we are more interested whether the formula is satisfiable if there occurs no undefined values. Hence, one has to include additional restrictions in UABE formulae in order to remove these undefined values as possible valid models.

Example 2.4. *We adapt the example of [ZHW⁺14] to show* UABE *formula. The formula 2.7 models that there exists non-empty array a that is strictly sorted, i.e. the value in array with lesser index has always smaller value than those of higher index. This formula demonstrates reasoning over all of the three theories: over the array a from theory $\mathcal{T}_{\mathbb{A}}$, over indexes i, j from theory $\mathcal{T}_{\mathbb{N}}$ and over individual array elements $a[i], a[j]$ from the theory $\mathcal{T}_{\mathbb{Z}_n}$.*

$$\exists a.\big(|a| > 0 \land \forall i, j : 0 \leq i < j < |a| \Rightarrow a[i] \leq_n a[j]\big) \tag{2.7}$$

We can construct a direct translation function f from the UABE logic to WS1S. Existence of such translation results has two consequences: (i) it gives an implementation of a decision procedure and (ii) it proves the decidability of the logic (by reduction to decidable problem). The actual translation rules are based on encoding of these three underlying theories into the domain of WS1S logic, which allows only two kinds of variables: first-order (corresponding to elements of $\mathbb{N}$) and second-order (corresponding to the subsets of $\mathbb{N}$). We will need to construct translation rules for (1) variables and constants of each subtheory, (2) atomic predicates of each subtheory and (3) logical connectives forming more complex formulae.

Encoding numeric constants and index variables (i.e. the elements of $\mathbb{N}$) is straightforward as they can be expressed in the domain of WS1S already and so we do not have to encode them in any way. On contrary the elements of arrays, i.e. variables described by theory $\mathcal{T}_{\mathbb{Z}_n}$, can be seen as n-bit non-negative integers[7] represented by bitmask (e.g. 9 can be expressed as 1001 in bits). Given an array element x, we will denote the ith bit of x as x^i. We can encode x in the WS1S as the set $S_x = \{i \mid x^i = 1\}$, i.e. the set containing all indexes of bits, where bits are equal to one in the bit representation of value of x.

[7] Note that we can model even negative integers, but then we would have to dedicate one bit to be a *sign bit*.

The array theory $\mathcal{T}_{\mathbb{A}}$ builds on a similar notion — we can view an array as an unbounded sequence of its elements. Assume that the array a stores elements of $\mathbb{Z}_n$ which can be consequently represented by n bits. Then a can then be encoded as n sets $a_0, a_1, \ldots, a_{n-1}$, where each set a_i contains indexes of elements in an array, where ith bit is equal to one, i.e. $a_i = \{j \ |a[j]^i = 1\}$. Finally as we remarked, we include a variable $|a|$, which models the length of the array. So to summarize the encoding of all three sorts in the WS1S: each index variable i corresponds to a first-order variable i', each element variable x corresponds to a second-order set S_x, and each array a corresponds to n second-order sets $a_0, \ldots a_{n-1}$ encoding each bit and first-order variable $|a|$ encoding its length. We finish the description of the encoding with the following three formalized rules of translation f for variables:

$$f(a) \overset{\text{def}}{=} a_0, \ldots, a_{n-1} = \{j \mid a[j]^0 = 1\}, \ldots, \{j \mid a[j]^{n-1} = 1\} \qquad \text{(array var)}$$

$$f(i) \overset{\text{def}}{=} i \qquad \text{(index var)}$$

$$f(x) \overset{\text{def}}{=} S_x \overset{\text{def}}{=} \{i \mid x^i = 1\} \qquad \text{(element var)}$$

The actual translation of formula φ then consists of two phases: (1) translating atomic formulae according to the rule table on the leaves of syntactical structure of φ and (2) translating the logical connectives, i.e. existential quantifications, negations and conjunctions according to the nested syntactic structure of φ in bottom-up manner.

$$f(x <_n y) \overset{def}{=} \bigvee_{d=0}^{n-1} \left[\neg d \in S_x \wedge d \in S_y \wedge \bigwedge_{d'=d+1}^{n-1} (d' \in S_x \Leftrightarrow d' \in S_y)\right] \qquad \text{(element cmp)}$$

$$f(x = a[i]) \overset{def}{=} i < |a| \wedge \bigwedge_{d=0}^{n-1} d \in S_x \Leftrightarrow i \in a_d \qquad \text{(array read)}$$

$$f(a = b\{i \leftarrow x\}) \overset{def}{=} |a| = |b| \wedge i < |a| \wedge \bigwedge_{d=0}^{n-1} (i \in a_d \Leftrightarrow d \in S_x) \wedge$$
$$\wedge \forall j : \left[(j \neq i \wedge j < |a|) \Rightarrow \bigwedge_{d=0}^{n-1} (j \in a_d \Leftrightarrow j \in b_d)\right] \qquad \text{(array write)}$$

Above, three selected rules for translating atomic UABE formulae into WS1S are listed. Note that there are two ways how we can interpret testing whether bit is set in the WS1S logic — one for indexed arrays and one for element variables. In element variables, we can test if ith bit is set in variable x by testing if the index i is in the set S_x. On the other hand, we can test if ith bit is set in array element on the index j if the index j is in the set a_i. These two variants can then be used in rules e.g. for comparison of elements or for reading from an array.

$$f(\exists x : \varphi) \overset{def}{=} \exists S_x : f(\varphi) \tag{element ex}$$
$$f(\exists i : \varphi) \overset{def}{=} \exists i : f(\varphi) \tag{index ex}$$
$$f(\exists a : \varphi) \overset{def}{=} \exists |a|, a_0, \ldots, a_{n-1} : f(\varphi) \tag{array ex}$$
$$f(\neg\varphi) \overset{def}{=} \neg f(\varphi) \tag{negation}$$
$$f(\varphi \land \psi) \overset{def}{=} f(\varphi) \land f(\psi) \tag{conjunction}$$

Rules for logical connectives are straightforward. The translation function f is propagated towards leaves within the nested structure of the formulae. Quantified variables are transformed w.r.t encoding of the variables, hence existential quantification over array results into multiple variables in WS1S.

The resulting formulae naturally grow in size, but at most but no more than one order of magnitude (w.r.t the construction of the transition rules). However, one also has to note that since each array is translated to $n + 1$ variables in WS1S, the deciding time may be increased considerably in practical implementations. Such an increase is due to the usual usage of MTBDDs to represent the transition relations in corresponding automata as the efficiency of underlying libraries for MTBDDs may vary greatly.

Example 2.5. *In the following, we revise our Example 2.7 translate it to the WS1S. For an exact rules that were used refer to the original paper of [ZHW+14].*

$$\begin{aligned}&\exists |a|, a_0, a_1 : \forall i, j : 0 < i < |a| \Rightarrow \\ &\Big(j < |a| \land \big[(\neg i \in a_0 \land j \in a_0 \land i \in a_1 \Leftrightarrow j \in a_1) \lor (\neg i \in a_1 \land j \in a_1)\big]\Big)\end{aligned}$$

We use the set of selected formulae of UABE logic as another benchmark to evaluate both of the methods we proposed for deciding WS1S. In particular, we adapt formulae provided by the authors of UABE logic [ZHW+14], which encode various array properties, such as whether an array is sorted, whether any subarray of an array is sorted, or whether an array is a Fibonacci sequence. We further parametrize some of these benchmarks with a parameter k to stress test the scalability of our methods measured against the well-established procedure of Mona. For an example of the used formulae from this benchmarks refer to [FHJ+16].

3. Nested Antichains for WS1S

The classical approach for deciding WS1S, e.g. as implemented within the Mona tool and as recalled in Chapter 3, works with deterministic automata. It uses determinization extensively, and it relies on efficient minimization of deterministic automata to suppress the complexity blow-up. However, the worst-case exponential complexity of determinization often significantly harms the performance of the tool. But we believe that we can alleviate this problem by exploiting some of the recent works on efficient methods for handling non-deterministic automata—in particular, works on efficient testing of language inclusion and universality of finite automata [DR10, WDHR06, ACH$^+$10] and works on reducing the size of finite automata using simulation relations [BG00, ABH$^+$08]. These methods can handle non-deterministic automata while avoiding the determinization, and it has been shown they provide great efficiency improvements[1], e.g. in [BHH$^+$08] (abstract regular model checking) or in [HHR$^+$12] (shape analysis). We thus make a major step towards building the entire decision procedure of WS1S on non-deterministic automata using similar techniques. We propose a generalization of the antichain algorithms of [DR10] to address the main bottleneck of the automata-based decision procedure for WS1S, i.e. the source of its complexity: the elimination of alternating quantifiers, which — when implemented on the automata level — produces nondeterministic FAs, and is followed by determinisation needed to allow subsequent negations. Our proposed approach was first published in [FHLV15] and then its extended version in [FHLV19].

We repeat that the classical automata-based decision procedure translates the input WS1S formula into a finite word automaton such that its language represents all models of the formula. The automaton is built in a bottom-up manner according to the syntactic structure of the formula, starting with predefined automata for its literals (called "atomic" automata in the following) and applying a corresponding automata operation for every logical connective and quantifier ($\wedge, \vee, \neg, \exists$).

We can explain the source of the nonelementary complexity of the procedure on an example formula of the form $\varphi' = \exists X_m \forall X_{m-1} \dots \forall X_2 \exists X_1 : \varphi_0$. First, we replace universal quantifiers by negation and existential quantification, which results into the formula $\varphi = \exists X_m \neg \exists X_{m-1} \dots \neg \exists X_2 \neg \exists X_1 : \varphi_0$. The algorithm then builds a sequence of automata for the sub-formulae $\varphi_0, \varphi_0^\sharp, \dots, \varphi_{m-1}, \varphi_{m-1}^\sharp$ of φ where $\varphi_i^\sharp = \exists X_{i+1} : \varphi_i$ and $\varphi_{i+1} = \neg \varphi_i^\sharp$ for $0 \leq i < m$. Every automaton in the sequence is constructed from the previous one by applying automata operations corresponding to negation or elimination of the existential quantifier. The latter corresponds to modifying automata transitions and may potentially introduce non-determinism.

[1]Naturally, the worst-case exponential complexity of these methods is an inherent property, however, the average complexity can indeed be improved so the methods can be used in practice.

However, the typical approach of complementing an NFA, i.e., determinising it first and then switching final and nonfinal states, may result into an exponential blowup: given an automaton for ψ, the automaton for $\neg\psi$ is constructed by the classical automata-theoretic construction consisting of determinization by the subset construction followed by swapping of the sets of final and non-final states. Since the subset construction is exponential in the worst case, the worst-case complexity of the procedure on the given φ is then a tower of exponentials with one level for every quantifier alternation in φ. Note that this high computational cost cannot be avoided completely—indeed, the nonelementary complexity is an inherent property of the problem.

An overview of the proposed algorithm. Instead we propose an algorithm for processing alternating quantifiers in the prefix of a formula which avoids the explicit determinization (and hence the associated exponential blowup) of automata in the classical procedure and significantly reduces the state space explosion associated with it. Our algorithm is based on a generalization of the antichain principle used for deciding universality and language inclusion of finite automata [WDHR06, ACH$^+$10]. We generalized the antichain algorithms so that instead of processing only one level of the chain of automata, we process the whole chain of quantifications with i alternations on-the-fly. Basically this means we are working with automata states that are sets of sets of sets ... of states of the automaton representing φ_0 of the nesting depth i (this corresponds to i levels of subset construction being done on-the-fly). In our algorithm we use nested symbolic terms to represent sets of such automata states (in particular, we use nested upward and downward closed sets as we discussed in Section 3.1) and a generalized version of antichain pruning based on a notion of subsumption that descends recursively down the structure of the terms while pruning on all their levels.

However, note that our proposed nested antichain approach has its own limitations: currently we can only process a quantifier prefix of a formula, after which we return the answer to the validity query, but not an automaton representing all models of the input formula. That is, we cannot use the optimized algorithm for processing inner negations and alternating quantifiers which are not a part of the quantifier prefix or unground formulae. Naturally we wish to extend our approach to arbitrary formulae as a follow-up work, which we will discuss in the Chapter 4.

An experimental evaluation. We have implemented the proposed approach in a prototype tool called dWiNA and compared its performance with other publicly available WS1S solvers on both generated formulae and formulae obtained from various verification tasks. From our experiments, we have obtained encouraging results showing that there are cases in which dWiNA outperforms Mona as well as other recently proposed decision procedures. This shows that our approach has a great potential to handle even more complex formulae and we believe it can be pushed even further, making WS1S scale enough for new classes of applications. However, our tool still failed on many other formulae, which show that we still have much room for improvements. Indeed, we propose an improved solution based on so-called language terms in Chapter 4.

Contributions. We summarize our contributions to WS1S achieved by our first proposed approach linked with the dWiNA tool:

1. By generalization of antichain techniques, we develop a decision procedure that can efficiently process long chains of quantifiers in the given formulae.
2. We show in our experimental evaluation that we improve the state of the art of WS1S decision procedures. In particular, we report on a series of parametric families of formulae, where we outperformed the state-of-the-art approaches.

Outline. The chapter is structured into three sections. In Section 3.2, we introduce our decision procedure based on antichain principles. In particular, we outline how to compute symbolic representants of final and non-final states for automaton $\mathcal{A}_{\varphi_i}$ corresponding to the ith quantifier alternation. We briefly describe the implementation of this procedure and report on its experimental evaluation in Section 3.3. Finally, we conclude the chapter with Section 3.4 and propose possible future research directions.

3.1. Downward and Upward Closed Sets

We start by introducing the notion of downward and upward closed sets. These sets are necessary to represent the nested structure of automata corresponding to WS1S formulae in our decision procedure. We want to process the whole chain of quantifier alternations *on-the-fly*[2] — requiring to process the negation of the formula which leads to the complementation of an automaton, which, in particular, must have as an input a deterministic automaton. So, instead of costly process of explicit determinisation and construction of whole automaton we represent final (resp. non-final) states by upward (resp. downward) closed sets. In experimental evaluation of our tool dWiNA (see Section 3.3), we show that this representation yields a significant reduction in explored state space.

For a set D and a set $\mathbb{S} \subseteq 2^D$, we use $\downarrow\mathbb{S}$ to denote the *downward closure* of $\mathbb{S}$, i.e. the set $\downarrow\mathbb{S} = \{R \subseteq D \mid \exists S \in \mathbb{S} : R \subseteq S\}$, and $\uparrow\mathbb{S}$ to denote the *upward closure* of $\mathbb{S}$, i.e. the set $\uparrow\mathbb{S} = \{R \subseteq D \mid \exists S \in \mathbb{S} : R \supseteq S\}$. The set $\mathbb{S}$ is in both cases called the set of *generators* of $\uparrow\mathbb{S}$ or $\downarrow\mathbb{S}$ respectively. For instance, if $D = \{1, 2, 3\}$, the downward closure $\downarrow\{\{1, 2\}, \{3\}\} = \{\emptyset, \{1\}, \{2\}, \{1, 2\}, \{3\}\}$ and the upward closure $\uparrow\{\{1, 2\}, \{3\}\} = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3\}, \{1, 2, 3\}\}$. A set $\mathbb{S}$ is *downward closed* if it equals its downward closure, $\mathbb{S} = \downarrow\mathbb{S}$, and *upward closed* if it equals to its upward closure, $\mathbb{S} = \uparrow\mathbb{S}$. The *choice* operator $\coprod$ (sometimes also called the unordered Cartesian product) is an operator that, given a set of sets $\mathbb{D} = \{D_1, \ldots, D_n\}$, returns the set of all sets $\{d_1, \ldots, d_n\}$ obtained by taking one element d_i from every set D_i. Formally,

$$\coprod\mathbb{D} = \big\{\{d_1, \ldots, d_n\} \mid (d_1, \ldots, d_n) \in \prod_{i=1}^{n} D_i\big\} \tag{3.1}$$

where $\prod$ denotes the Cartesian product. Note that for a set D, $\coprod\{D\}$ is the set of all singleton subsets of D, i.e. $\coprod\{D\} = \{\{d\} \mid d \in D\}$. Further note that if any D_i is the empty set $\emptyset$, the result is $\coprod\mathbb{D} = \emptyset$. The following lemmata show important properties of $\coprod$.

Lemma 3.1. *Let $\mathbb{X}$ and $\mathbb{Y}$ be sets of sets. Then it holds that*

$$\uparrow\coprod\mathbb{X} \cap \uparrow\coprod\mathbb{Y} = \uparrow\coprod(\mathbb{X} \cup \mathbb{Y}). \tag{3.2}$$

Proof. From the definition of the $\coprod$ operator, it holds that

$$\begin{aligned} \uparrow\coprod\mathbb{X} &= \uparrow\big\{\{x_1, \ldots, x_n\} \,\big|\, (x_1, \ldots, x_n) \in \prod\mathbb{X}\big\} \quad \text{and} \\ \uparrow\coprod\mathbb{Y} &= \uparrow\big\{\{y_1, \ldots, y_m\} \,\big|\, (y_1, \ldots, y_m) \in \prod\mathbb{Y}\big\}. \end{aligned} \tag{3.3}$$

[2]In Section 2.3 we outlined the complexity issue with processing the quantifier alternation and remarked that these alternations are the source of WS1S huge complexity.

Notice that the intersection of a pair of upward closed sets given by their generators can be constructed by taking all pairs of generators (X, Y), s.t. X is from $\coprod\mathbb{X}$ and Y is from $\coprod\mathbb{Y}$, and constructing the set $X \cup Y$. It is easy to see that $X \cup Y$ is a generator of $\uparrow\coprod\mathbb{X} \cap \uparrow\coprod\mathbb{Y}$ and that $\uparrow\coprod\mathbb{X} \cap \uparrow\coprod\mathbb{Y}$ is generated by all such pairs, i.e. that $\uparrow\coprod\mathbb{X} \cap \uparrow\coprod\mathbb{Y}$ is equal to

$$\uparrow\big\{\{x_1, \dots, x_n\} \cup \{y_1, \dots, y_m\} \,\big|\, (x_1, \dots, x_n) \in \prod X \wedge (y_1, \dots, y_m) \in \prod Y\big\}. \tag{3.4}$$

We observe that this set can be also expressed as

$$\uparrow\big\{\{x_1, \dots, x_n, y_1, \dots, y_m\} \,\big|\, (x_1, \dots, x_n, y_1, \dots y_m) \in \prod(X \cup Y)\big\} \tag{3.5}$$

or, to conclude the proof, as $\uparrow\coprod(\mathbb{X} \cup \mathbb{Y})$. □

Lemma 3.2. *Let $\mathbb{R}$ be a set of sets. Then, it holds that*

$$\uparrow\coprod\mathbb{R} = \bigcap_{R_j \in \mathbb{R}} \uparrow\coprod\{R_j\}. \tag{3.6}$$

Proof. Because intersection and union are both associative operations and the set $\mathbb{R} = \{R_1, \dots, R_n\}$, this lemma is a simple consequence of Lemma 3.1. □

As we stated, we can use upward and downward closed sets to represent sets of states of finite automata, in particular, we will use them to represent intermediate sets of final (resp non-final) states of the *ith* sub-automaton corresponding to *ith* quantifier alternation in the formula φ. Lemmata 3.1 and 3.2 are then necessary in fixpoint computations which are used to iteratively compute these final (resp. non-final states) — the core of our procedure.

Lemma 3.3. *(Equation 3.5) Let $R_j \subseteq Q_{i-1}$ and ω be a symbol over $\pi_i(\mathbb{X})$ for $i > 0$. Then*

$$cpre_{[\delta_i,\omega]}(\uparrow\coprod\{R_j\}) = \uparrow\coprod\big\{pre_{[\delta^\sharp_{i-1},\omega]}(R_j)\big\}. \tag{3.7}$$

Proof. First, we show that the set $cpre_{[\delta_i,\omega]}(\uparrow\coprod\{R_j\})$ is upward closed. Second, we show that all elements of the set $\coprod\big\{pre_{[\delta^\sharp_{i-1},\omega]}(R_j)\big\}$ are contained in $cpre_{[\delta_i,\omega]}(\uparrow\coprod\{R_j\})$. Finally, we show that for every element T in the set $cpre_{[\delta_i,\omega]}(\uparrow\coprod\{R_j\})$ there is a smaller element S in the set $\coprod\big\{pre_{[\delta^\sharp_{i-1},\omega]}(R_j)\big\}$.

1. Proving that $cpre_{[\delta_i,\omega]}(\uparrow\coprod\{R_j\})$ is upward closed: Consider a state $S \in Q_i$ s.t. $S \in cpre_{[\delta_i,\omega]}(\uparrow\coprod\{R_j\})$. From the definition of *cpre*, it holds that

$$post_{[\delta_i,\omega]}(\{S\}) \subseteq \uparrow\coprod\{R_j\}, \tag{3.8}$$

   and from the definition of δ_i, it holds that

$$post_{[\delta_i,\omega]}(\{S\}) = \{post_{[\delta^\sharp_{i-1},\omega]}(S)\}. \tag{3.9}$$

For $T \supseteq S$, it clearly holds that

$$post_{[\delta^\sharp_{i-1},\omega]}(T) \supseteq post_{[\delta^\sharp_{i-1},\omega]}(S) \tag{3.10}$$

and, therefore, it also holds that

$$post_{[\delta_i,\omega]}(\{T\}) = \{post_{[\delta^\sharp_{i-1},\omega]}(T)\} \subseteq \uparrow\coprod\{R_j\}. \tag{3.11}$$

Therefore, $T \in cpre_{[\delta_i,\omega]}(\uparrow\coprod\{R_j\})$ and the set $cpre_{[\delta_i,\omega]}(\uparrow\coprod\{R_j\})$ is upward closed.

2. Proving that for all $S \in \coprod\left\{pre_{[\delta^\sharp_{i-1},\omega]}(R_j)\right\}$ it holds that $S \in cpre_{[\delta_i,\omega]}(\uparrow\coprod\{R_j\})$: From the properties of $\coprod$, it holds that $S = \{s\}$ is a singleton. Because $s \in pre_{[\delta^\sharp_{i-1},\omega]}(R_j)$, there is a transition $s \xrightarrow{\omega} r \in \delta^\sharp_{i-1}$ for some $r \in R_j$. Since $post_{[\delta^\sharp_{i-1},\omega]}(S) \supseteq \{r\}$, it follows from the definition of δ_i that $post_{[\delta_i,\omega]}(\{S\}) = \{T\}$ where $T \supseteq \{r\}$, and so $T \in \uparrow\coprod\{R_j\}$ and $post_{[\delta_i,\omega]}(\{S\}) \subseteq \uparrow\coprod\{R_j\}$. We use the definition of *cpre* to conclude that $S \in cpre_{[\delta_i,\omega]}(\uparrow\coprod\{R_j\})$.

3. Proving that for every $T \in cpre_{[\delta_i,\omega]}(\uparrow\coprod\{R_j\})$ there exists some element $S \in \coprod\left\{pre_{[\delta^\sharp_{i-1},\omega]}(R_j)\right\}$ such that $S \subseteq T$: From $T \in cpre_{[\delta_i,\omega]}(\uparrow\coprod\{R_j\})$ and the definition of δ_i, we have that

$$post_{[\delta_i,\omega]}(\{T\}) = \{P\} \subseteq \uparrow\coprod\{R_j\} \tag{3.12}$$

for P s.t. $post_{[\delta^\sharp_{i-1},\omega]}(T) = P$. Since $P \in \uparrow\coprod\{R_j\}$, there exists $r \in R_j \cap P$ and $t \in T$ s.t. $t \xrightarrow{\omega} r \in \delta^\sharp_{i-1}$. Because $t \in pre_{[\delta^\sharp_{i-1},\omega]}(\{r\})$, we choose $S = \{t\}$ and we are done.

□

Lemma 3.4. *(Equation 3.6) Let $R_j \subseteq Q_{i-1}$ and ω be a symbol over $\pi_i(\mathbb{X})$ for $i > 0$. Then*

$$pre_{[\delta_i,\omega]}(\downarrow\{R_j\}) = \downarrow\{cpre_{[\delta^\sharp_{i-1},\omega]}(R_j)\}. \tag{3.13}$$

Proof. First, we show that $pre_{[\delta_i,\omega]}(\downarrow\{R_j\})$ is downward closed. Second, we show that $S = cpre_{[\delta^\sharp_{i-1},\omega]}(R_j)$ is in $pre_{[\delta_i,\omega]}(\downarrow\{R_j\})$. Finally, we show that every element T in $pre_{[\delta_i,\omega]}(\downarrow\{R_j\})$ is smaller than S.

1. Proving that $pre_{[\delta_i,\omega]}(\downarrow\{R_j\})$ is downward closed: Consider a state $S' \in Q_i$ s.t. $S' \in pre_{[\delta_i,\omega]}(\downarrow\{R_j\})$. From the definitions of *pre* and δ_i, it holds that

$$post_{[\delta_i,\omega]}(\{S'\}) = \{post_{[\delta^\sharp_{i-1},\omega]}(S')\} \subseteq \downarrow\{R_j\}, \tag{3.14}$$

and, therefore, $post_{[\delta^\sharp_{i-1},\omega]}(S') \in \downarrow\{R_j\}$. For $T \subseteq S'$, it clearly holds that

$$post_{[\delta^\sharp_{i-1},\omega]}(T) \subseteq post_{[\delta^\sharp_{i-1},\omega]}(S') \tag{3.15}$$

and so it also holds that

$$post_{[\delta_i,\omega]}(\{T\}) = \{post_{[\delta^\sharp_{i-1},\omega]}(T)\} \subseteq \downarrow\{R_j\}. \tag{3.16}$$

Therefore, $T \in pre_{[\delta_i,\omega]}(\downarrow\{R_j\})$ and $pre_{[\delta_i,\omega]}(\downarrow\{R_j\})$ is downward closed.

2. Proving that $S = \mathit{cpre}_{[\delta^\sharp_{i-1},\omega]}(R_j) \in \mathit{pre}_{[\delta_i,\omega]}(\downarrow\{R_j\})$: From the definition of *cpre*, it holds that

$$\mathit{post}_{[\delta^\sharp_{i-1},\omega]}(S) = S' \subseteq R_j. \tag{3.17}$$

   Further, from the definition of δ_i, it holds that $S \xrightarrow{\omega} S' \in \delta_i$ and, therefore, $S \in \mathit{pre}_{[\delta_i,\omega]}(\downarrow\{R_j\})$.

3. Proving that for every $T \in \mathit{pre}_{[\delta_i,\omega]}(\downarrow\{R_j\})$ it holds that $T \subseteq S$: From $T \in \mathit{pre}_{[\delta_i,\omega]}(\downarrow\{R_j\})$, we have that $T \xrightarrow{\omega} P \in \delta_i$ for $P \subseteq R_j$, and, from the definition of δ_i, we have that $P = \mathit{post}_{[\delta^\sharp_{i-1},\omega]}(T)$. From $P = \mathit{post}_{[\delta^\sharp_{i-1},\omega]}(T)$ and the definition of *cpre*, it is easy to see that $T \subseteq \mathit{cpre}_{[\delta^\sharp_{i-1},\omega]}(P)$, and, moreover

$$P \subseteq R_j \quad \Longrightarrow \quad \mathit{cpre}_{[\delta^\sharp_{i-1},\omega]}(P) \subseteq \mathit{cpre}_{[\delta^\sharp_{i-1},\omega]}(R_j). \tag{3.18}$$

   Therefore, we can conclude that $T \subseteq \mathit{cpre}_{[\delta^\sharp_{i-1},\omega]}(R_j) = S$.

□

Lemmata 3.4 and 3.3 are also necessary for the fixpoint computation of final (resp non-final) states. In Section 2.3, we discussed how to handle existential quantification, which consists of two phases: *projection* (i.e. removing tracks in transition relation corresponding to the quantified variables) and *saturation* (i.e. pumping the final states of the automaton ensuring it will accept all of the encodings of valid models of formulae). Computing *cpre* and *pre* relations of the upward (resp downward) closed sets corresponds to the one step of the saturation phase, when we expand the set of final (resp non-final sets) with every state backward reachable by $\bar{0}$ symbols[3] from previously computed final (resp. non-final) states until fixpoint is reached.

[3]Recollect we pointed out there is an issue with potentially infinite extensions of encodings of valid models by chain of $\bar{0}$

3.2. Nested Antichains for Alternating Quantifiers

We now present our approach for dealing with alternating quantifiers in WS1S formulae in more details. Let us consider a ground formula φ of the form

$$\varphi = \underbrace{\neg\,\exists\mathcal{X}_m\,\neg \ldots \neg\,\exists\mathcal{X}_2 \underbrace{\neg\,\exists\mathcal{X}_1 : \varphi_0(\mathbb{X})}_{\varphi_1}}_{\varphi_m} \tag{3.19}$$

where each $\mathcal{X}_i$ is a set of variables $\{X_a, \ldots, X_b\}$, $\exists\mathcal{X}_i$ is an abbreviation for a non-empty sequence $\exists X_a \ldots \exists X_b$ of consecutive existential quantifications, and φ_0 is an arbitrary formula called the *matrix* of φ. Note that the problem of checking validity or satisfiability of a formula with free variables can be easily reduced to this form as follows: given $\mathcal{X}_f = \mathrm{free}(\varphi)$, i.e. the set of free variables of φ, we can check validity (resp. satisfiability) of an unground formula φ by checking validity of the formula $\psi = \forall\mathcal{X}_f : \varphi$ (resp. $\psi = \exists\mathcal{X}_f : \varphi$.

The classical procedure presented in Section 2.3 computes a sequence of automata $\mathcal{A}_{\varphi_0}, \mathcal{A}_{\varphi_0^\sharp}, \ldots, \mathcal{A}_{\varphi_{m-1}^\sharp}, \mathcal{A}_{\varphi_m}$ where for all $0 \leq i \leq m-1$, $\varphi_i^\sharp = \exists\mathcal{X}_{i+1} : \varphi_i$ and $\varphi_{i+1} = \neg\varphi_i^\sharp$. The φ_i's are subformulae of φ shown in (3.19) corresponding to the ith quantifier alternation. Since eliminating existential quantification on the automata level introduces non-determinism (due to the projection on the transition relation), every $\mathcal{A}_{\varphi_i^\sharp}$ may be non-deterministic. The computation of $\mathcal{A}_{\varphi_{i+1}}$ then has to involve a subset construction and becomes exponential. The worst-case complexity of eliminating the whole prefix is therefore the tower of exponentials of the height m. Even though the construction may be optimized, e.g. by minimizing every $\mathcal{A}_{\varphi_i}$ (which is implemented by MONA), the size of the generated automata can quickly become intractable.

The basic algorithm. The main idea of our algorithm is inspired by the antichain algorithms [DR10] for testing language universality of an automaton $\mathcal{A}$. In a nutshell, testing universality of $\mathcal{A}$ is testing whether in the complement $\overline{\mathcal{A}}$ of $\mathcal{A}$ (which is created by determinization via subset construction, followed by swapping final and non-final states), an initial state can reach a final state. The crucial idea of the antichain algorithms is based on the following: (i) The search for the final state can be done on the fly while constructing $\overline{\mathcal{A}}$. (ii) The sets of states that arise during the search are closed (upward or downward, depending on the variant of the algorithm). (iii) The computation can be done symbolically on the generators of these closed sets. We also noticed it is enough to keep only the extreme generators of the closed sets (maximal for downward closed, minimal for upward closed). The generators that are not extreme (we say that they are *subsumed*) can be pruned away, which, in our experience, vastly reduces the search space.

We notice that individual steps of the algorithm for constructing $\mathcal{A}_\varphi$ are very similar to testing the universality. Automaton $\mathcal{A}_{\varphi_i}$ arises by subset construction from $\mathcal{A}_{\varphi^\sharp_{i-1}}$, and to compute $\mathcal{A}_{\varphi^\sharp_i}$, it is necessary to compute the set of final states $F^\sharp_i$. Those are states backward reachable from the final states of $\mathcal{A}_{\varphi_i}$ via a subset of transitions of δ_i (those labelled by symbols projected to $\bar{0}$ by π_{i+1}). To compute $F^\sharp_i$, the antichain algorithms could be actually taken off-the-shelf and run with $\mathcal{A}_{\varphi^\sharp_{i-1}}$ in the role of the input $\mathcal{A}$ and $\mathcal{A}_{\varphi^\sharp_i}$ in the role of $\overline{\mathcal{A}}$. This approach, however, has the following two problems. First, antichain algorithms do not produce the automaton $\overline{\mathcal{A}}$ (here $\mathcal{A}_{\varphi^\sharp_i}$), but only a symbolic representation of a set of (backward) reachable states (here of $F^\sharp_i$). Since $\mathcal{A}_{\varphi^\sharp_i}$ is the input of the construction of $\mathcal{A}_{\varphi_{i+1}}$, the construction of $\mathcal{A}_\varphi$ could not continue. The other problem is that the size of the input $\mathcal{A}_{\varphi^\sharp_{i-1}}$ of the antichain algorithm is only limited by the tower of exponentials of the height $i-1$, and this might be already far out of reach.

So our main contribution is an algorithm that alleviates these two problems. We based it on a novel way of performing not only one, but all the $2m$ steps of the construction of $\mathcal{A}_\varphi$ on the fly. Moreover, we use a nested symbolic representation of sets of states and a form of nested subsumption pruning on all levels of their structure. This is achieved by a substantial refinement of the basic ideas of antichain algorithms.

3.2.1. Structure of the Algorithm

We start by explaining the top-level structure of our on-the-fly algorithm for efficient handling the quantifier alternation. Following the construction of automata described in Section 3.2, the structure of the automata sequence from the previous section, $\mathcal{A}_{\varphi_0}$,$\mathcal{A}_{\varphi^\sharp_0}$, $\ldots$, $\mathcal{A}_{\varphi^\sharp_{m-1}}$, $\mathcal{A}_{\varphi_m}$, can be described using the following recursive definition. We use $\pi_i(C)$ for any mathematical structure C to denote projection of all variables in $\mathcal{X}_1 \cup \cdots \cup \mathcal{X}_i$ from C.

Let $\mathcal{A}_{\varphi_0} = (Q_0, \delta_0, I_0, F_0)$ be an FA over $\mathbb{X}$. Then, for each $0 \leq i < m$, the FAs $\mathcal{A}_{\varphi^\sharp_i}$ and $\mathcal{A}_{\varphi_{i+1}}$ are over the alphabet $\pi_{i+1}(\mathbb{X})$ and have from the construction the following structure:

$$\begin{array}{rl|rl} \mathcal{A}_{\varphi^\sharp_i} & = (Q_i, \delta^\sharp_i, I_i, F^\sharp_i) \text{ where} & \mathcal{A}_{\varphi_{i+1}} & = (Q_{i+1}, \delta_{i+1}, I_{i+1}, F_{i+1}) \text{ where} \\ \delta^\sharp_i & = \pi_{i+1}(\delta_i) \text{ and} & \delta_{i+1} & = \left\{ R \xrightarrow{\tau} post[\delta^\sharp_i, \tau](R) \,\middle|\, R \in Q_{i+1} \right\}, \\ F^\sharp_i & = \mu Z \,.\, F_i \cup pre[\delta^\sharp_i, \bar{0}](Z). & Q_{i+1} & = 2^{Q_i}, \quad I_{i+1} = \{I_i\}, \text{ and } F_{i+1} = \Downarrow\{Q_i \setminus F^\sharp_i\}. \end{array}$$

We recall that $\mathcal{A}_{\varphi^\sharp_i}$ directly corresponds to the existential quantification of all variables in $\mathcal{X}_i$ (cf. Section 2.3), and $\mathcal{A}_{\varphi_{i+1}}$ directly corresponds to the complement of $\mathcal{A}_{\varphi^\sharp_i}$ (cf. Chapter 2).

A crucial observation behind our approach is that, because φ is ground, $\mathcal{A}_\varphi$ is an FA over an empty set of variables, and, therefore, $\mathcal{L}(\mathcal{A}_\varphi)$ is either the empty set $\emptyset$ or the set $\{\bar{0}\}^*$ (as described in Section 2.3). Therefore, we need to distinguish between these two cases only. To determine which of them holds, we do not need to explicitly construct the automaton $\mathcal{A}_\varphi$. Instead, it suffices to check whether $\mathcal{A}_\varphi$ accepts the empty string ϵ. This is equivalent to checking existence of a state that is at the same time final and initial, that is

$$\models \varphi \quad \text{iff} \quad I_m \cap F_m \neq \emptyset. \tag{3.20}$$

To compute I_m from I_0 is straightforward (it equals to $\{\{\ldots\{\{I_0\}\}\ldots\}\}$ nested m-times). In the rest of the section, we will describe how to compute F_m (in particular, its symbolic representation), and how to test whether it intersects with I_m.

The algorithm takes advantage of the fact that to represent final states, one can use their complement, the set of non-final states. For $0 \leq i \leq m$, we write N_i and $N_i^\sharp$ to denote the sets of non-final states $Q_i \setminus F_i$ of $\mathcal{A}_i$ and $Q_i \setminus F_i^\sharp$ of $\mathcal{A}_i^\sharp$ respectively. The algorithm will then instead of computing the sequence of automata $\mathcal{A}_{\varphi_0}, \mathcal{A}_{\varphi_0^\sharp}, \ldots, \mathcal{A}_{\varphi_{m-1}^\sharp}, \mathcal{A}_{\varphi_m}$ only compute the sequence $F_0, F_0^\sharp, N_1, N_1^\sharp, \ldots$ up to either F_m (if m is even) or N_m (if m is odd), which still suffices for testing the validity of φ. The algorithm starts with F_0 (which can be obtained using classical automata construction as presented in Section 2.3) and uses the following recursive equations:

$$\begin{array}{llllll} \text{(i)} & F_{i+1} & = \downarrow\{N_i^\sharp\}, & \text{(ii)} & F_i^\sharp & = \mu Z \,.\, F_i \cup \mathit{pre}_{[\delta_i^\sharp,\bar{0}]}(Z), \\ \text{(iii)} & N_{i+1} & = \uparrow\coprod\{F_i^\sharp\}, & \text{(iv)} & N_i^\sharp & = \nu Z \,.\, N_i \cap \mathit{cpre}_{[\delta_i^\sharp,\bar{0}]}(Z). \end{array} \tag{3.21}$$

Intuitively, (i) and (ii) come directly from definitions of $\mathcal{A}_i$ and $\mathcal{A}_i^\sharp$. (iii) is a dual of (i): N_{i+1} contains all subsets of Q_i that contain at least one state from $F_i^\sharp$ (cf. the definition of the $\coprod$ operator). Finally, (iv) is a dual of (ii): in the k-th iteration of the greatest fixpoint computation, the current set of states Z will contain all states that cannot reach an F_i state over $\bar{0}$ within k steps. In the next iteration, only those states of Z are kept such that all their $\bar{0}$-successors are in Z. Hence, the new value of Z is the set of states that cannot reach F_i over $\bar{0}$ in $k+1$ steps, and the computation stabilizes with the set of states that cannot reach F_i over $\bar{0}$ in any number of steps.

In the next two sections, we will show that both of the above fixpoint computations can be carried out symbolically on representatives of upward and downward closed sets. Particularly, in Sections 3.2.2 and 3.2.3, we show how the fixpoints from (ii) and (iv) can be computed symbolically, using subsets of Q_{i-1} as representatives (generators) of upward/downward closed subsets of Q_i. Section 3.2.4 explains how the above symbolic fixpoint computations can be carried out using nested terms of depth i as a symbolic representation of computed states of Q_i. Section 3.2.5 shows how to test the emptiness of $I_m \cap F_m$ on the symbolic terms, and Section 3.2.6 describes the subsumption relation used to minimize the symbolic term representation used within computations of (ii) and (iv).

3.2.2. Computing $N_i^\sharp$ on Representatives of $\uparrow\coprod\mathcal{R}$-sets

Computing $N_i^\sharp$ at each odd level of the hierarchy of automata is done by computing the greatest fixpoint of the function from Equation 3.21(iv):

$$f_{N_i^\sharp}(Z) = N_i \cap \mathit{cpre}_{[\delta_i^\sharp,\bar{0}]}(Z). \tag{3.22}$$

We will show that the whole fixpoint computation from Equation 3.21(iv) can be carried out symbolically on the representatives of Z due to the following two properties: (a) all intermediate values of Z have the form $\uparrow\coprod\mathcal{R}$, where $\mathcal{R} \subseteq Q_i$, so the sets $\mathcal{R}$ can be used as their symbolic representatives, and (b) *cpre* and $\cap$ can be computed on such a symbolic representation efficiently (as we have shown in Section 3.1).

Let us start with the computation of $\mathit{cpre}_{[\delta_i^\sharp,\tau]}(Z)$ where $\tau \in \pi_{i+1}(\bar{0})$, assuming that Z is of the form $Z = \uparrow\coprod\mathcal{R}$, represented by $\mathcal{R} = \{R_1, \dots, R_n\}$. Observe that a set of symbolic representatives $\mathcal{R}$ stands for the intersection of denotations of individual representatives, that is

$$\uparrow\coprod\mathcal{R} = \bigcap_{R_j \in \mathcal{R}} \uparrow\coprod\{R_j\}. \tag{3.23}$$

The set $\mathit{cpre}_{[\delta_i^\sharp,\tau]}(Z)$ can thus be written as the *cpre*-image $\mathit{cpre}_{[\delta_i^\sharp,\tau]}(\bigcap\mathcal{S})$ of the intersection of the elements of a set $\mathcal{S} = \big\{\uparrow\coprod\{R_1\}, \dots, \uparrow\coprod\{R_n\}\big\}$. Further, because *cpre* distributes over $\cap$, we can compute the *cpre*-image of an intersection by computing intersection of the *cpre*-images, i.e.

$$\mathit{cpre}_{[\delta_i^\sharp,\tau]}(\bigcap\mathcal{S}) = \bigcap_{S \in \mathcal{S}} \mathit{cpre}_{[\delta_i^\sharp,\tau]}(S). \tag{3.24}$$

By the definition of $\delta_i^\sharp$ (where $\delta_i^\sharp = \pi_{i+1}(\delta_i)$), the set $\mathit{cpre}_{[\delta_i^\sharp,\tau]}(S)$ can be computed using the transition relation δ_i for the price of further refining the intersection. In particular,

$$\mathit{cpre}_{[\delta_i^\sharp,\tau]}(S) = \bigcap_{\omega \in \pi_{i+1}^{-1}(\tau)} \mathit{cpre}_{[\delta_i,\omega]}(S). \tag{3.25}$$

Intuitively, $\mathit{cpre}_{[\delta_i^\sharp,\tau]}(S)$ contains states from which every transition labelled by *any* symbol that is projected to τ by π_{i+1} has its target in S.

Using (3.24), (3.25), and the fact that $Z = \bigcap\big\{\uparrow\coprod\{R_j\} \mid R_j \in \mathcal{R}\big\}$, we obtain

$$\mathit{cpre}_{[\delta_i^\sharp,\tau]}(Z) = \bigcap_{\substack{R_j \in \mathcal{R} \\ \omega \in \pi_{i+1}^{-1}(\tau)}} \mathit{cpre}_{[\delta_i,\omega]}(\uparrow\coprod\{R_j\}). \tag{3.26}$$

To compute the individual conjuncts $\mathit{cpre}_{[\delta_i,\omega]}(\uparrow\coprod\{R_j\})$, we take advantage of the special form of the operand $\uparrow\coprod\{R_j\}$ and the fact that δ_i is, by its definition (obtained from determinization via subset construction), *monotone* w.r.t. $\supseteq$. That is, if $P \xrightarrow{\omega} P' \in \delta_i$ for some $P, P' \in Q_i$, then for every $R \supseteq P$, there is $R' \supseteq P'$ s.t. $R \xrightarrow{\omega} R' \in \delta_i$. Due to the monotonicity, the $\mathit{cpre}_{[\delta_i,\omega]}$-image of an upward closed set is also upward closed. Moreover, we observe that it can be computed symbolically using *pre* on elements of its generators. Particularly, for a set $\uparrow\coprod\{R_j\}$, we get the following lemma:

Lemma 3.5. *Let $R_j \subseteq Q_{i-1}$ and ω be a symbol over $\pi_i(\mathbb{X})$ for $i > 0$. Then*

$$cpre_{[\delta_i,\omega]}(\uparrow \coprod\{R_j\}) = \uparrow \coprod \left\{pre_{[\delta^\sharp_{i-1},\omega]}(R_j)\right\}. \tag{3.27}$$

Proof. First, we show that the set $cpre_{[\delta_i,\omega]}(\uparrow \coprod\{R_j\})$ is upward closed. Second, we show that all elements of the set $\coprod \left\{pre_{[\delta^\sharp_{i-1},\omega]}(R_j)\right\}$ are contained in $cpre_{[\delta_i,\omega]}(\uparrow \coprod\{R_j\})$. Finally, we show that for every element T in the set $cpre_{[\delta_i,\omega]}(\uparrow \coprod\{R_j\})$ there is a smaller element S in the set $\coprod \left\{pre_{[\delta^\sharp_{i-1},\omega]}(R_j)\right\}$.

1. Proving that $cpre_{[\delta_i,\omega]}(\uparrow \coprod\{R_j\})$ is upward closed: Consider a state $S \in Q_i$ s.t. $S \in cpre_{[\delta_i,\omega]}(\uparrow \coprod\{R_j\})$. From the definition of *cpre*, it holds that

$$post_{[\delta_i,\omega]}(\{S\}) \subseteq \uparrow \coprod\{R_j\}, \tag{3.28}$$

and from the definition of δ_i, it holds that

$$post_{[\delta_i,\omega]}(\{S\}) = \{post_{[\delta^\sharp_{i-1},\omega]}(S)\}. \tag{3.29}$$

For $T \supseteq S$, it clearly holds that

$$post_{[\delta^\sharp_{i-1},\omega]}(T) \supseteq post_{[\delta^\sharp_{i-1},\omega]}(S) \tag{3.30}$$

and, therefore, it also holds that

$$post_{[\delta_i,\omega]}(\{T\}) = \{post_{[\delta^\sharp_{i-1},\omega]}(T)\} \subseteq \uparrow \coprod\{R_j\}. \tag{3.31}$$

Therefore, $T \in cpre_{[\delta_i,\omega]}(\uparrow \coprod\{R_j\})$ and the set $cpre_{[\delta_i,\omega]}(\uparrow \coprod\{R_j\})$ is upward closed.

2. Proving that for all of the elements $S \in \coprod \left\{pre_{[\delta^\sharp_{i-1},\omega]}(R_j)\right\}$ it holds that $S \in cpre_{[\delta_i,\omega]}(\uparrow \coprod\{R_j\})$: From the properties of $\coprod$, it holds that $S = \{s\}$ is a singleton. Because $s \in pre_{[\delta^\sharp_{i-1},\omega]}(R_j)$, there is a transition $s \xrightarrow{\omega} r \in \delta^\sharp_{i-1}$ for some $r \in R_j$. Since $post_{[\delta^\sharp_{i-1},\omega]}(S) \supseteq \{r\}$, it follows from the definition of δ_i that $post_{[\delta_i,\omega]}(\{S\}) = \{T\}$ where $T \supseteq \{r\}$, and so $T \in \uparrow \coprod\{R_j\}$ and $post_{[\delta_i,\omega]}(\{S\}) \subseteq \uparrow \coprod\{R_j\}$. We use the definition of *cpre* to conclude that $S \in cpre_{[\delta_i,\omega]}(\uparrow \coprod\{R_j\})$.

3. Proving that for every $T \in cpre_{[\delta_i,\omega]}(\uparrow \coprod\{R_j\})$ there exists some element $S \in \coprod \left\{pre_{[\delta^\sharp_{i-1},\omega]}(R_j)\right\}$ such that $S \subseteq T$: From $T \in cpre_{[\delta_i,\omega]}(\uparrow \coprod\{R_j\})$ and the definition of δ_i, we have that

$$post_{[\delta_i,\omega]}(\{T\}) = \{P\} \subseteq \uparrow \coprod\{R_j\} \tag{3.32}$$

for P s.t. $post_{[\delta^\sharp_{i-1},\omega]}(T) = P$. Since $P \in \uparrow \coprod\{R_j\}$, there exists $r \in R_j \cap P$ and $t \in T$ s.t. $t \xrightarrow{\omega} r \in \delta^\sharp_{i-1}$. Because $t \in pre_{[\delta^\sharp_{i-1},\omega]}(\{r\})$, we choose $S = \{t\}$ and we are done.

□

Intuitively, sets with *post*-images above a singleton $\{p\} \in \big\{\{p\} \mid p \in R_j\big\} = \uparrow\coprod\{R_j\}$ are those that contain at least one state $q \in Q_{i-1}$ s.t. $q \xrightarrow{\omega} p \in \delta^\sharp_{i-1}$. Combining (3.26) and Lemma 3.5 yields

$$cpre_{[\delta^\sharp_i,\tau]}(Z) = \bigcap_{\substack{R_j \in \mathcal{R} \\ \omega \in \pi^{-1}_{i+1}(\tau)}} \uparrow\coprod\big\{pre_{[\delta^\sharp_{i-1},\omega]}(R_j)\big\}. \tag{3.33}$$

By applying (3.23), we get the final formula for $cpre_{[\delta^\sharp_i,\tau]}(Z)$:

$$cpre_{[\delta^\sharp_i,\tau]}(Z = \uparrow\coprod\mathcal{R}) \;=\; \uparrow\coprod\big\{pre_{[\delta^\sharp_{i-1},\omega]}(R_j) \mid \omega \in \pi^{-1}_{i+1}(\tau), R_j \in \mathcal{R}\big\}. \tag{3.34}$$

In order to compute $f_{N^\sharp_i}(Z)$, it remains to intersect $cpre_{[\delta^\sharp_i,\bar{0}]}(Z)$, computed using (3.34), with N_i. By Equation 3.21(iii), N_i equals $\uparrow\coprod\{F^\sharp_{i-1}\}$, and, by (3.23), the intersection can be done symbolically as

$$f_{N^\sharp_i}(Z) = \uparrow\coprod\Big(\{F^\sharp_{i-1}\} \cup \big\{pre_{[\delta^\sharp_{i-1},\omega]}(R_j) \mid \omega \in \pi^{-1}_{i+1}(\bar{0}), R_j \in \mathcal{R}\big\}\Big). \tag{3.35}$$

Finally, note that a symbolic application of $f_{N^\sharp_i}$ to $Z = \uparrow\coprod\mathcal{R}$ represented as the set $\mathcal{R}$ reduces to computing *pre*-images of the elements of $\mathcal{R}$, which are then put next to each other, together with $F^\sharp_{i-1}$. The computation starts from $N_i = \uparrow\coprod\{F^\sharp_{i-1}\}$, represented by $\{F^\sharp_{i-1}\}$, and each of its steps, implemented by (3.35), preserves the form of sets $\uparrow\coprod\mathcal{R}$, represented by $\mathcal{R}$.

3.2.3. Computing $F^\sharp_i$ on Representatives of $\downarrow\mathcal{R}$-sets

Similarly as in the previous section, the computation of $F^\sharp_i$ at each even level of the automata hierarchy is performed by computing the least fixpoint of the function

$$f_{F^\sharp_i}(Z) = F_i \cup pre_{[\delta^\sharp_i,\bar{0}]}(Z). \tag{3.36}$$

We will show that the whole fixpoint computation from Equation 3.21(ii) can be again carried out symbolically due to the following two properties: (a) all intermediate values of Z are of the form $\downarrow\mathcal{R}$, where $\mathcal{R} \subseteq Q_i$, meaning that sets $\mathcal{R}$ can be used as their symbolic representatives, and (b) *pre* and $\cup$ can be computed efficiently on such a symbolic representation. The computation is a simpler analogy of the one presented in Section 3.2.2.

We start with the computation of $pre_{[\delta_i^\sharp,\tau]}(Z)$ where $\tau \in \pi_{i+1}(\mathbb{X})$, assuming that Z is of the form $\downarrow\mathcal{R}$, represented by $\mathcal{R} = \{R_1, \dots, R_n\}$. A simple analogy to (3.23) and (3.24) of Section 3.2.2 is that the union of downward closed sets is a downward closed set generated by the union of their generators, i.e.

$$\downarrow\mathcal{R} = \bigcup_{R_j \in \mathcal{R}} \downarrow\{R_j\} \tag{3.37}$$

and that *pre* distributes over the union operator, i.e.

$$pre_{[\delta_i^\sharp,\tau]}(\bigcup \mathcal{R}) = \bigcup_{R_j \in \mathcal{R}} pre_{[\delta_i^\sharp,\tau]}(\downarrow\{R_j\}). \tag{3.38}$$

An analogy of (3.25) holds too:

$$pre_{[\delta_i^\sharp,\tau]}(S) = \bigcup_{\omega \in \pi_{i+1}^{-1}(\tau)} pre_{[\delta_i,\omega]}(S). \tag{3.39}$$

Intuitively, $pre_{[\delta_i^\sharp,\tau]}(S)$ contains states from which *at least one* transition labelled by *any* symbol that is projected to τ by π_{i+1} leads to the target in S. Using (3.38), (3.39), and the fact that $Z = \bigcup \big\{\downarrow\{R_j\} \mid R_j \in \mathcal{R}\big\}$, we obtain

$$pre_{[\delta_i^\sharp,\tau]}(Z) = \bigcup_{\substack{R_j \in \mathcal{R} \\ \omega \in \pi_{i+1}^{-1}(\tau)}} pre_{[\delta_i,\omega]}(\downarrow\{R_j\}). \tag{3.40}$$

To compute individual disjuncts $pre_{[\delta_i,\omega]}(\downarrow\{R_j\})$, we take advantage of the fact that every $\downarrow\{R_j\}$ is downward closed, and that δ_i is, by its definition (determinization by subset construction), *monotone* w.r.t. $\subseteq$. That is, if $P \xrightarrow{\omega} P' \in \delta_i$ for some $P, P' \in Q_i$, then for every $R \subseteq P$, there is $R' \subseteq P'$ s.t. $R \xrightarrow{\omega} R' \in \delta_i$. Due to the monotonicity, the $pre_{[\delta_i,\omega]}$-image of a downward closed set is downward closed. Moreover, we observe that this image can be computed symbolically using *cpre* only on elements of its generators. In particular, for a set $\downarrow\{R_j\}$, we get the following lemma, which is a dual of Lemma 3.5:

Lemma 3.6. *Let $R_j \subseteq Q_{i-1}$ and ω be a symbol over $\pi_i(\mathbb{X})$ for $i > 0$. Then*

$$pre_{[\delta_i,\omega]}(\downarrow\{R_j\}) = \downarrow\{cpre_{[\delta_{i-1}^\sharp,\omega]}(R_j)\}. \tag{3.41}$$

Proof. First, we show that $pre_{[\delta_i,\omega]}(\downarrow\{R_j\})$ is downward closed. Second, we show that $S = cpre_{[\delta_{i-1}^\sharp,\omega]}(R_j)$ is in $pre_{[\delta_i,\omega]}(\downarrow\{R_j\})$. Finally, we show that every element T in $pre_{[\delta_i,\omega]}(\downarrow\{R_j\})$ is smaller than S.

1. Proving that $\mathit{pre}[\delta_i,\omega](\downarrow\{R_j\})$ is downward closed: Consider a state $S' \in Q_i$ s.t. $S' \in \mathit{pre}[\delta_i,\omega](\downarrow\{R_j\})$. From definitions of *pre* and δ_i, it holds that

$$\mathit{post}[\delta_i,\omega](\{S'\}) = \{\mathit{post}[\delta^\sharp_{i-1},\omega](S')\} \subseteq \downarrow\{R_j\}, \tag{3.42}$$

   and, therefore, $\mathit{post}[\delta^\sharp_{i-1},\omega](S') \in \downarrow\{R_j\}$. For $T \subseteq S'$, it clearly holds that

$$\mathit{post}[\delta^\sharp_{i-1},\omega](T) \subseteq \mathit{post}[\delta^\sharp_{i-1},\omega](S') \tag{3.43}$$

   and so it also holds that

$$\mathit{post}[\delta_i,\omega](\{T\}) = \{\mathit{post}[\delta^\sharp_{i-1},\omega](T)\} \subseteq \downarrow\{R_j\}. \tag{3.44}$$

   Therefore, $T \in \mathit{pre}[\delta_i,\omega](\downarrow\{R_j\})$ and $\mathit{pre}[\delta_i,\omega](\downarrow\{R_j\})$ is downward closed.

2. Proving that $S = \mathit{cpre}[\delta^\sharp_{i-1},\omega](R_j) \in \mathit{pre}[\delta_i,\omega](\downarrow\{R_j\})$: From the definition of *cpre*, it holds that

$$\mathit{post}[\delta^\sharp_{i-1},\omega](S) = S' \subseteq R_j. \tag{3.45}$$

   Further, from the definition of δ_i, it holds that $S \xrightarrow{\omega} S' \in \delta_i$ and, therefore, $S \in \mathit{pre}[\delta_i,\omega](\downarrow\{R_j\})$.

3. Proving that for every $T \in \mathit{pre}[\delta_i,\omega](\downarrow\{R_j\})$ it holds that $T \subseteq S$: From $T \in \mathit{pre}[\delta_i,\omega](\downarrow\{R_j\})$, we have that $T \xrightarrow{\omega} P \in \delta_i$ for $P \subseteq R_j$, and, from the definition of δ_i, we have that $P = \mathit{post}[\delta^\sharp_{i-1},\omega](T)$. From $P = \mathit{post}[\delta^\sharp_{i-1},\omega](T)$ and the definition of *cpre*, it is easy to see that $T \subseteq \mathit{cpre}[\delta^\sharp_{i-1},\omega](P)$, and, moreover

$$P \subseteq R_j \quad \Longrightarrow \quad \mathit{cpre}[\delta^\sharp_{i-1},\omega](P) \subseteq \mathit{cpre}[\delta^\sharp_{i-1},\omega](R_j). \tag{3.46}$$

   Therefore, we can conclude that $T \subseteq \mathit{cpre}[\delta^\sharp_{i-1},\omega](R_j) = S$.

□

Intuitively, the sets with the *post*-images below R_j are those that do not have an outgoing transition leading outside R_j. The largest such a set is $\mathit{cpre}[\delta^\sharp_{i-1},\omega](R_j)$. Combining (3.40) with Lemma 3.6 yields

$$\mathit{pre}[\delta^\sharp_i,\tau](Z) = \bigcup_{\substack{R_j\in\mathcal{R}\\ \omega\in\pi^{-1}_{i+1}(\tau)}} \downarrow\{\mathit{cpre}[\delta^\sharp_{i-1},\omega](R_j)\} \tag{3.47}$$

Using (3.37), we get the final formula for $\mathit{pre}[\delta^\sharp_i,\tau](Z)$:

$$\mathit{pre}[\delta^\sharp_i,\tau](Z = \downarrow\mathcal{R}) \;=\; \downarrow\{\mathit{cpre}[\delta^\sharp_{i-1},\omega](R_j) \mid \omega \in \pi^{-1}_{i+1}(\tau), R_j \in \mathcal{R}\}. \tag{3.48}$$

To compute $f_{F^\sharp_i}(Z)$, it remains to unite $\mathit{pre}[\delta^\sharp_i,\bar{0}](Z)$, computed using (3.48), with F_i. From Equation 3.21(i), F_i equals $\downarrow\{N^\sharp_{i-1}\}$, so the union can be done symbolically as

$$f_{F^\sharp_i}(Z) = \downarrow\Big(\{N^\sharp_{i-1}\} \cup \big\{\mathit{cpre}[\delta^\sharp_{i-1},\omega](R_j) \mid \omega \in \pi^{-1}_{i+1}(\bar{0}), R_j \in \mathcal{R}\big\}\Big). \tag{3.49}$$

Therefore, a symbolic application of $f_{F_i^\sharp}$ to $Z = \downarrow\mathcal{R}$ represented using the set $\mathcal{R}$ reduces to computing *cpre*-images of elements of $\mathcal{R}$, which are put next to each other, together with $N_{i-1}^\sharp$. The computation starts from $F_i = \downarrow\{N_{i-1}^\sharp\}$, represented by $\{N_{i-1}^\sharp\}$, and each of its steps, implemented by (3.49), preserves the form of sets $\downarrow\mathcal{R}$, represented by $\mathcal{R}$.

3.2.4. Computation of $F_i^\sharp$ and $N_i^\sharp$ on Symbolic Terms

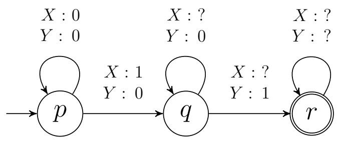
In Sections 3.2.2 and 3.2.3 we have shown how sets of states arising within the fixpoint computations from Equations 3.21(ii) and 3.21(iv) can be represented symbolically using representatives that are sets of states of the lower level (that represent either final or non-final states of previous automaton in the hierarchy). Again, we will represent these sets of states of the lower level only symbolically. When we compute the fixpoint of level i, we will work with nested symbolic representation of states of depth i. Particularly, sets of states of Q_k, for $0 \leq k \leq i$, are represented by *terms of level k* where a term of level 0 is a subset of Q_0, a term of level $2j+1$, for $j \geq 0$, is of the form $\uparrow\coprod\{t_1, \ldots, t_n\}$ where $t_1, \ldots, t_n$ are terms of level $2j$, and a term of level $2j$, for $j > 0$, is of the form $\downarrow\{t_1, \ldots, t_n\}$ where $t_1, \ldots, t_n$ are terms of level $2j-1$.

The computation of *cpre* and $f_{N_{2j+1}^\sharp}$ on a term of level $2j+1$ and computation of *pre* and $f_{F_{2j}^\sharp}$ on a term of level $2j$ then becomes a recursive procedure that descends via the structure of the terms and produces again a term of level $2j+1$ or $2j$ respectively. In the case of *cpre* and $f_{N_{2j+1}^\sharp}$ called on a term of level $2j+1$, Equation (3.34) reduces the computation to a computation of *pre* on its sub-terms of level $2j$, which is again reduced by (3.48) to a computation of *cpre* on terms of level $2j-1$, and so on until the bottom level where the algorithm computes *pre* on the terms of level 0 (subsets of Q_0). The case of *pre* and $f_{F_{2j}^\sharp}$ called on a term of level $2j$ is symmetrical.

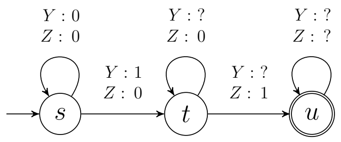
Example 3.1. *We will demonstrate the run of our algorithm on the following example formula:*

$$\varphi \equiv \underbrace{\neg\exists X\neg\exists Y\underbrace{\neg\underbrace{\exists Z : \underbrace{X < Y \land Y < Z}_{\varphi_0}}_{\varphi_0^\sharp}}_{\varphi_1}}_{\substack{\iddots \\ \varphi_3}}$$

Note that we extended the minimal syntax introduced in Section 2 with two additional atomic predicates and one additional logical connective (added to easily obtain automata suitable for the demonstration of our algorithm).



(a) $\mathcal{A}_{X<Y}$



(b) $\mathcal{A}_{Y<Z}$

Figure 3.1.: Atomic automata $\mathcal{A}_{X<Y}$ and $\mathcal{A}_{Y<Z}$

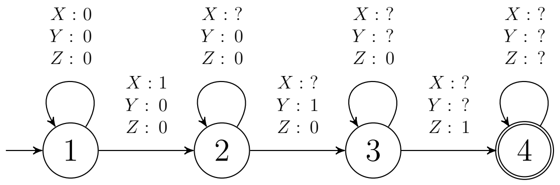


Figure 3.2.: Automaton $\mathcal{A}_0$ for the formula $\varphi_0 \equiv X < Y \land Y < Z$

The semantics of the atomic formula $X < Y$ is defined as

$$\begin{aligned} X < Y \quad \equiv \quad & \Big(\exists x : x \in X \land \forall y : y \in Y \land \exists W : \\ & \quad (\exists u : u \in W \land y = u + 1) \land \\ & \quad \big(\forall w : w \in W \land (\exists w' : w' \in W \land w = w' + 1) \lor w = x\big)\Big) \\ & \land \exists y' : y' \in Y, \end{aligned} \tag{3.50}$$

where we use the first-order variable quantification in the standard meaning. Informally, $X < Y$ denotes that both X and Y are non-empty and that the least element of X is strictly smaller than any element of Y.

We build the base automaton $\mathcal{A}_{\varphi_0}$ corresponding to the base formula $\varphi_0 \equiv X < Y \land Y < Z$ by (i) cylindrification of the atomic automata $\mathcal{A}_{X<Y}$ and $\mathcal{A}_{Y<Z}$ depicted in Figures 3.1(a) and 3.1(b), respectively, and by (ii) constructing the intersection automaton $\mathcal{A}_0 = \mathcal{A}_{X<Y} \cap \mathcal{A}_{Y<Z}$. The minimal non-deterministic automaton $\mathcal{A}_0$ is depicted in Figure 3.2. The symbol ? denotes that the value on the given track can contain both 0 or 1.

Note that, in the example, we encode values of first-order variables by the first occurrence of 1 in the given variable track representing the value of first-order variable, i.e. $x:$ 00100… , $x:$ 00101… , $x:$ 00111 or $x:$ 00110… are all valid encodings of the mapping $\{x \mapsto 2\}$. Moreover, we remark that both our tools and the Mona *tool adapt this encoding in the practice.*

Recall that our method decides the validity of φ by symbolically computing the sequence of sets $F_0^\sharp, N_1, N_1^\sharp, F_2, F_2^\sharp, N_3$, corresponding to the sequence of automata $\mathcal{A}_{\varphi_0^\sharp}$, $\mathcal{A}_{\varphi_1}$, $\mathcal{A}_{\varphi_1^\sharp}$, $\mathcal{A}_{\varphi_2}$, $\mathcal{A}_{\varphi_2^\sharp}$, $\mathcal{A}_{\varphi_3}$, with each of the sets represented using a symbolic term, and then finally checks whether $I_3 \cap N_3 \neq \emptyset$.

Let us show how this sequence is computed using our approach. Once we have constructed the base automaton, we first process the existential quantification of the variable Z, i.e. the subformula $\varphi_0^\sharp \equiv \exists Z : \varphi_0$. The first set in the sequence, $F_0^\sharp$, is obtained using a fixpoint computation given by Equation 3.21(ii), that is,

$$F_0^\sharp = \mu W \,.\, F_0 \cup pre_{[\delta_0^\sharp, \bar{0}]}(W).$$

This computation returns the set of states backward-reachable from F_0 via $\bar{0}$ transitions of $\delta_0^\sharp$. Here, the zero symbol $\bar{0}$ corresponds to the mapping $\begin{smallmatrix} X: & 0 \\ Y: & 0 \end{smallmatrix}$ of the free variables of the subformula $\varphi_0^\sharp$. The set F_0 of states of the base automaton $\mathcal{A}_0$, from which the computation starts, equals to $\{4\}$. Since we are processing $\exists Z$ in the formula, the transition relation $\delta_0^\sharp$ can be obtained by removing the track corresponding to the variable Z from δ_0. For instance, from the transition $1 \xrightarrow{\begin{smallmatrix} X: & 1 \\ Y: & 0 \\ Z: & 0 \end{smallmatrix}} 2$, we obtain the transition $1 \xrightarrow{\begin{smallmatrix} X: & 1 \\ Y: & 0 \end{smallmatrix}} 2$. However, according to (3.40), instead of removing the track, our algorithm rather computes the predecessors on the original transition relation δ_0 according to symbols where the value in the concerned Z-track is arbitrary. The set of such symbols is obtained using the inverse operation of projection. In particular, the inverse operation of projection $\pi_{[Z]}^{-1}(\begin{smallmatrix} X: & 0 \\ Y: & 0 \end{smallmatrix})$, which is used in the fixpoint computation of $F_0^\sharp$, equals to the set $\left\{ \begin{smallmatrix} X: & 0 \\ Y: & 0 \\ Z: & 0 \end{smallmatrix}, \begin{smallmatrix} X: & 0 \\ Y: & 0 \\ Z: & 1 \end{smallmatrix} \right\}$. The fixpoint computation is then carried out as follows:

$$\begin{aligned}
F_0^\sharp &= F_0 \cup pre\big[\delta_0^\sharp, \begin{smallmatrix} X: & 0 \\ Y: & 0 \end{smallmatrix}\big](F_0) \cup pre^2\big[\delta_0^\sharp, \begin{smallmatrix} X: & 0 \\ Y: & 0 \end{smallmatrix}\big](F_0) \cup \cdots \\
&= F_0 \cup \Big(\bigcup_{\substack{q \in F_0 = \{4\} \\ \omega \in \pi_{[Z]}^{-1}(\begin{smallmatrix} X: & 0 \\ Y: & 0 \end{smallmatrix})}} pre_{[\delta_0, \omega]}(q) \Big) \cup \cdots && \text{[by (3.40)]} \\
&= F_0 \cup \left(pre\left[\delta_0, \begin{smallmatrix} X: & 0 \\ Y: & 0 \\ Z: & 0 \end{smallmatrix}\right](4) \cup pre\left[\delta_0, \begin{smallmatrix} X: & 0 \\ Y: & 0 \\ Z: & 1 \end{smallmatrix}\right](4) \right) \cup \cdots \\
&= \{4\} \cup (\{3,4\} \cup \{4\}) \cup \cdots
\end{aligned}$$

After two iterations, the fixpoint is fully computed, yielding the term

$$t_{[F_0^\sharp]} = F_0^\sharp = \{3,4\}.$$

Next, we have to process the negation in the subformula $\varphi_1 \equiv \neg\exists Z : \varphi_0$, which leads to the computation of the term $t_{[N_1]}$ using Equation 3.21(iii), yielding the term

$$t_{[N_1]} = \uparrow\coprod\{F_0^\sharp\} = \uparrow\coprod\big\{\{3,4\}\big\}.$$

The algorithm continues by computing the term for the set of states $N_1^\sharp$, corresponding to the subformula $\varphi_1^\sharp \equiv \exists Y : \varphi_1$, which implies a need to process another quantifier level (namely, that of variable Y). Similarly to the previous computation, the transition relation $\delta_1^\sharp$ can be obtained by removing the track corresponding to the variable Y. This means that the fixpoint computation needs to compute cpre with the symbol $\bar{0}$ that now corresponds to the symbol $\begin{smallmatrix} X: & 0 \end{smallmatrix}$. Instead of that, however, a computation over δ_1 with symbols with arbitrary values of Y will be used. In particular, the set of such symbols will be obtained by (3.26) using the inverse projection of Y, which yields the set $\pi_{[Y]}^{-1}(\begin{smallmatrix} X: & 0 \end{smallmatrix}) = \{\begin{smallmatrix} X: & 0 \\ Y: & 0 \end{smallmatrix}, \begin{smallmatrix} X: & 0 \\ Y: & 1 \end{smallmatrix}\}$. More concretely, the computation of $N_1^\sharp$ is performed according to Equation 3.21(iv) as follows:

$$N_1^\sharp = \nu W \,.\, N_1 \cap cpre_{[\delta_1^\sharp, \bar{0}]}(W).$$

To compute the above, Equation (3.35) is used to transform the problem of computing the $cpre_{[\delta_1,\omega']}$-image of a term into a computation of a series of $pre_{[\delta_0^\sharp,\omega]}$-images of its sub-terms, which is carried out using (3.49) in the same way as when computing $t_{[F_0^\sharp]}$, resulting into the following fixpoint computation:

$$\begin{aligned}
N_1^\sharp &= N_1 \cap cpre\Big[\delta_1^\sharp, \begin{smallmatrix} X: & 0 \end{smallmatrix}\Big](N_1) \cap cpre^2\Big[\delta_1^\sharp, \begin{smallmatrix} X: & 0 \end{smallmatrix}\Big](N_1) \cap \cdots \\
&= N_1 \cap \Big(\bigcap_{\substack{Q \in N_1 \\ \omega \in \pi_{[Y]}^{-1}(\begin{smallmatrix} X: & 0 \end{smallmatrix})}} cpre_{[\delta_1,\omega]}(Q) \Big) \cap \cdots && \text{[by (3.26)]} \\
&= N_1 \cap \Big(cpre\Big[\delta_1, \begin{smallmatrix} X: & 0 \\ Y: & 0 \end{smallmatrix}\Big](\uparrow\coprod\{\{3,4\}\}) \cap cpre\Big[\delta_1, \begin{smallmatrix} X: & 0 \\ Y: & 1 \end{smallmatrix}\Big](\uparrow\coprod\{\{3,4\}\}) \Big) \cap \cdots \\
&= N_1 \cap \Big(\uparrow\coprod\Big\{pre\Big[\delta_0^\sharp, \begin{smallmatrix} X: & 0 \\ Y: & 0 \end{smallmatrix}\Big](\{3,4\})\Big\} \cap \uparrow\coprod\Big\{pre\Big[\delta_0^\sharp, \begin{smallmatrix} X: & 0 \\ Y: & 1 \end{smallmatrix}\Big](\{3,4\})\Big\} \Big) \cap \cdots \\
& && \text{[by Lemma 3.5]} \\
&= N_1 \cap \Big(\uparrow\coprod\Big\{ \bigcup_{\substack{q \in \{3,4\} \\ \omega \in \pi_{[Z]}^{-1}(\begin{smallmatrix} X: & 0 \\ Y: & 0 \end{smallmatrix})}} pre_{[\delta_0,\omega]}(q) \Big\} \cap \uparrow\coprod\Big\{ \bigcup_{\substack{q \in \{3,4\} \\ \omega \in \pi_{[Z]}^{-1}(\begin{smallmatrix} X: & 0 \\ Y: & 1 \end{smallmatrix})}} pre_{[\delta_0,\omega]}(q) \Big\} \Big) \cap \cdots && \text{[by (3.40)]} \\
&= N_1 \cap \Big(\uparrow\coprod\Big\{ pre\left[\delta_0, \begin{smallmatrix} X: & 0 \\ Y: & 0 \\ Z: & 0 \end{smallmatrix}\right](3) \cup pre\left[\delta_0, \begin{smallmatrix} X: & 0 \\ Y: & 0 \\ Z: & 1 \end{smallmatrix}\right](3) \cup \\
&\qquad\qquad pre\left[\delta_0, \begin{smallmatrix} X: & 0 \\ Y: & 0 \\ Z: & 0 \end{smallmatrix}\right](4) \cup pre\left[\delta_0, \begin{smallmatrix} X: & 0 \\ Y: & 0 \\ Z: & 1 \end{smallmatrix}\right](4) \Big\} \cap \\
&\qquad \uparrow\coprod\Big\{ pre\left[\delta_0, \begin{smallmatrix} X: & 0 \\ Y: & 1 \\ Z: & 0 \end{smallmatrix}\right](3) \cup pre\left[\delta_0, \begin{smallmatrix} X: & 0 \\ Y: & 1 \\ Z: & 1 \end{smallmatrix}\right](3) \cup \\
&\qquad\qquad pre\left[\delta_0, \begin{smallmatrix} X: & 0 \\ Y: & 1 \\ Z: & 0 \end{smallmatrix}\right](4) \cup pre\left[\delta_0, \begin{smallmatrix} X: & 0 \\ Y: & 1 \\ Z: & 1 \end{smallmatrix}\right](4) \Big\} \Big) \cap \cdots \\
&= \uparrow\coprod\{\{3,4\}\} \cap (\uparrow\coprod\{\{3,4\}\} \cap \uparrow\coprod\{\{2,3,4\}\}) \cap \cdots \\
&= \uparrow\coprod\{\{3,4\}\} \cap (\uparrow\coprod\{\{3,4\} \cup \{2,3,4\}\}) \cap \cdots && \text{[by Lemma 3.1]} \\
&= \uparrow\coprod\{\{3,4\}\} \cap \uparrow\coprod\{\{2,3,4\}\} \cap \cdots
\end{aligned}$$

Note that in implementation we do not have to compute the term pre$\left[\delta_0^\sharp, \begin{smallmatrix} X: & 0 \\ Y: & 0 \end{smallmatrix}\right](\{3,4\})$ *as it was already computed in the previous iteration of the algorithm, and thus we can use caching of intermediate results to obtain an even more efficient decision procedure. We end up with the term*

$$t_{[N_1^\sharp]} = \uparrow\coprod\big\{\{3,4\},\{2,3,4\}\big\}.$$

We continue with processing of the second negation by computing the term corresponding to the set F_2 *of automaton* A_{φ_2} *for the subformula* $\varphi_2 \equiv \neg\exists Y : \varphi_1$ *using Equation 3.21(i) to obtain the term*

$$t_{[F_2]} = \downarrow\{N_1^\sharp\} = \downarrow\Big\{\uparrow\coprod\big\{\{3,4\},\{2,3,4\}\big\}\Big\}.$$

Next, we process the last quantifier corresponding to the formula $\varphi_2^\sharp \equiv \exists X : \varphi_2$*. The symbolic fixpoint computation of* $F_2^\sharp$ *from Equation 3.21(ii) then starts from* F_2 *and uses an iterative application of* $pre_{[\delta_2^\sharp,\bar{0}]}$ *according to the equation*

$$F_2^\sharp = \mu W \,.\, F_2 \cup pre_{[\delta_2^\sharp,\bar{0}]}(W).$$

Note that, since in $\varphi_2^\sharp$*, all of the variables are projected away, the zero symbol* $\bar{0}$ *now corresponds to the mapping* $\emptyset$ *of the empty set of free variables to the set* $\{0,1\}$*.The inverse projection of the symbol* $\bar{0}$ *is then the set* $\pi_{[X]}^{-1}(\emptyset) = \{\ X:\ 0\ ,\ X:\ 1\ \}$*. The fixpoint computation proceeds similarly to the computation of* $t_{[F_0^\sharp]}$*. Using (3.48), we transform the computation of the image of* $pre_{[\delta_2^\sharp,\omega'']}$ *into the computation of a series of* $cpre_{[\delta_1^\sharp,\omega']}$*-images of the sub-terms of* $t_{[N_1^\sharp]}$*. These are in turn transformed by (3.34) into a computation of a series of* $pre_{[\delta_0^\sharp,\omega]}$*-images of sub-sub-terms of* $t_{[F_0^\sharp]}$*, i.e. subsets of* Q_0*. For our example, this yields a fixpoint computation analogous to the previous computation of the* $t_{[F_0^\sharp]}$*, resulting in the term*

$$t_{[F_2^\sharp]} = \downarrow\Big\{\uparrow\coprod\big\{\{3,4\},\{2,3,4\}\big\},\uparrow\coprod\big\{\{3,4\},\{2,3,4\},\{1,2,3,4\}\big\}\Big\}.$$

Finally, using Equation 3.21(iii), we process the last negation corresponding to the formula $\varphi \equiv \varphi_3 \equiv \neg\exists X : \varphi_2$*, which yields the final term representing* N_3*, namely,*

$$t_{[N_3]} = \uparrow\coprod\bigg\{\downarrow\Big\{\uparrow\coprod\big\{\{3,4\},\{2,3,4\}\big\},\uparrow\coprod\big\{\{3,4\},\{2,3,4\},\{1,2,3,4\}\big\}\Big\}\bigg\}.$$

Now, it remains to check whether $I_3 \cap F_3 \neq \emptyset$ *using the computed term* $t_{[N_3]}$*. We will show how to evaluate this intersection in the next section.*

3.2.5. Testing $I_m \cap F_m \neq \emptyset$ on Symbolic Terms

Due to the special form of the set I_m (every I_i, where $1 \leq i \leq m$, is the singleton set $\{I_{i-1}\}$, cf. Section 3.2.1), the test $I_m \cap F_m \neq \emptyset$ can be done efficiently over the symbolic terms representing F_m. Since $I_m = \{I_{m-1}\}$ is a singleton set, testing $I_m \cap F_m \neq \emptyset$ is equivalent to testing $I_{m-1} \in F_m$. But, if m is odd, our approach computes the symbolic representation of N_m instead of F_m. However, obviously, since N_m is the complement of F_m, it simply holds that $I_{m-1} \in F_m \iff I_{m-1} \notin N_m$. Our way of testing $I_{m-1} \in {\downarrow}\mathbb{S}$ on a symbolic representation of the set ${\downarrow}\mathbb{S}$ of level m is based on these following equations:

$$\{q\} \in {\downarrow}\mathbb{S} \iff \exists S \in \mathbb{S} : q \in S \tag{3.56}$$

$$\{q\} \in {\uparrow}\coprod\mathbb{S} \iff \forall S \in \mathbb{S} : q \in S \tag{3.57}$$

and, for $i = 0$,

$$I_0 \in {\uparrow}\coprod\mathbb{S} \iff \forall S \in \mathbb{S} : I_0 \cap S \neq \emptyset. \tag{3.58}$$

Given a symbolic term $t[R]_m$ of level m representing a set $R_m \subseteq Q_m$, testing the emptiness of $I_m \cap R_m$ or $I_m \subseteq R_m$ can be done over $t[R_m]$ by a recursive procedure that descends along the structure of $t[R_m]$ using (3.56) and (3.57), essentially generating an AND-OR tree, terminating the descent by an application of (3.58).

Example 3.2. *To finish Example 3.1, we need to test whether $I_3 \cap F_3 = \emptyset$. This is equivalent to checking whether $I_3 \subseteq N_3$, i.e., whether $\{\{\{\{1\}\}\}\} \subseteq N_3$, which holds iff $I_2 = \{\{\{1\}\}\} \in N_3$, using $t[N_3] = {\uparrow}\coprod\{F_2^\sharp\}$ to represent N_3. From (3.57), we get that*

$$I_2 = \{\{\{1\}\}\} \in {\uparrow}\coprod\{F_2^\sharp\} \iff I_1 = \{\{1\}\} \in F_2^\sharp$$

because $F_2^\sharp$ is the denotation of the only sub-term $t[F_2^\sharp]$ of $t[N_3]$. Equation (3.56) establishes that

$$\begin{aligned} & I_1 = \{\{1\}\} \in F_2^\sharp \iff \\ & \quad \{1\} \in {\uparrow}\coprod\big\{\{3,4\},\{2,3,4\}\big\} \ \lor \ \{1\} \in {\uparrow}\coprod\big\{\{3,4\},\{2,3,4\},\{1,2,3,4\}\big\}. \end{aligned}$$

Each of the disjuncts can then be further reduced by (3.57) into a conjunction of membership queries on the base level, which is solved using (3.58) as follows:

$$\begin{aligned} & I_1 = \{\{1\}\} \in F_2^\sharp \iff \\ & \quad (1 \in \{3,4\} \land 1 \in \{2,3,4\}) \ \lor \ (1 \in \{3,4\} \land 1 \in \{2,3,4\} \land 1 \in \{1,2,3,4\}) \end{aligned}$$

Since none of the disjuncts is satisfied, we have that $I_1 \notin F_2^\sharp$, so $I_2 \notin N_3$, implying that $I_2 \in F_3$. We conclude that $I_3 \subseteq N_3$ and hence $\models \varphi$.

3.2.6. Subsumption of Symbolic Terms

Although the use of symbolic terms instead of an explicit enumeration of sets of states itself considerably reduces the searched space, an even greater degree of reduction can be obtained using subsumption inside the symbolic representatives to reduce their size, similarly as in the antichain algorithms [WDHR06]. For any set of sets $\mathbb{S}$ containing a pair of distinct elements $R, T \in \mathbb{S}$ s.t. $R \subseteq T$, it holds that

$$\downarrow\mathbb{S} = \downarrow(\mathbb{S} \setminus \{R\}) \quad \text{and} \quad \uparrow\coprod\mathbb{S} = \uparrow\coprod(\mathbb{S} \setminus \{T\}). \tag{3.59}$$

Therefore, if $\mathbb{S}$ is used to represent the set $\downarrow\mathbb{S}$, the element R is *subsumed* by T and can be removed from $\mathbb{S}$ without changing its denotation. Likewise, if $\mathbb{S}$ is used to represent $\uparrow\coprod\mathbb{S}$, the element T is *subsumed* by R and can be removed from $\mathbb{S}$ without changing its denotation. We can thus simplify any symbolic term by pruning out its sub-terms that represent elements subsumed by elements represented by other sub-terms, without changing the denotation of the term.

Computing subsumption on terms can be done using the following two equations:

$$\downarrow\mathbb{R} \subseteq \downarrow\mathbb{S} \quad \iff \quad \forall R \in \mathbb{R} : \exists S \in \mathbb{S} : R \subseteq S \tag{3.60}$$

$$\uparrow\coprod\mathbb{R} \subseteq \uparrow\coprod\mathbb{S} \quad \iff \quad \forall S \in \mathbb{S} : \exists R \in \mathbb{R} : R \subseteq S. \tag{3.61}$$

Using (3.60) and (3.61), testing subsumption of terms of level i reduces to testing subsumption of terms of level $i - 1$. The procedure for testing subsumption of two terms descends along the structure of the term, using (3.60) and (3.61) on levels greater than 0, and on level 0, where terms are subsets of Q_0, it tests subsumption by set inclusion.

Example 3.3. *In Example 3.1, we can use the inclusions of* $\{3,4\} \subseteq \{2,3,4\} \subseteq \{1,2,3,4\}$ *and (3.59) to reduce* $t_{[N_1^\sharp]} = \uparrow\coprod\big\{\{3,4\},\{2,3,4\}\big\}$ *and the intermediate term* $t = \uparrow\coprod\big\{\{3,4\},\{2,3,4\},\{1,2,3,4\}\big\}$ *to the terms*

$$t_{[N_1^\sharp]}' = \uparrow\coprod\big\{\{2,3,4\}\big\} \text{ and}$$
$$t' = \uparrow\coprod\big\{\{1,2,3,4\}\big\} \text{ respectively.}$$

Moreover, Equation (3.61) implies that the term $t' = \uparrow\coprod\big\{\{1,2,3,4\}\big\}$ *is subsumed by* $t_{[N_1^\sharp]}' = \uparrow\coprod\big\{\{2,3,4\}\big\}$, *and so we can reduce* $t_{[F_2^\sharp]} = \downarrow\Big\{\uparrow\coprod\big\{\{2,3,4\}\big\}, \uparrow\coprod\big\{\{1,2,3,4\}\big\}\Big\}$ *to*

$$t_{[F_2^\sharp]}' = \downarrow\Big\{\uparrow\coprod\big\{\{2,3,4\}\big\}\Big\}.$$

3.3. Experimental Evaluation

We have implemented a prototype of our approach in the tool called dWiNA [FHLV14]. We built it over the frontend of the Mona tool to parse the input formula into an internal representation in the form of FAs encoded using the MTBDD-based representation from the `libvata` library [LŠV12]. dWiNA supports two modes of operation. In Mode I, we use Mona to generate the minimal deterministic automaton $\mathcal{A}_{\varphi_0}$ corresponding to the matrix of the tested formula. Since the input formula may not be in the prenex normal form (i.e., a prefix of quantifiers followed by a quantifier-free matrix), the matrix here corresponds to the subformula under the topmost quantifier, or, if there is no single top-most quantifier, to the entire formula. The automaton is then translated into the `libvata` format, and our algorithm is run on top of the `libvata`-represented automaton. In Mode II, we first transform the input formula into the prenex normal form where the occurence of negation in the matrix is limited to literals, and then construct a non-deterministic automaton $\mathcal{A}_{\varphi_0}$ for the matrix directly using `libvata`.

We evaluated dWiNA against two classes of benchmarks: formulae arising in verification of pointer programs using the method based on the logic Strand [MQ11] (as presented in Section 2.4.1), and several parametric families of manually constructed formulae, from which some were originally designed as show cases for evaluation of other tools. The main focus of our experiment was on comparing dWiNA with Mona, but we carried out some comparison with other available tools too. Namely, we compared dWiNA with an implementation of the coalgebraic decision procedure [Tra15], which we refer to as Coalg, a decision procedure based on symbolic automata [DV14], which we refer to as SFA, and the tool Toss implementing a procedure based on the Shelah's decomposition [GK10], which were briefly described in related works.

A comparison of dWiNA with Mona on Strand formulae. Table 3.1 shows the comparison of dWiNA and Mona against formulae arising in the shape analysis based on the logic Strand [MQ11]. For this comparison, we used dWiNA in Mode I. We measured the time the tools took for processing the quantifier prefix of the formulae. Overall, dWiNA was comparable and sometimes slightly slower than Mona. However, we also compared the sum of the numbers of states of all automata generated by Mona when processing the quantifier prefix with the number of symbolic terms generated by dWiNA. The state spaces generated by dWiNA are about one or two orders of magnitude smaller than those generated by Mona. This makes us believe that with enough optimization, dWiNA as a tool could become better even time-wise.

An attempt to run dWiNA on this benchmark in Mode II was unsuccessful since `libvata` was not able to construct the matrix automaton in a reasonable time. This is because the construction implemented within `libvata`, which is based on non-deterministic automata, is not optimized. In particular, it uses no automata reduction (whereas deterministic minimization is one of the key features of Mona).

Table 3.1.: Results for formulae obtained from verification tasks of STRAND [MQ11]

Benchmark	Time [s]		Space [states]	
	MONA	dWiNA	MONA	dWiNA
bubblesort-else	0.01	0.01	1285	19
bubblesort-if-else	0.02	0.23	4260	234
bubblesort-if-if	0.12	1.14	8390	28
sorted-list-insert-after-loop	0.01	0.01	167	36
sorted-list-insert-before-head	0.01	0.01	43	45
sorted-list-insert-before-loop	0.01	0.01	103	47
sorted-list-insert-error-error	0.01	0.01	103	47
sorted-list-insert-in-loop	0.01	0.01	463	59
sorted-list-reverse-after-loop	0.01	0.01	179	110
sorted-list-reverse-before-loop	0.01	0.01	179	110
sorted-list-reverse-in-loop	0.02	0.02	1311	271
sorted-list-search-after-loop	0.01	0.01	90	274
sorted-list-search-before-loop	0.01	0.01	90	274
sorted-list-search-in-loop	0.01	0.02	1311	84

A comparison of dWiNA with Mona on synthetic benchmarks. To demonstrate that our approach can, indeed, scale significantly better than the explicit automata construction, we created several parametric families of WS1S formulae. Their basic formulae express relations among subsets of $\mathbb{N}_0$, such as the existence of certain transitive relations, singleton sets, or intervals (their more elaborated definitions can be found in [FHLV14]). From these, we algorithmically generated families of formulae with a larger quantifier depth, regardless of the meaning of the generated formulae (though their semantics is still nontrivial).

In Table 3.2, we give results that we obtained from experimenting with one of the families, called `HornSub`, where the basic formula expresses existence of an ascending chain of n sets ordered w.r.t. $\subset$.[4] The parameter k stands for the number of alternations in the prefix of the formula:

$$\exists Y : \neg\exists X_1 \neg \ldots \neg\exists X_k, \ldots, X_n : \bigwedge_{1 \leq i < n} \big(X_i \subseteq Y \land X_i \subset X_{i+1}\big) \Rightarrow X_{i+1} \subseteq Y.$$

We see that dWiNA clearly outperforms MONA, showing that in many cases it is unnecessary to construct the whole state space of the corresponding automaton $\mathcal{A}_\varphi$ in order to decide the validity of the formula φ. We ran these experiments in Mode II of dWiNA (the experiment in Mode I was not successful due to a too costly conversion of a large matrix automaton from MONA to `libvata`).

[4]Results for the other families are very similar and hence skipped here. An interested reader is referred to [FHLV14] for more detailed measurements and evaluation.

Table 3.2.: Results from experiments with the `HornSub` formulae

k	Time [s]		Space [states]	
	Mona	dWiNA	Mona	dWiNA
2	0.20	0.01	25 517	44
3	0.57	0.01	60 924	50
4	1.79	0.02	145 765	58
5	4.98	0.02	349 314	70
6	∞	0.47	∞	90

All of the experiments above, targeted at a comparison of dWiNA and Mona only, were performed on an Intel Core i7-4770@3.4 GHz processor with 32 GiB RAM.

A comparison of dWiNA with other tools. Our last set of experiments aims at a comparison with other available implementations of WS1S decision procedures, namely Toss [GK10], SFA [DV14], and Coalg [Tra15]. Since the tools support a limited set of syntactic features, we could only use a subset of the available benchmark formulae. Namely, we took the parametric families of formulae `HornLeq` from [DV14] and `HornIn` from [GK10], originally proposed to evaluate the performance of SFA and Toss, respectively, and our parametric family of formulae `SetClosed`.[5] The basic formula of the `SetClosed` family expresses the non-existence of an interval set. The parameter n stands for the number of existential quantifications in the prefix of the formula:

$$\exists X_1, \ldots, X_n : \forall x : \neg \forall y, z : \bigwedge_{1 \leq i \leq n} \Big((x \in X_i \land x \leq y \land y \leq z \land z \in X_i) \Rightarrow y \in X_i \Big)$$

This experiment had to be evaluated on a different machine with a system that meets the requirements of all the tools[6], with an Intel Core i7-4770@3.4 GHz processor and 16GiB RAM, running Debian GNU/Linux. Table 3.3 gives the run times of the tools. We use ∞ in case the time exceeded 2 minutes and `oom` to denote that the tool ran out of memory. We can see that while Toss performs best on their own benchmarks, dWiNA outperforms the other tools on the rest of the formulae.

[5]Note that the `HornSub` family is not supported by Toss and Coalg, and thus we chose a comparably complex family of `SetClosed` to present the overall comparison.

[6]Note that the Toss tool required specific version of OCaml that was not available for a stable debian build.

Table 3.3.: Experiments with parametric families of formulae

Benchmark	Mona	Toss	Coalg	SFA	dWiNA
`HornLeq` [DV14]					
horn-leq06	0.01	0.02	1.10	0.01	0.01
horn-leq07	0.01	0.02	11.09	0.01	0.01
horn-leq08	0.01	0.02	101.48	0.01	0.01
horn-leq09	0.01	0.02	∞	0.01	0.01
horn-leq11	0.05	0.03	∞	0.02	0.01
horn-leq13	0.19	0.04	∞	0.02	0.01
horn-leq14	0.45	0.04	∞	0.02	0.01
horn-leq15	1.19	0.05	∞	0.03	0.02
horn-leq16	3.35	0.05	∞	0.03	0.02
horn-leq17	9.07	0.05	∞	0.03	0.02
horn-leq18	22.89	0.06	∞	0.03	0.02
horn-leq19	`oom`	0.06	∞	0.03	0.03
`HornIn` [GK10]					
horn-in04	0.01	0.01	0.02	0.27	0.01
horn-in05	0.01	0.01	0.14	0.76	0.03
horn-in06	0.01	0.02	1.07	2.65	0.13
horn-in07	0.01	0.02	8.50	8.31	0.29
horn-in08	0.01	0.02	68.05	32.44	1.16
horn-in09	0.03	0.03	∞	∞	3.42
horn-in10	0.09	0.04	∞	∞	18.40
horn-in11	0.20	0.04	∞	∞	54.74
horn-in12	0.48	0.04	∞	∞	∞
horn-in13	1.20	0.04	∞	∞	∞
horn-in14	2.95	0.05	∞	∞	∞
horn-in15	7.26	0.05	∞	∞	∞
horn-in16	`oom`	0.06	∞	∞	∞
`SetClosed`					
set-closed01	0.01	0.02	0.04	0.01	0.01
set-closed02	0.01	0.02	∞	0.13	0.01
set-closed03	0.01	0.18	∞	0.14	0.01
set-closed04	0.34	∞	∞	13.96	0.01
set-closed05	∞	∞	∞	∞	0.01
set-closed07	∞	∞	∞	∞	0.01
set-closed09	∞	∞	∞	∞	0.10
set-closed11	∞	∞	∞	∞	0.95
set-closed12	∞	∞	∞	∞	3.61
set-closed13	∞	∞	∞	∞	14.3
set-closed14	∞	∞	∞	∞	69.08
set-closed15	∞	∞	∞	∞	∞

3.4. Conclusion

We proposed a novel approach for dealing with alternating quantifications within the automata-based decision procedure for WS1S which we first proposed in [FHLV15] and then published its extended version in [FHLV19]. Our approach is based on a generalization of the idea of the so-called antichain algorithm for testing universality or language inclusion of finite automata. Our approach processes a prefix of the formula with an arbitrary number of quantifier alternations on-the-fly using an efficient symbolic representation of the state space, enhanced with subsumption pruning. Our experimental results are encouraging and show that the direction that we started in [FHLV15]—i.e. using modern techniques for non-deterministic automata in the context of deciding WS1S formulae—is rather promising.

In the next chapter we will lift the symbolic *pre*/*cpre* operators to a more general notion of terms that will allow us to work with general sub-formulae instead (that may include logical connectives and nested quantifiers). Such algorithm could then be run over arbitrary formulae, without the need of the transformation into the prenex normal form. Moreover, we will show that this also opens a way of adopting optimizations used in other tools as well as syntactical optimizations of the input formula such as anti-prenexing.

4. Lazy Automata Techniques for WS1S

In preliminaries, we have shown that the classical WS1S decision procedure builds an automaton A_φ accepting all encodings of models of the given formula φ in a form of finite words, and only then tests whether the language of A_φ is empty. As we have already discussed in Chapter 3, the bottleneck of this procedure is the size of A_φ, which can be huge due to the fact that the derivation of A_φ involves many nested automata product constructions and complementation steps, preceded by determinization. We have demonstrated how one can avoid this bottleneck when processing the topmost quantifier prefix of the given formulae in Chapter 3. However, we limited ourselves to processing of this prefix only and hence could not process arbitrary formula efficiently, and, moreover, in quite some cases the decision procedure still led to a state-space explosion.

Hence, our next goal is to avoid more of the state-space explosion involved in the *explicit* construction and to handle formulae without a need to transform them into the prenex normal form. We represent automata *symbolically* and, while constructing A_φ, we test the emptiness of A_φ *on the fly* which allows us to omit the state space irrelevant to the emptiness test. We build on two main principles: *lazy evaluation* and *subsumption-based pruning.* These principles have, to some degree, already appeared in the so-called antichain-based testing of language universality and inclusion of finite automata [WDHR06]. However, the richer structure of the WS1S decision problem allows us to elaborate on these principles in novel ways and utilize their power even more.

Overview of our algorithm. We propose an algorithm which originates in the classical WS1S decision procedure, in which models of formulae are encoded by finite words over a multi-track binary alphabet where each track corresponds to a variable of φ. In t_φ, the atomic formulae of φ are replaced by predefined automata accepting languages of their models. Boolean operators ($\wedge$, $\vee$, and $\neg$) are turned into the corresponding set operators ($\cup$, $\cap$, and complement) over the languages of models. An existential quantification $\exists X$ becomes a sequence of two operations. First, a projection π_X removes information about valuations of the quantified variable X from symbols of the alphabet, i.e. the so-called *projection* operation. After the projection, the resulting language L may, however, encode some but not necessarily *all* encodings of the models (a problem, we have outlined in the Chapter 2). In particular, encodings with some specific numbers of trailing $\bar{0}$'s, used as a padding, may be missing. To obtain a language containing *all* encodings of the models, L must be extended to include encodings with any number of trailing $\bar{0}$'s.

In the method presented in Chapter 3, this process was handled by defined fixpoint computations. Here instead, we take the (right) $\bar{0}^*$-quotient of L, written $L - \bar{0}^*$, which is the set of all prefixes of words of L with the remaining suffix in $\bar{0}^*$. We give an example WS1S formula φ in equation (4.1) and its language term $t[\varphi]$ in equation (4.2). The dotted operators represent operators over language terms. See Fig. 4.2 for the automata $\mathcal{A}_{\mathrm{Sing}(X)}$ and $\mathcal{A}_{Y=X+1}$.

$$\varphi \equiv \exists X\colon \mathrm{Sing}(X) \wedge (\exists Y\colon Y = X + 1) \tag{4.1}$$

$$t[\varphi] \equiv \pi_X(\{\mathcal{A}_{\mathrm{Sing}(X)} \mathbin{\dot{\cap}} (\pi_Y(\mathcal{A}_{Y=X+1}) \mathbin{\dot{-}} \bar{0}^*)\}) \mathbin{\dot{-}} \bar{0}^* \tag{4.2}$$

The novelty of our work is that we test the emptiness of L_φ directly over $t[\varphi]$. The term is used as a symbolic representation of the automata that would be explicitly constructed in the classical procedure: inductively to the terms structure, starting from the leaves and combining the automata of sub-terms by standard automata constructions that implement the term operators. Instead of first building the automaton and only then testing emptiness, we test it on the fly during the construction. This offers opportunities to prune out portions of the state space that turn out not to be relevant for the test.

A sub-term $t[\psi]$ of $t[\varphi]$, corresponding to a sub-formula ψ, represents final states of the automaton $\mathcal{A}_\psi$ accepting the language encoding models of ψ. Predecessors of the final states represented by $t[\psi]$ correspond to quotients of $t[\psi]$. All states of $\mathcal{A}_\psi$ could hence be constructed by applying quotient operation on $t[\psi]$ until fixpoint. By working with terms, our procedure can often avoid building large parts of the automata when they are not necessary for answering the emptiness query. For instance, when testing the emptiness of the language of a term $t_1 \cup t_2$, we adopt the *lazy approach* (in this particular case, the so-called *short-circuit evaluation*) and first test the emptiness of the language of t_1; if it is non-empty, we do not need to process t_2. Testing language emptiness of terms arising from quantified sub-formulae is more complicated since they translate to $-\bar{0}^*$ quotients. We evaluate the test on $t - \bar{0}^*$ by iterating the $-\bar{0}$ quotient from t. We either conclude with the positive result as soon as one of the iteration computes a term with a non-empty language, or with the negative one if the fixpoint of the quotient construction is reached. The fixpoint condition is that the so-far computed quotients *subsume* the newly constructed ones, where subsumption is a relation under-approximating inclusion of languages represented by terms. We also use subsumption to prune the set of computed terms so that only an *antichain* of the terms maximal wrt subsumption is kept.

Besides lazy evaluation and subsumption, our approach can benefit from multiple further optimizations. For example, it can be *combined* with the *explicit WS1S decision procedure*, which can be used to transform arbitrary sub-terms of t_φ to automata. These automata can then be rather small due to minimization, which cannot be applied in the on-the-fly approach (the automata can, however, also explode due to determinisation and product construction, hence this technique comes with a trade-off). We also propose a novel way of *utilising BDD-based encoding* of automata transition functions in the Mona style for computing quotients of terms. Finally, our method can exploit various methods of *logic-based pre-processing*, such as *anti-prenexing*, which, in our experience, can often significantly reduce the search space of fixpoint computations.

Experiments. We have implemented our decision procedure in a tool called GASTON and compared its performance with other publicly available WS1S solvers on benchmarks from various sources. In our experiments, GASTON managed to win over all other solvers on various parametric families of formulae that were designed—mostly by authors of other tools—to stress-test WS1S solvers. Moreover, GASTON was able to significantly outperform MONA and other solvers on a number of formulae obtained from various formal verification tasks, in particular formulae describing properties of singly-linked lists [MQ11] and arrays [ZHW$^+$14] (see Section 2.4). This shows that our approach is applicable in practice and has a great potential to handle more complex formulae than those so far obtained in WS1S applications. We believe the efficiency of our approach can be pushed much further, making WS1S scale enough for new classes of applications.

Contributions. We summarize our contributions to WS1S achieved by our second proposed approach linked with the Gaston tool:

1. Instead of the explicit automata construction described in Section 2.3, we develop an on-the-fly decision procedure based on the so-called language terms that can efficiently process arbitrary formulae. Contrary to the classical procedure, our method can avoid costly determinisation in many cases.

2. We propose a combination of our procedure with the classical decision procedure for WS1S as implemented, e.g. by the MONA tool. This allows one to exploit the key optimizations of both approaches, i.e. minimization and lazy evaluation.

3. Besides the novel decision procedure, we develop a series of optimizations that are not limited to our approach only. Other tools and methods can exploit these optimizations, such as, e.g. anti-prenexing, to achieve better efficiency.

4. In our experimental evaluation, we demonstrate we improve the state of the art of WS1S decision procedures, especially on formulae describing program invariants of advanced data structures. In particular, we report on a series of benchmarks used for verification of programs, where we outperformed the MONA tool.

5. We perform an extensive evaluation of all the publicly available tools (listed in Section 2.3.1) on a series of benchmarks that are used to stress-test WS1S decision procedures. We present a fair speed comparison of all of the available tools.

Outline. The chapter is structured into three sections. In Section 4.1, we introduce the notion of language terms and propose a basic version of the decision procedure for WS1S based on such terms. Next, in Section 4.2, we show how to optimize this basic procedure. In particular, we show how to exploit the lazy evaluation principles and subsumption to prune out large portions of the generated state space. In addition, we also describe other optimizations and heuristics, such as the anti-prenexing technique or a combination of our procedure with the classical decision procedure as described in Chapter 2. We briefly describe the implementation of the Gaston tool and the experimental evaluation comparing our procedure with other tools in the Section 4.3. Finally, we conclude this chapter with Section 4.4 and propose possible future directions of our research.

4.1. Satisfiability via Language Term Evaluation

First, we introduce the basic version of our symbolic algorithm for deciding satisfiability of a WS1S formula φ with a set of variables $\mathbb{V}$. To simplify the presentation, we consider the particular case of *ground* formulae (i.e. formulae without free variables), for which satisfiability corresponds to validity. Satisfiability of a formula with free variables can be reduced to this case by prefixing it with existential quantification over the free variables. If φ is ground, the language $\mathcal{L}^{\mathbb{V}}(\varphi)$ is either $\Sigma_{\mathbb{V}}^*$ in the case φ is valid, or empty if φ is invalid. Then, to decide the validity of φ, it suffices to test if $\epsilon \in \mathcal{L}^{\mathbb{V}}(\varphi)$.

Our algorithm evaluates the so-called *language term* $t_{[\varphi]}$, a symbolic representation of the language $\mathcal{L}^{\mathbb{V}}(\varphi)$, whose structure reflects the construction of $\mathcal{A}_\varphi$. It is a (finite) term generated by the following grammar:

$$t ::= \mathcal{A} \mid t \mathbin{\dot{\cup}} t \mid t \mathbin{\dot{\cap}} t \mid \bar{t} \mid \pi_{\mathcal{X}}(t) \mid t \mathbin{\dot{-}} \alpha \mid t \mathbin{\dot{-}} \alpha^* \mid T$$

where $\mathcal{A}$ is a finite automaton over the alphabet $\Sigma_{\mathbb{V}}$, α is a symbol $\tau \in \Sigma_{\mathbb{V}}$ or a set $S \subseteq \Sigma_{\mathbb{V}}$ of symbols, and T is a finite set of terms. We use dotted variants of the operators to distinguish the syntax of language terms manipulated by our algorithm from the cases when we wish to denote the semantical meaning of the operators. A term of the form $t \mathbin{\dot{-}} \alpha^*$ is called a *star quotient*, or shortly a *star*, and a term $t \mathbin{\dot{-}} \tau$ is a *symbol quotient*. Both are also called *quotients*. The *language* $\mathcal{L}(t)$ *of a term* t is obtained by taking the languages of the automata in its leaves and combining them using the term operators. Terms with the same languages are *language-equivalent*. The special terms T, having the form of a set, represent intermediate states of fixpoint computations used to eliminate star quotients. The language of a set T equals to the *union* of the languages of its elements. The reason for having two ways of expressing a union of terms is a different treatment of $\dot{\cup}$ and T, which will be discussed later. We use the standard notion of isomorphism of two terms, extended with having two set terms isomorphic iff they contain isomorphic elements.

A formula φ is initially transformed into the term $t_{[\varphi]}$ by replacing every atomic sub-formula ψ in φ by the automaton $\mathcal{A}_\psi$ accepting $\mathcal{L}^{\mathbb{V}}(\psi)$, and by replacing the logical connectives with dotted term operators according to equations (2.3)–(2.6) of Section 2.3. The core of our algorithm is evaluation of the ϵ-membership query $\epsilon \in t_{[\varphi]}$, which will also trigger further rewriting of the term.

$$\epsilon \in T \text{ iff } \epsilon \in t \text{ for some } t \in T \tag{4.3}$$

$$\epsilon \in t \mathbin{\dot{\cup}} t' \text{ iff } \epsilon \in t \text{ or } \epsilon \in t' \tag{4.4}$$

$$\epsilon \in t \mathbin{\dot{\cap}} t' \text{ iff } \epsilon \in t \text{ and } \epsilon \in t' \tag{4.5}$$

$$\epsilon \in \bar{t} \text{ iff not } \epsilon \in t \tag{4.6}$$

$$\epsilon \in \pi_{\mathcal{X}}(t) \text{ iff } \epsilon \in t \tag{4.7}$$

$$\epsilon \in \mathcal{A} \text{ iff } I(\mathcal{A}) \cap F(\mathcal{A}) \neq \emptyset \tag{4.8}$$

The ϵ-membership query on a quotient-free term is evaluated using equivalences (4.3) to (4.8). These equivalences reduce tests on terms to Boolean combinations of tests on their sub-terms and allow pushing the test towards the automata at the term's leaves. Equivalence (4.8) then reduces it to testing intersection of the initial states $I(\mathcal{A})$ and the final states $F(\mathcal{A})$ of an automaton.

Evaluation of quotient terms. Equivalences (4.3) to (4.7) do not apply to quotients, which arise from quantified sub-formulae (cf. equation (2.6) in Section 2.3). A quotient is therefore (in the basic version) first rewritten into a language-equivalent quotient-free form. This rewriting corresponds to saturating the set of final states of an automaton in the explicit decision procedure with all states in their *pre**-image over $\bar{0}$. In our procedure, we use rules (4.9) and (4.10).

$$\pi_{\mathcal{X}}(T) \mathbin{\dot{-}} \bar{0}^* \to \pi_{\mathcal{X}}(T \mathbin{\dot{-}} \pi_{\mathcal{X}}(\bar{0})^*) \tag{4.9}$$

Rule (4.9) transforms the term into a form in which a star quotient is applied on a plain set of terms rather than on a projection. A star quotient of a set is then eliminated using a fixpoint computation that saturates the set with all quotients of its elements wrt the set of symbols $S = \pi_{\mathcal{X}}(\bar{0})$. A single iteration is implemented using rule (4.10).

$$T \mathbin{\dot{-}} S^* \to \begin{cases} T & \text{if } T \ominus S \sqsubseteq T \\ (T \cup (T \ominus S)) \mathbin{\dot{-}} S^* & \text{otherwise} \end{cases} \tag{4.10}$$

In rule (4.10), $T \ominus S$ is the set $\{t \mathbin{\dot{-}} \tau \mid t \in T \wedge \tau \in S\}$ of quotients of terms in T wrt symbols of S. (Note that rule (4.10) uses the identity $S^* = \{\epsilon\} \cup S^* S$.) Termination of the fixpoint computation is decided based on the subsumption relation $\sqsubseteq$: some relation that under-approximates language inclusion of terms. When the condition holds, then the language of T is stable wrt quotient operation by S, i.e. $\mathcal{L}(T) = \mathcal{L}(T \mathbin{\dot{-}} S^*)$. In the basic algorithm, we use term isomorphism for $\sqsubseteq$; later, we provide a more precise subsumption with a good trade-off between precision and cost. Note that an iteration of rule (4.10) can be implemented efficiently by the standard worklist algorithm, which extends T only with quotients $T' \ominus S$ of terms T' that were added to T in the previous iteration.

$$(t \mathbin{\dot{\cup}} t') \mathbin{\dot{-}} \tau \to (t \mathbin{\dot{-}} \tau) \mathbin{\dot{\cup}} (t' \mathbin{\dot{-}} \tau) \tag{4.11}$$

$$(t \mathbin{\dot{\cap}} t') \mathbin{\dot{-}} \tau \to (t \mathbin{\dot{-}} \tau) \mathbin{\dot{\cap}} (t' \mathbin{\dot{-}} \tau) \tag{4.12}$$

$$\bar{t} \mathbin{\dot{-}} \tau \to \overline{t \mathbin{\dot{-}} \tau} \tag{4.13}$$

$$\pi_{\mathcal{X}}(t) \mathbin{\dot{-}} \tau \to \pi_{\mathcal{X}}(t \mathbin{\dot{-}} \pi_{\mathcal{X}}(\tau)) \tag{4.14}$$

$$\mathcal{A} \mathbin{\dot{-}} \tau \to \mathit{pre}_{[\tau]}(\mathcal{A}) \tag{4.15}$$

The set $T \ominus S$ introduces quotient terms of the form $t \mathbin{\dot{-}} \tau$, for $\tau \in \Sigma_{\mathbb{V}}$, which also need to be eliminated to facilitate the ϵ-membership test. This can be done using rewriting rules (4.11) to (4.15), where $\mathit{pre}_{[\tau]}(\mathcal{A})$ is $\mathcal{A}$ with its set of final states F replaced by $\mathit{pre}_{[\tau]}(F)$, i.e. its predecessors.

If t is quotient-free, then rules (4.11)–(4.14) applied to $t \mathbin{\dot{-}} \tau$ push the symbol down the structure of t towards the leaves, where it is eliminated by rule (4.15). Otherwise, if t is not quotient-free, it can be re-written using (4.9)–(4.15). In particular, if t is a star quotient of a quotient-free term, then the quotient-free form of t can be obtained by iterating rule (4.10), combined with rules (4.11)–(4.15) to transform the new terms in T into a quotient-free form. Finally, terms with multiple quotients can be rewritten to the quotient-free form inductively to the structure. Every inductive step rewrites some star quotient of a quotient-free sub-term into the quotient-free form. Note this is bound to terminate since the terms generated by performing quotient operation on a star have the same structure as the original term, differing only in the leaves. As the number of the states is finite, so is the number of the terms. Hence, it will eventually terminate.

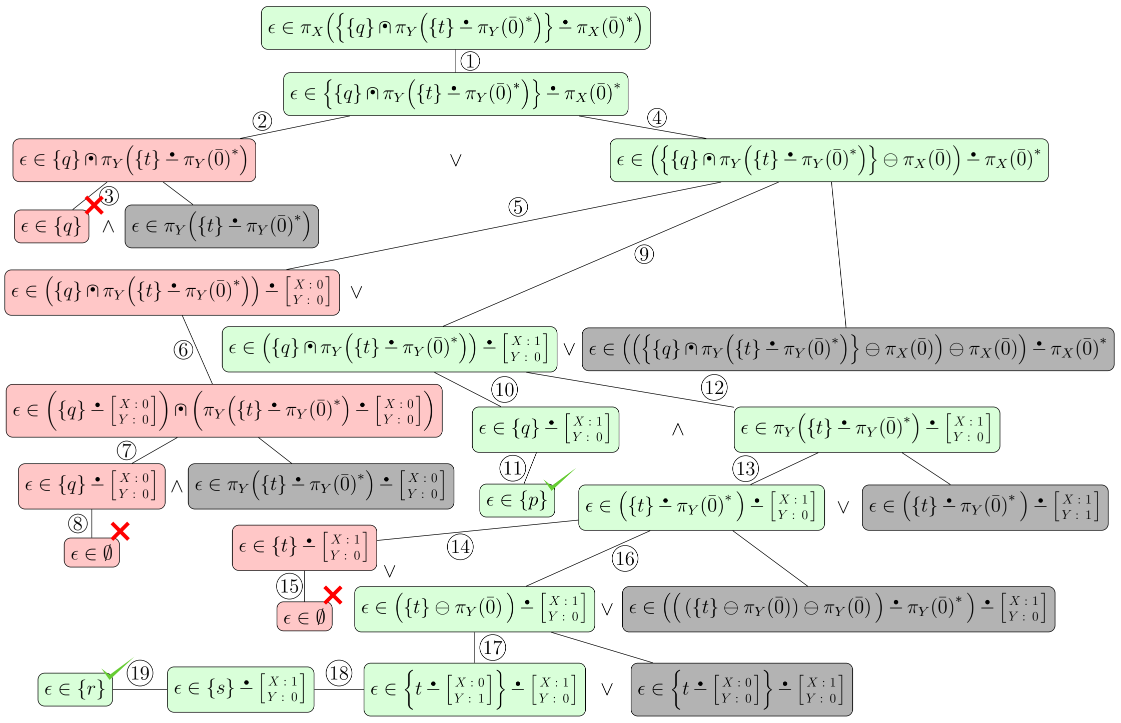
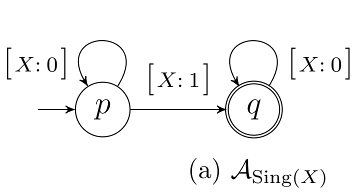
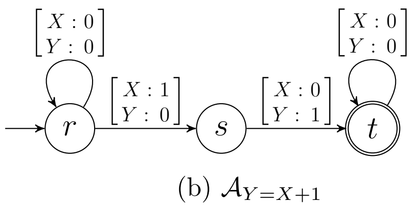


Figure 4.1.: Example of deciding the validity of the formula $\varphi \equiv \exists X : \mathrm{Sing}(X) \wedge (\exists Y : Y = X + 1)$ using the basic procedure.



(a) $\mathcal{A}_{\mathrm{Sing}(X)}$



(b) $\mathcal{A}_{Y=X+1}$

Figure 4.2.: Example automata

Example 1. We will show how our procedure works using an example of testing the satisfiability of the formula $\varphi \equiv \exists X : \mathrm{Sing}(X) \wedge (\exists Y : Y = X + 1)$. We start by rewriting φ into a *term* $t_{[\varphi]}$ representing its language $\mathcal{L}^{\mathbb{V}}(\varphi)$:

$$t_{[\varphi]} \equiv \pi_X(\{\{q\} \Cap \pi_Y(\{t\} \mathbin{\dot{-}} \pi_Y(\bar{0})^*)\} \mathbin{\dot{-}} \pi_X(\bar{0})^*)$$

(note that we have already used rule (4.9) twice). In the example, a set R of states will denote an automaton obtained from $\mathcal{A}_{\mathrm{Sing}(X)}$ or $\mathcal{A}_{Y=X+1}$ (cf. Fig. 4.2) by setting the final states to R. Red nodes in the computation tree denote ϵ-membership tests that failed and green nodes those that succeeded. Grey nodes denote tests that were not evaluated.

As noted previously, it holds that $\models \varphi$ iff $\epsilon \in t_{[\varphi]}$. The sequence of computation steps for determining the ϵ-membership test is shown using the computation tree in Fig. 4.1. The nodes contain ϵ-membership tests on terms and the test of each node is equivalent to a conjunction or disjunction of tests of its children. On the other hand, leafs of the form $\epsilon \in R$ are evaluated as testing intersection of R with the initial states of the corresponding automaton. We remark that, in the example, we also use the *lazy evaluation*—a technique (described in Section 4.2.2), which allows us to evaluate ϵ-membership tests on partially computed fixpoints.

The computation starts at the root of the tree and proceeds along the edges in the order given by their circled labels. Edges ② and ④ were obtained by a partial unfolding of a fixpoint computation by rule (4.10) and immediately applying ϵ-membership test on the obtained terms. After step ③, we conclude that $\epsilon \notin \{q\}$ since $\{p\} \cap \{q\} = \emptyset$, which further refutes the whole conjunction below ②, so the overall result depends on the sub-tree starting by ④. Steps ⑤ and ⑨ represent another application of rule (4.10), which transforms $\pi_X(\bar{0})$ to the symbols $\left[\begin{smallmatrix}X: & 0\\ Y: & 0\end{smallmatrix}\right]$ and $\left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]$, respectively. Branch ⑤ pushes the $\dot{-}\left[\begin{smallmatrix}X: & 0\\ Y: & 0\end{smallmatrix}\right]$ quotient to the leaf term using rules (4.12) and (4.5) and eventually fails because the predecessors of $\{q\}$ over the symbol $\left[\begin{smallmatrix}X: & 0\\ Y: & 0\end{smallmatrix}\right]$ in $\mathcal{A}_{\mathrm{Sing}(X)}$ is the empty set. On the other hand, the evaluation of branch ⑨ continues using rule (4.12), succeeding in branch ⑩. Branch ⑫ is further evaluated by projecting the quotient $\dot{-}\left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]$ wrt Y (rule 4.14) and unfolding the inner star quotient zero times (when the test ⑭ failed) and once (when the test ⑯ succeeded). The unfolding of one symbol eventually succeeds in step ⑲, which leads to the conclusion that formula φ is, indeed, valid. Note that thanks to the lazy evaluation, none of the fixpoint computations had to be fully unfolded.

4.2. Towards An Efficient Algorithm

In this section, we show how to build an efficient algorithm based on the symbolic term rewriting approach presented in Section 4.1. This symbolic approach offers many optimization opportunities that are to a large degree orthogonal to those of the explicit approach. The main difference is in the available techniques for reducing the explored automata state space. While the explicit construction in MONA profits mainly from calling *automata minimization* after every step of the inductive construction, the symbolic algorithm can instead exploit generalized *subsumption* and *lazy evaluation*. Unfortunately, neither the explicit nor the symbolic approach seems to be compatible with both minimization and lazy techniques (at least in their pure variant, disregarding the possibility or a combination of the two approaches that we will presented later in this section).

Efficient data structures have a major impact on performance of the decision procedure. The efficiency of the explicit procedure implemented in MONA is to a large degree due to the BDD-based representation of automata transition relations. BDDs compactly represent transition functions over large alphabets and provide efficient implementation of operations needed in the explicit algorithm. Our symbolic algorithm can, on the other hand, benefit from a representation of terms as DAGs where all occurrences of the same sub-term are represented by a unique DAG node. Moreover, we assume the nodes to be associated with languages rather than with concrete terms (allowing the term associated with a node to change during its further processing, without a need to transform the DAG structure as long as the language of the term does not change).

We also show that despite our algorithm uses a completely different data structure than the explicit one, it can still exploit a BDD-based representation of transitions of the automata in the leaves of terms. Moreover, our symbolic algorithm can also be *combined* with the explicit algorithm. Particularly, it turns out that, sometimes, it pays off to translate to automata sub-formulae larger than atomic ones. Our procedure can then be viewed as an extension of MONA that takes over once MONA stops managing. Lastly, optimizations on the level of formulae often have a huge impact on the performance of our algorithm. The technique that we found most helpful is the so-called *anti-prenexing*. We elaborate on all these optimizations in the rest of this section.

4.2.1. Subsumption

Our first technique for reducing the explored state space is based on the notion of *subsumption* between symbolic terms, which is similar to the subsumption used in antichain-based universality and inclusion checking over finite automata [WDHR06]. We define subsumption as the relation $\sqsubseteq_s$ on terms that is given by equivalences (4.16)–(4.21). Notice that, in rule (4.16), all terms of T are tested against all terms of T', while in rule (4.17), the left-hand side term t_1 is not tested against the right-hand side term t'_2 (and similarly for t_2 and t'_1).

$$T \sqsubseteq_s T' \text{ iff } \forall t \in T\ \exists t' \in T' : t \sqsubseteq_s t' \tag{4.16}$$

$$t_1 \mathbin{\dot\cup} t_2 \sqsubseteq_s t'_1 \mathbin{\dot\cup} t'_2 \text{ iff } t_1 \sqsubseteq_s t'_1 \text{ and } t_2 \sqsubseteq_s t'_2 \tag{4.17}$$

$$t_1 \mathbin{\dot\cap} t_2 \sqsubseteq_s t'_1 \mathbin{\dot\cap} t'_2 \text{ iff } t_1 \sqsubseteq_s t'_1 \text{ and } t_2 \sqsubseteq_s t'_2 \tag{4.18}$$

$$\overline{t} \sqsubseteq_s \overline{t'} \text{ iff } t \sqsupseteq_s t' \tag{4.19}$$

$$\pi_{\mathcal{X}}(t) \sqsubseteq_s \pi_{\mathcal{X}}(t') \text{ iff } t \sqsubseteq_s t' \tag{4.20}$$

$$\mathcal{A} \sqsubseteq_s \mathcal{A}' \text{ iff } F(\mathcal{A}) \subseteq F(\mathcal{A}') \tag{4.21}$$

The reason why the dotted operator $\dot\cup$ is order-sensitive is that the terms on different sides of the $\dot\cup$ are assumed to be built from automata with disjoint sets of states (originating from different sub-formulae of the original formula), and hence the subsumption test on them can never conclude positively. The subsumption under-approximates language inclusion and can therefore be used for $\sqsubseteq$ in rule (4.10). It is far more precise than isomorphism and its use leads to an earlier termination of fixpoint computations.

Moreover, $\sqsubseteq_s$ can be used to prune star quotient terms $T \mathbin{\dot-} S^*$ while preserving their language. Since the semantics of the set T is the union of the languages of its elements, then elements subsumed by others can be removed while preserving the language. T can thus be kept in the form of an *antichain* of $\sqsubseteq_s$-incomparable terms. This pruning corresponds to using the rewriting rule (4.22):

$$T \to T \setminus \{t\} \qquad\qquad \text{if there is } t' \in T \setminus \{t\} \text{ with } t \sqsubseteq_s t'. \tag{4.22}$$

4.2.2. Lazy Evaluation

Our approach works in the top-down nature. Hence, we can postpone evaluation of some of the computation branches in case the so-far evaluated part is sufficient for determining the result of the evaluated ϵ-membership or subsumption test. We call this optimization *lazy evaluation.* A basic variant of lazy evaluation *short-circuits* elimination of quotients from branches of $\dot\cup$ and $\dot\cap$ terms. When we test whether $\epsilon \in t \mathbin{\dot\cup} t'$ (rule (4.4)), we first evaluate, e.g. the test $\epsilon \in t$, and when it holds, we can completely avoid exploring t' and evaluating quotients there. We can proceed analogously when we test $\epsilon \in t \mathbin{\dot\cap} t'$, if one of the two terms is shown not to contain ϵ. Rules (4.17) and (4.18) offer similar opportunities for short-circuiting evaluation of subsumption of $\dot\cup$ and $\dot\cap$ terms.

Let us note that subsumption is in a different position than ϵ-membership since correctness of our algorithm depends on the precision of the ϵ-membership test, but subsumption may be evaluated in any way that under-approximates inclusion of languages of terms (and over-approximates isomorphism in order to guarantee termination). Hence, ϵ-membership test must enforce eliminating quotients until it can conclude the result, while there is a choice in the case of the subsumption. If subsumption is tested on quotients, it can either eliminate them, or it can return the (safe) negative answer. However, this choice comes with a trade-off. Subsumption eliminating quotients is more expensive but also more precise. The higher precision allows better pruning of the state space and earlier termination of fixpoint computation, which, according to our empirical experience, pays off.

Lazy evaluation can also reduce the number of iterations of a star operator. We can compute iterations *on demand*, only when we require them in the underlying tests. The idea is to try to conclude a test $\epsilon \in T \dot{-} S^*$ based on the intermediate state T of the fixpoint computation. We can perform such conclusion since $\mathcal{L}(T)$ always under-approximates $\mathcal{L}(T \dot{-} S^*)$, hence if $\epsilon \in \mathcal{L}(T)$, then $\epsilon \in \mathcal{L}(T \dot{-} S^*)$. Continuing the fixpoint computation is then unnecessary.

The above mechanism alone is, however, rather insufficient in the case of nested stars. Assume that an inner star fixpoint computation was terminated in a state $T \dot{-} S^*$ when ϵ was found in T for the first time. Every unfolding of an outer star then propagates $\dot{-}\,\tau$ quotients towards $T \dot{-} S^*$. We have, however, no way of eliminating it from $(T \dot{-} S^*) \dot{-} \tau$ other than finishing the unfolding of $T \dot{-} S^*$ first (which eliminates the inner star). The need to fully unfold $T \dot{-} S^*$ would render the earlier lazy evaluation of the ϵ-membership test worthless. To remove this deficiency, we need a way of eliminating the $\dot{-}\,\tau$ quotient from the intermediate state of $T \dot{-} S^*$.

$$T \dot{-} S^* \rightarrow T \dot{-} S^* \succcurlyeq T \tag{4.23}$$

$$\epsilon \in t \succcurlyeq t' \;\text{ if }\; \epsilon \in t' \tag{4.24}$$

$$t \succcurlyeq T \not\sqsupseteq_s t' \text{ if }\; T \not\sqsupseteq_s t' \tag{4.25}$$

$$t \succcurlyeq T \sqsupseteq_s t' \text{ if }\; T \sqsupseteq_s t' \tag{4.26}$$

The elimination is achieved by letting the star quotient $T \dot{-} S^*$ explicitly "publish" its intermediate state T using rule (4.23). The symbol $\succcurlyeq$ is read as "*is under-approximated by.*" Rules (4.24)–(4.26) allow to conclude ϵ-membership and subsumption by testing the under-approximation on its right-hand side (notice the distinction between "if" and the "iff" used in the rules earlier).

$$(t \succcurlyeq T) \dot{-} S \rightarrow ((t \succcurlyeq T) \dot{-} S) \succcurlyeq T \ominus S \tag{4.27}$$

Symbol quotients that come from the unfolding of an outer star can too be evaluated on the approximation using rule (4.27), which then applies the symbol-set quotient on the approximation T of the inner term t, and publishes the result on the right-hand side of $\succcurlyeq$. The left-hand side still remembers the original term $t \dot{-} S$.

$$T \succcurlyeq T' \rightarrow T \tag{4.28}$$

Terms arising from rules (4.23) and (4.27) allow an efficient update in the case an inner term t spawns a new, more precise approximation. In the process, rule (4.28) is used to remove old outdated approximations, if better ones are already available.

Example 4.1. *We will now illustrate how we can apply these rules and how we can implement them efficiently on an evaluation from Example 1. Note that, in Example 1, the partial unfoldings of the fixpoints that are tested for ϵ-membership, e.g. the branch under step ②, are under-approximations of a star quotient term. For instance, branch ⑭ corresponds to testing ϵ-membership in the right-most approximation of the following term $\Big(((\{t\} \dot{-} \pi_Y(\bar{0})^*) \not\succcurlyeq \{t\}) \dot{-} \left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]\Big) \not\succcurlyeq \{t\} \dot{-} \left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]$ by rule (4.24) (the branch determines that $\epsilon \notin \{t\} \dot{-} \left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]$). The result of ⑭ cannot conclude the top-level ϵ-membership test because $\{t\} \dot{-} \left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]$ is just an under-approximation of term $(\{t\} \dot{-} \pi_Y(\bar{0})^*) \dot{-} \left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]$. Therefore, we need to compute a better approximation of the term and try to conclude the test on it. We compute it by first applying rule (4.28) twice to discard obsolete approximations ($\{t\}$ and $\{t\} \dot{-} \left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]$), followed by applying rule (4.10) to replace $(\{t\} \dot{-} \pi_Y(\bar{0})^*) \dot{-} \left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]$ with $\big((\{t\} \cup (\{t\} \ominus \pi_Y(\bar{0}))) \dot{-} \pi_Y(\bar{0})^*\big) \dot{-} \left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]$. Let $\beta = \{t\} \cup (\{t\} \ominus \pi_Y(\bar{0}))$. Then, we can use the rules (4.23) and (4.27), to rewrite the term $(\beta \dot{-} \pi_Y(\bar{0})^*) \dot{-} \left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]$ into $\Big((\beta \dot{-} \pi_Y(\bar{0})^* \not\succcurlyeq \beta) \dot{-} \left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]\Big) \not\succcurlyeq \beta \ominus \left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]$, where $\beta \ominus \left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]$ is the approximation used in step ⑯, and re-evaluate the ϵ-membership test on it.*

Implemented naïvely, the computation of subsequent approximations of fixpoint terms would involve a lot of redundancy, e.g. in $\beta \dot{-} \left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]$ we would need to recompute the term $\{t\} \dot{-} \left[\begin{smallmatrix}X: & 1\\ Y: & 0\end{smallmatrix}\right]$, which was already computed in step ⑮. The mechanism can, however, be implemented efficiently so that it completely avoids the redundant computations. Firstly, we can maintain a cache of already evaluated terms and never evaluate the same term repeatedly. Secondly, suppose that a term $t \dot{-} S^*$ has been unfolded several times into intermediate states $(T_1 = \{t\}) \dot{-} S^*, T_2 \dot{-} S^*, \ldots, T_n \dot{-} S^*$. One more unfolding using (4.10) would rewrite $T_n \dot{-} S^*$ into $T_{n+1} = (T_n \cup (T_n \ominus S)) \dot{-} S^*$. When computing the set $T_n \ominus S$, however, we do not need to consider the whole set T_n, but only those elements that are in T_n and are not in T_{n-1} (since $T_n = T_{n-1} \cup (T_{n-1} \ominus S)$, all elements of $T_{n-1} \ominus S$ are already in T_n).

Thirdly, in the direct acyclic graph (DAG) representation of terms, which we will describe in Section 4.2.3, a term $(T \cup (T \ominus S)) \dot{-} S^* \not\succcurlyeq T \cup (T \ominus S)$ is represented by the set of terms obtained by evaluating $T \ominus S$, a pointer to the term $T \dot{-} S^*$ (or rather to its associated DAG node), and the set of symbols S. The cost of keeping the history of quotienting together with the under-approximation (on the right-hand side of $\not\succcurlyeq$) is hence only a pointer and a set of symbols.

4.2.3. Efficient Data Structures

Further, we will introduce two important techniques that we use in our implementation that concern (1) representation of symbolic terms and (2) utilisation of BDD-based symbolic representation of transition functions of automata in the leaves of the terms.

Representation of language terms. We keep each term in the form of a DAG such that all isomorphic instances of the same term are represented as a unique DAG node. Moreover, when we rewrite a term into a language-equivalent one, we still associate it with the same DAG node. Newly computed sub-terms are always first compared against the existing ones, and, if possible, associated with an existing DAG node of an existing isomorphic term. The fact that isomorphic terms are always represented by the same DAG node makes it possible to test isomorphism of a new and previously processed term efficiently—it is enough to test that their direct sub-terms are represented by identical DAG nodes (let us note that we do not look for language equivalent terms because of the high cost of such a check).

We also cache results of membership and subsumption queries. The key to the cache is the identity of DAG nodes, not the represented sub-terms, which has the advantage that results of tests over a term are available in the cache even after it is rewritten according to $\to$ (as it is still represented by the same DAG node). The cache together with the DAG representation is especially efficient when evaluating a new subsumption or ϵ-membership test since although the result is not in the cache, the results for its sub-terms often are. As further optimization, we expanded our cache to maintain subsumptions closed under the transitivity. In our experience, even though we are forced to compute the transitivity in the cache, which can be costly, in process we profit from the faster subsumption tests.

BDD-based symbolic automata. One of the central challenges for our algorithm is coping efficiently with large sets of symbols. Notice that rules (4.10) and (4.14) compute a quotient for each of the symbols in the set $\pi_{\mathcal{X}}(\tau)$ separately. Since the number of the symbols is $2^{|\mathcal{X}|}$, this can easily make the computation infeasible. And, indeed, in our benchmarks, the set $\mathcal{X}$ tends to be quite high leading not only into a big number of symbols, but also into large underlying automata.

Mona handles large sets of symbols by using a BDD-based symbolic representation of transition relations of automata as follows: The alphabet symbols of the automata are assignments of Boolean values to the free variables $X_1, \ldots, X_n$ of a formula. The transitions leading from a state q can be expressed as a function $f_q : 2^{\{X_1,\ldots,X_n\}} \to Q$ from all assignments to states such that $(q, \tau, q') \in \delta_q$ iff $f_q(\tau) = q'$. The function f_q is encoded as a multi-terminal BDD (MTBDD) with variables $X_1, \ldots, X_n$ and terminals from the set Q (essentially, it is a DAG where a path from the root to a leaf encodes a set of transitions). The BDD `apply` operation is then used to efficiently implement the computation of successors of a state via a large set of symbols, and to facilitate essential constructions such as product, determinization, and minimization. We use Mona to create automata in leaves of our language terms exploiting its efficient minimization and automata representation. But, to fully utilize their BDD-based symbolic representation, we first had to overcome the following two problems.

First, our algorithm has to compute predecessors of states, while the BDDs of MONA are designed to compute successors. So to use `apply` to compute states backwards, the BDDs would have to be turned into a representation of the inverted transition function. But, this is a costly process and, moreover, according to our experience, prone to produce much larger BDDs. We have resolved this issue by inverting only the edges of the original BDDs and by implementing a variant of `apply` that runs upwards from the leaves of the original BDDs, against the direction of the original BDD edges. Unfortunately, this solution cannot be as efficient as the normal `apply` because, unlike standard BDDs, the DAG that arises by inverting BDD edges is non-deterministic, which brings complications. Nevertheless, it still allows an efficient implementation of *pre* that works well in the case of our implementation.

However, we are facing a more fundamental problem: our algorithm can use `apply` to compute predecessors over the compact representation provided by BDDs only on the level of explicit automata in the leaves of terms. Symbols generated by projection during the evaluation of complex terms must be, on the contrary, enumerated explicitly. For instance, the projection $\pi_{\mathcal{X}}(t)$ with $\mathcal{X} = \{X_1, \ldots, X_n\}$ generates 2^n symbols, with no obvious option for reduction.

Our idea to overcome this explosion is to treat nodes of BDDs as regular automata states. Intuitively, this means replacing words over $\Sigma_{\mathcal{X}}$ that encode models of formulae by words over the alphabet $\{0, 1\}$: every symbol $\tau \in \Sigma_{\mathcal{X}}$ is replaced by the *string* τ over $\{0, 1\}$. Then, instead of computing a quotient over, e.g. the set $\pi_{\mathcal{X}}(\bar{0})$ of the size 2^n, we compute only quotients over the 0's and 1's. Each application of quotient takes us only one level down in the BDDs representing the transition relation of the automata in the leaves of the term. For every variable X_i, we obtain terms over nodes on the i-th level of the BDDs as -0 and -1 quotients of the terms at the level $i - 1$. The maximum number of terms in each level is thus 2^i. In the worst case, this causes roughly the same blow-up as when enumerating the "long" symbols. The advantage of this techniques is, however, that the blow-up can now be dramatically reduced by using subsumption to prune sets of terms on the individual BDD levels. We believe that treating nodes of BDDs as ordinary states is an idea worth further elaboration, and probably relevant also in the context of symbolic automata [DV14].

4.2.4. Combination of Symbolic and Explicit Algorithms

It is possible to replace sub-terms of a language term by a language-equivalent automaton built by the explicit algorithm before starting our symbolic algorithm. The main benefit of this is that the explicitly constructed automata have a simpler, flat structure and, most of all, can be minimized. The minimization, however, requires to explicitly construct the whole automaton, which might, despite the benefit of minimization, be a too large overhead. The combination hence represents a trade-off between the lazy evaluation and subsumption of the symbolic algorithm, and the minimization and flat automata structure of the explicit one.

The overall effect depends on the strategy of choice of the sub-formulae to be translated into automata, and, of course, on the efficiency of the implementation of the explicit algorithm (where we can leverage the extremely efficient implementation of MONA). In Section 4.3, we mention one particular strategy of choosing sub-formulae to be translated to automata that was efficient in our experiments, but we believe that there is a space for more involved heuristics in this are that could be further explored.

4.2.5. Anti-prenexing

Before we rewrite an input formula to a symbolic term, we first pre-process the formula based on several language-preserving identities. Particularly, we move quantifiers down based on rules (4.29)–(4.32), which we call the *anti-prenexing*. Moving a quantifier down in the abstract syntax tree of a formula speeds up the fixpoint computation induced by the quantifier. In effect, one costlier fixpoint computation is replaced by several cheaper computations in the sub-formulae. This is almost always helpful since if the original fixpoint computation unfolds, e.g. a union of two terms, the two fixpoint computations obtained by anti-prenexing will each unfold only one operand of the union. The number of union terms in the original fixpoint is roughly the product of the numbers of terms in the simpler fixpoints. Other rules, such as rule (4.31), reduce the scope of the quantified variables which obviously simplifies the fixpoints as well.

$$\exists X : \varphi \rightarrow \varphi \text{ iff } X \notin \mathit{free}(\varphi) \tag{4.29}$$
$$\exists X : \varphi \vee \psi \rightarrow (\exists X : \varphi) \vee (\exists X : \psi) \tag{4.30}$$
$$\exists X : \varphi \wedge \psi \rightarrow \varphi \wedge (\exists X : \psi) \text{ iff } X \notin \mathit{free}(\varphi) \tag{4.31}$$
$$\exists X : \varphi \wedge \psi \rightarrow (\exists X : \varphi) \wedge \psi \text{ iff } X \notin \mathit{free}(\psi) \tag{4.32}$$

Further, in order to push quantifiers even deeper into the formula syntax tree, we first reorder the formula by several heuristics (e.g. we reorder subtrees of a large conjuction to isolate occurrences of a quantified variable into a smaller subformula) and move negations down in the structure towards the leaves using De Morgan's laws. Note that we also experimented with using rule (4.33) based on distributivity laws, however, this only led to implementation complications, and, in the end, we did not measure any satisfying results.

$$\exists X : \varphi \wedge (\psi \vee \rho) \rightarrow (\exists X : \varphi \wedge \psi) \vee (\exists X : \varphi \wedge \rho) \tag{4.33}$$

We conclude that anti-prenexing is a powerful optimization and is not limited to our decision procedure only. In our experiments, we show that even the simple rules (4.29)-(4.32) can provide a speed-up of several orders of magnitude on some benchmarks. However, we believe that there is still a room for improvements, such as exploring other language-preserving formulae transformations that could lead to other cheap optimizations.

4.3. Experimental Evaluation

We have implemented the optimized procedure in a prototype tool GASTON[1]. Our tool uses the front-end of MONA to parse input formulae, to construct their corresponding abstract syntax trees, and to explicitly construct automata for sub-formulae (as mentioned in Section 4.2.4). We explored several heuristics for choosing the sub-formulae to be converted to automata by MONA. In the end, we convert only quantifier-free sub-formulae and negations of innermost quantifiers to automata since MONA can usually handle them without any state-space explosion. GASTON, together with all the benchmarks described below and their detailed results, is freely available [FHJ+16].

In the automata representation of MONA, MTBDDs are used to encode transition relations of automata by representing *post*-images of states. Since our approach computes *pre*-images of states, we had to deal with the issue of inverting the transition relation. For that we created an efficient wrapper over the MTBDDs of MONA (see Section 4.2.3).

We have compared GASTON's performance with that of MONA, DWINA implementing our previous approach [FHLV15], which we introduced in the Chapter 3, the TOSS tool implementing the method of [GK10], and the implementations of the decision procedures of [Tra15] and [DV14] (which we denote as COALG and SFA, respectively).[2] In our experiments, we consider formulae obtained from various formal verification tasks (in particular, formulae that were used to describe invariants of advanced data structures, which we described in Section 2.4) as well as parametric families of formulae designed to stress-test WS1S decision procedures.[3] We performed the experiments on a machine with the Intel Core i7-2600@3.4 GHz processor and 16 GiB RAM running Debian GNU/Linux.

A comparison of Gaston with Mona on Strand and UABE formulae. In Table 4.1, we show results of our experiments with formulae from the recent work of [ZHW+14] (denoted as `UABE` below), which uses WS1S to reason about programs with unbounded arrays of bounded elements (see Section 2.4.2). Table 4.2 gives results of our experiments with formulae derived from the WS1S-based shape analysis of [MQ11] (denoted as `Strand`; see Section 2.4.1). In the table, we use `sl` to denote `Strand` formulae over sorted lists and `bs` for formulae from verification of the bubble sort procedure. For this set of experiments, we considered MONA and GASTON only since the other tools were missing key features (e.g. atomic predicates) needed to handle the formulae. In the `UABE` benchmark, we used GASTON with the last optimization of Section 4.2.3 (treating MTBDD nodes as automata states) to efficiently handle quantifiers over large numbers of variables. In particular, without the optimization, GASTON hit 11 more timeouts. On the other hand, this optimization was not efficient (and hence not used) in `Strand`.

[1]The name was chosen to pay homage to Gaston, an Africa-born brown fur seal who escaped the Prague Zoo during the floods in 2002 and made a heroic journey for freedom of over 300 km to Dresden. There he was caught and subsequently died due to exhaustion and infection.

[2]We are not comparing with JMOSEL [TWMS06] as we did not find it available any more on the Internet.

[3]We note that GASTON currently does not perform well on formulae with many Boolean variables and M2L formulae appearing in benchmarks such as `Secrets` [KMS02] or `Strand2` [MPQ11, MPQ], which are not included in our experiments. To handle such formulae, further optimizations of GASTON such as MONA's treatment of Boolean variables via a dedicated transition are needed.

Table 4.1.: The comparison of Mona and Gaston on UABE benchmark. Each formula describes invariant or property of an array and was generated from the specification of UABE logic, which we described in the Section 2.4.2. We measured the time in seconds and space in overall number of automata states for Mona and overall number of symbolic terms for Gaston. In each row, we highlight the clear winner in bold and we use ∞if the runtime of a tool exceeded 2 minutes.

Formula	Mona		Gaston	
	Time	**Space**	**Time**	**Space**
a-a	**1.71**	30 253	∞	∞
ex10	**7.71**	131 835	12.67	82 236
ex11	4.40	2 393	**0.18**	4 156
ex12	**0.13**	2 591	6.31	68 159
ex13	**0.04**	2 601	1.19	16 883
ex16	**0.04**	3 384	0.28	3 960
ex17	3.52	165 173	**0.17**	3 952
ex18	**0.27**	19 463	∞	∞
ex2	0.18	26 565	**0.01**	1 841
ex20	1.46	1 077	**0.27**	12 266
ex21	**1.68**	30 253	∞	∞
ex4	**0.08**	6 797	0.50	22 442
ex6	**4.05**	27 903	22.69	132 848
ex7	0.90	857	**0.01**	594
ex8	7.69	106 555	**0.03**	1 624
ex9	7.16	586 447	9.41	412 417
fib	**0.10**	8 128	24.19	126 688

The tables compare the overall time (in seconds) the tools needed to decide the formulae, and they also try to characterize the sizes of the generated state spaces. For the latter, we count the overall number of states of the generated automata for Mona, and the overall number of generated sub-terms for Gaston. The tables contain just a part of the results, the full results can be found in [FHJ+16]. We use ∞ in case the running time exceeded 2 minutes, oom to denote that the tool ran out of memory, $+k$ to denote that we added k quantifier alternations to the original benchmark, and N/A to denote that the benchmark requires some key feature or atomic predicate unsupported by the given tool. On Strand, Gaston is mostly comparable, in two cases better, and in one case worse than Mona. On UABE, Gaston outperformed Mona on six out of twenty-three benchmarks, it was worse on ten formulae, and comparable on the rest. The results thus confirm that our approach can defeat Mona on many formulae in practice.

Table 4.2.: The comparison of Mona and Gaston on `Strand` benchmark. Each formula describes invariant of a linear data structure and was generated from the specification of Strand logic, which we described in the Section 2.4.1. We measured the time in seconds and space in overall number of automata states for Mona and overall number of symbolic terms for Gaston. In each row, we highlight the clear winner in bold.

Formula	Mona		Gaston	
	Time	**Space**	**Time**	**Space**
`bs-loop-else`	0.05	14 469	0.04	2 138
`bs-loop-if-else`	0.19	61 883	**0.08**	3 207
`bs-loop-if-if`	0.38	127 552	**0.18**	5 428
`sl-insert-after-loop`	**0.01**	2 634	0.36	5 066
`sl-insert-before-head`	0.01	678	0.01	541
`sl-insert-before-loop`	0.01	1 448	0.01	656
`sl-insert-in-loop`	0.02	5 945	0.01	1 079
`sl-reverse-after-loop`	0.01	1 941	0.01	579
`sl-search-in-loop`	0.08	23 349	0.03	3 247

A comparison of Gaston with other tools. The second part of our experiments concerns parametric families of WS1S formulae used for evaluation in [GK10, FHLV15, DV14], and also parametric versions of selected `UABE` formulae [ZHW$^+$14]. Each of these families has one parameter (whose meaning is explained in the respective works). Table 4.4 gives times needed to decide instances of the formulae for the parameter having value of 20. If the tools did not manage this value of the parameter, we give in parentheses the highest value of the parameter for which the tools succeeded. The full results are available in [FHJ$^+$16]. In this set of experiments, Gaston managed to win over the other tools on many of their own benchmark formulae. In the first six rows of Table 4.4, the superior efficiency of Gaston was caused mainly by anti-prenexing. It turns out that this optimization of the input formula is universally effective. When run on anti-prenexed formulae, the performance of the other tools was mostly comparable to that of Gaston. The last two benchmarks (parametric versions of formulae from `UABE`) show, however, that Gaston's performance does not stand on anti-prenexing only. Despite that its effect here was negligible (similarly as for all the original benchmarks from `UABE` and `Strand`), Gaston still clearly outperformed Mona. However, we could not compare with other tools on these formulae due to a missing support of the used features (e.g. first-order constants).

Table 4.3.: The comparison of all available tools on parametric families of formulae. Each row corresponds to a parametric family of formulae used to stress-test WS1S decision procedures. We measured the time in seconds. We use ∞ in case the running time exceeded 2 minutes, `oom` to denote that the tool ran out of memory, $+k$ to denote that we added k quantifier alternations to the original benchmark, and `N/A` to denote that the benchmark requires some key feature or atomic predicate unsupported by the given tool. We highlight the clear winners in bold. In case the tool hit the timeout or ran out of memory, we list in the parenthesis the highest value of the parameter of the family for which the tool succeeded.

Table 4.4.: Experiments with parametric families of formulae

Benchmark	MONA	dWiNA	TOSS	COALG	SFA	GASTON
`HornLeq` [DV14]	`oom`(18)	**0.03**	0.08	∞(08)	**0.03**	**0.01**
`HornLeq` (+3) [DV14]	`oom`(18)	∞(11)	0.16	∞(07)	∞(11)	**0.01**
`HornLeq` (+4) [DV14]	`oom`(18)	∞(13)	**0.04**	∞(06)	∞(11)	**0.01**
`HornIn`[GK10]	`oom`(15)	∞(11)	0.07	∞(08)	∞(08)	**0.01**
`HornTrans` [FHLV15]	86.43	∞(14)	`N/A`	`N/A`	38.56	**1.06**
`SetSingle` [FHLV15]	`oom`(04)	∞(08)	0.10	`N/A`	∞(03)	**0.01**
`Ex8` [ZHW+14]	`oom`(08)	`N/A`	`N/A`	`N/A`	`N/A`	**0.15**
`Ex11(10)` [ZHW+14]	`oom`(14)	`N/A`	`N/A`	`N/A`	`N/A`	**1.62**

4.4. Conclusion and Future Work

We have presented a novel WS1S decision procedure based on symbolic, term-based representation of the languages of formulae, with on-the-fly emptiness testing, optimized by, e.g. lazy evaluation and subsumption. Our experiments proved that the approach is competitive and often better than state-of-the-art methods, including MONA.

But let us emphasize that, like with MONA, optimizations play a crucial role for the efficiency of our tool—without them, the basic approach is much less efficient. We believe there are many further optimization possible. Let us briefly enumerate some of the further possible optimization opportunities. First, our utilization of BDDs is not optimal since MONA gives us efficient *post* only. We would greatly benefit from an explicit procedure producing automata encodings with efficient *pre*. Second, as we mention in Section 4.3, performance of our tool could be improved by a specialized treatment of Boolean variables. This type of variables is common in many challenging benchmarks and so its support in WS1S tools is especially demanded. A plausible solution is to integrate our approach with SAT/SMT technology or to adapt techniques of MONA that dedicates special transitions in automata for Boolean variables. At last, another logical step would be to use abstraction over the language terms or adapt simulation techniques.

Note that many of the optimizations are of a heuristic nature. As we have seen in our experimental evaluation, we used different sets of optimization for some benchmarks as they had different effects. It would be helpful to know more about the cases where they can be successfully applied.

We would also like to generalize or at least extend our approach to other kinds of logics, such as M2L(str) or WS2S. The latter, has many practical applications. In particular, the generalization to WS2S would allow us to define, e.g. invariants of tree-like structures. However, handling more complex formulae, e.g. in the logic M2L(str), such as those mentioned in Sec. 4.3, would obviously require specific optimizations.

Part II.

Using Static Analysis for Performance Analysis

5. From Shapes to Amortized Complexity

The state of the art of static performance analysis of C and C++ programs focuses mostly on resource bounds and termination analysis of integer programs. While this field is currently well-established, works focusing on heap-manipulating programs are rather rare since they require precise analysis of the shape of the heap. Moreover, most of the works on shape analysis are usually limited to linear structures or have a hard-coded support for a single data structure.

We can further motivate the research by the experimental evaluation of the resource bound analyser Loopus [SZV17], where authors analysed a large number of C programs. Loopus computes resource bounds based on the updates to integer variables, but it has only a limited support for pointers. One of the results of their experiments was that missing pointer and shape analysis was the most frequent reason for a failure to compute a resource bound. In [SZV17], the authors report that they obtained bounds for 753 of the 1659 functions in their extensive benchmark (45%), and that by a simple (but unsound) shape analysis they were able to increase the number to 1185 (71%).

Researchers usually transform the input heap-manipulating program into an integer one that is equal from the point of view of its termination or complexity. However, one has to first define the so-called shape numerical measures (or shape norms), that simulate the original data structures in the universe of integer programs. An example of a shape norm can be, e.g. the number of nodes in a tree, the length of a singly-linked list, or the height of a tree. Defining a suitable class of shape norms and inferring how their values change upon execution of program statements is the biggest challenge of such a transformation. Some classes are restricted to concrete data structures, such as singly-linked lists or trees. Other are applicable for a limited range of program constructions only, and cannot show resource bounds in many cases, e.g. when the resource bounds depend both on the shape of the heap and some integer constraints. Finally, many classes are not fully automatically usable and require manual involvement of the user.

In this chapter, we aim at automated resource bound analysis of heap-manipulating programs. We focus on analysing programs with dynamic data structures as they can be frequently found in systems code such as in operating system kernels or compilers. Good performance is a major concern in system code and has led to the use of highly advanced data structures such as, e.g. red-black trees, priority heaps, or lock-free linked lists — data structures that both are complex and prone to introducing errors. Thus, automated tool support is needed to increase the reliability of those systems and it can in turn lead to a better user experience. Resource bound analysis of programs with data structures has been addressed only by a few publications [GLS09, HR13, Atk11, AAG^{+}12, FG17].

We first define a new parametric class of shape norms expressing the distance between two distinct points (memory cells pointed by some program variables and/or selector chains (so-called access paths); we can refer to these as "pointers") in the shape over some selector paths (such as `left`, `right` or `next` field), which conforms to a (restricted) regular expression. Moreover, we show how one can automatically derive a set of shape norms from the control flow graph (CFG) of a program. We strive to get a small set of the norms such that the analysis is as efficient as possible. If this set turns out not to be sufficient, it can later be (automatically) refined. This way the approach becomes fully automated. Based on this new class of shape norms, we then propose an approach to resource bounds analysis that exploits state-of-the-art shape analysers to transforms the input heap-manipulating program to a corresponding integer program. The resulting integer program can then be analysed similarly as in other existing works in the area. Later we show, we improve on earlier results along several dimensions aiming at the automated resource bounds analysis of heap-manipulating programs that cannot be handled by existing approaches.

Overview of our approach. Our analysis works in three major steps. We first run a shape analysis and annotate the program with shape invariants. Second, using the results from the shape analysis, we create a corresponding integer abstraction of the program based on numeric information about the heap. Finally, we perform resource bound analysis purely on the resulting integer program.

The integer abstraction is based on our class of *shape norms*, i.e. numerical measures on dynamic data structures (e.g. the length of a linked list). Our first contribution in this chapter is the definition of a class of shape norms that expresses the longest distance between two points of interest in a shape graph defined in terms of basic concepts from graph theory. We propose a class of norms that are parametric by the program under analysis and that are extracted in a pre-analysis (with a possibility of extending the initial set of tracked norms during the subsequent analysis); the extracted norms then correspond to the selector paths found in the program.

The second contribution is a calculus for our class of shape norms that allows us to derive how the norms change along a program statement, i.e. if the norm is incremented, resp. decremented, or reset to some other expression. The calculus consists of two kinds of rules. (1) Rules that allow one to directly infer the change of a norm and do not need to take any additional information into account. (2) Rules that rely on the preceding shape analysis; the shape information is mainly used there for (a) dealing with pointer aliasing and (b) deriving an upper bound on the value of a norm from the result of the shape analysis (if possible). We point out that rules of the second kind encapsulate the points of the analysis where information about the shape is needed, and thus describe the minimal requirements on the preceding shape analysis. We believe that this separation of concern also allows one to use various other shape analysers if they satisfy the given criteria.

When creating the integer abstraction of the given heap-manipulating program, we could use all shape norms that we extracted from the program. However, we have an additional pre-analysis phase that eliminates norms that are not likely to be useful for the later bounds analysis. This reduction of norms has the benefit that it keeps the number of variables in the integer abstraction small. The number of extracted norms can be quadratic in the size of the program in the worst case, and adding quadratically many variables can be prohibitively expensive. The pre-analysis is therefore crucial to the efficiency of the later bound analysis. Moreover, the smaller number of additional variables increases the readability of the resulting integer abstraction and simplifies the development and debugging of subsequent analyses.

Finally, we perform resource bound analysis on the created integer abstraction. This design decision has two advantages. First, we can leverage the existing research on resource bound analysis for integer programs and do not have to develop a new bound analysis at all. Second, being able to analyse not only the shape but also integer changes has the advantage that we can analyse programs which mix integer iterations with data structure iterations; we illustrate this point on the flagship example of [Atk11], which combines iteration over data structures and integer loops in an intricate way (see Section 5.1 for details). So far, no other approach has inferred precise resource bounds for this loop.

Implementation and Experiments. We implemented the generation of the integer program on top of the shape analyser Forester [HHL$^+$15b]. We use the Loopus tool [SZV17] for inferring the computational complexity of the obtained integer abstractions. Our experimental evaluation demonstrates that the combination of these tools can yield a powerful analysis. We report on results for complex heap manipulating programs that could not be handled by previous approaches as witnessed by experimental evaluation against the tools AProVE [AFHG15] and COSTA [AAG$^+$12]. We remark that our implementation leverages the strengths of both Forester and Loopus. We inherit the capabilities of Forester to analyse complex data structures, and report on analysis results for doubly-linked lists, trees, or skip-lists. Moreover, our analysis of shape norms and its changes is precise enough to leverage the capabilities of Loopus for amortized complexity analysis — we report on the amortized analysis of the flagship example of [Atk11] in more details in Section 5.1, whose correct linear bound has to the best of our knowledge never been inferred fully automatically.

Related work on Resource Bounds Analysis. A majority of the related approaches is based on well-established approach that derives an integer program from an input heap-manipulating program followed by a dedicated analysis (e.g. termination or resource bounds) for integer programs. The transformation, however, has to be done conservatively, i.e. the derived integer program needs to simulate the original heap-manipulating program such that the results (i.e. the bounds) for the integer program hold for the original program. The related approaches then differ in the considered class of numeric measures on the heap, the data structures that can be analysed or the degree of automation.

Several approaches have targeted restricted classes of data structures such as singly-linked lists [BBH$^+$11, DBCO06, LQ06, YB02, BCC$^+$07] or trees [Rug04, HIRV07]. It is, however, unclear how to generalise these results to more complex data structures which require different numeric measures or combinations thereof.

A notable precursor to our work is the framework of [MTLT10] implemented in the Thor tool [MTLT08], which describes a general method for deriving integer abstractions of data structures. The automation of Thor, however, relies on the user for providing the shape predicates of interest (to the best of our knowledge, the implementation only supports list predicates; further predicates have to be added by the user). Further, we found out during our initial experiments that Thor needlessly tracks shape sizes not required for a later termination or bounds analysis. This can quickly lead to a bloat of the program under analysis.

A general abstract interpretation-style framework for combining shape and numerical abstract domains is described in [GLS09]. The paper focuses on tracking the partition sizes, i.e. the only considered norm is the number of elements in a data structure. Our framework is orthogonal to this approach: we can express different norms, e.g. the height of a tree, which cannot be expressed in [GLS09]; on the other hand, we use numeric information only in the second stage of the analysis which can be less precise than the reduced product construction of [GLS09].

An automated approach to amortized complexity analysis of object-oriented heap-manipulating programs is discussed in [HR13]. The approach is based on the idea of associating a potential to (refinements of) data structure classes. Typing annotations allow to derive a constraint system which is then solved in order to obtain valid potential annotations. However, the implementation is currently limited to linear resource bounds and appears to be restricted to list-like data structures.

The idea of using potentials for the analysis of data structures is also investigated in [Atk11]. The author extends the separation logic with resource annotations exploiting the idea of separation in order to associate resource units to every memory cell, resulting in an elegant Hoare-logic for resource analysis. But, the suggested approach is currently only semi-automated requiring the user to provide shape predicates and loop invariants manually.

In [AAG$^+$12], the authors propose an automated resource analysis for Java programs, implemented in the COSTA tool. Their technique is based on abstracting arrays into their sizes and linked structures into the length of the longest chain of pointers terminated by the `NULL` pointer, followed by the construction and solving of a system of recurrence equations. However, cyclic lists and more complicated data structures such as DLLs, are, to the best of our knowledge, out of the capabilities of this technique as they require more general numeric size measures.

The recent paper [FG17] investigates the automated resource analysis for Java programs and reports on its implementation in the AProVE tool. It is based on first translating a program to an integer transition system, and then using a bounds analyser to infer the complexity. The technique makes use of a single size-measure which is the number of nodes reachable from the heap node of interest together with the sum of all reachable integer cells. This norm is orthogonal to the norms considered in our work.

However, note that our norms do not take the content of integer cells into account and thus can be less precise when the termination depends on such integer cells. On the other hand, the norm of [FG17] does often not correspond to the size of interest: for example, in the case of an iteration over the top-level list of a list of lists, the relevant norm is the length of the top-level list and not the number of elements of all involved lists; similarly in case of a search in a sorted tree: the relevant size measure is the height of a tree and not the number of elements. Moreover, it is unclear how the norm of [FG17] deals with cyclic data structures; while the number of reachable elements is well-defined, it is unclear whether (and how) the norm changes when a pointer is advanced because the number of reachable nodes does not change.

Contributions. We summarize our contributions to resource bounds analysis:

1. In comparison with related approaches, we consider a wider class of shape norms.
2. We develop a calculus for deriving the numeric changes of the considered shape norms. The rules of our calculus precisely identify the information that is needed from a shape analyser. We believe that this definition of minimal shape information is useful for development of future resource bound analysis tools for programs with recursive data structures as well.
3. Our norms are not fixed in advance but derived from the program to be verified: we define a pre-analysis that reduces the number of considered norms. To our experience, this reduction is especially affecting the efficiency of the underlying resource bounds analysers, but it is also useful for reporting the derived integer abstraction to the user.
4. We demonstrate in an experimental validation that we obtain a powerful analysis. We report on iterations over complex data structures that could not be analysed before and discuss our fully-automated amortised analysis of a challenging example from the literature (c.f. Section 5.1).

Outline. The chapter is structured into six sections. In Section 5.2, we introduce basic preliminaries, such as the considered program model. We then propose a new class of shape norms in Section 5.3. Based on this new class of norms we propose a resource bounds analysis for heap-manipulating programs. We first illustrate the analysis in Section 5.1 on a showcase example from the work of [Atk11] and describe the analysis in more details in the following Section 5.4. We briefly describe the implementation details and the experimental evaluation on several benchmarks in Section 5.5. Finally, we conclude the chapter with Section 5.6 and propose further future research directions.

5.1. Running Example

We will illustrate how our approach fully automatically infers the linear complexity of the `mergeInner` function in Fig. 5.1, the inner loop of an in-place linked-list merge sort implementation, the flagship example of [Atk11].

```
Function void mergeInner(Node* list, int k)
    Node* p = list;
    Node* tail = NULL;
    Node* list = NULL;
    while $p \neq$ NULL $\wedge\ k > 0$ do
        Node* q = p;
        int j = k;
        while $j > 0 \wedge q \neq$ NULL do
            q = q→next;
            j−−;
        Node* pstop = q;
        qsize = k;
        while $(p \neq pstop) \vee (qsize > 0 \wedge q \neq$ NULL$)$ do
            Node* e;
            if $(p == pstop)$ then
                e = q;
                q = q→next;
                qsize−−;
            else if $(qsize == 0 \vee q ==$ NULL$)$ then
                e = p;
                p = p→next;
            else if $(p \rightarrow data \leq q \rightarrow data)$ then
                e = p;
                p = p→next;
            else
                e = q;
                q = q→next;
                qsize−−;
            if $(tail \neq$ NULL$)$ then
                tail→next = e;
            else
                list = e;
            tail = e;
        p = q;
    if $(tail ==$ NULL$)$ then
        return;
    else
        tail→next = NULL;
```

Figure 5.1.: The inner loop of an in-place merge sort implementation, taken from [Atk11]

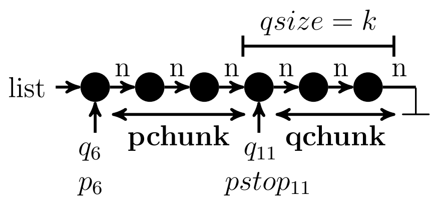


Figure 5.2.: Illustration of the initial phase of `mergeInner` function at *line 11.* The singly-linked list of length six is divided into two chunks (p-chunk and q-chunk) of length three. The variable *pstop* serves as a sentinel for the p-chunk, while the value $qsize = k$ serves as bound for the q-chunk. At *line 34*, p-chunk and q-chunk will be merged into one chunk of length 6. We use index i to denote the state of the variable at *line i*.

We first shortly describe the function and discuss the challenges it poses to an automatic resource bounds analysis. The outer loop at *line 5* iterates over the singly-linked list *list* in steps of size $2k$. The inner loop at *line 8* progresses k steps through the `next`-selector from pointer q. At *line 11* we remember the original position of q in the variable *pstop* and as consequence we have prepared two partitions of list *list*: (1) the partition of size k between variables p and *pstop* (which is equal to the position of the variable q), we refer to this partition as p-chunk, and (2) a partition which starts at q and has size *qsize* equal to k, we will refer to this partition as q-chunk. Figure 5.2 illustrates the state of the input singly-linked list at this point.

The second inner loop at *line 13* implements the ordered (ascending order) merging of the elements of both chunks. It compares the current element of the p-chunk with the current element of the q-chunk. In *line 26* an element `p`$\rightarrow$`data` of chunk p is swapped with an element `q`$\rightarrow$`data` of chunk q, if `p`$\rightarrow$`data` is greater than `q`$\rightarrow$`data`. Assuming that both chunks were already sorted, their merged chunk of size 2k will be sorted when reaching the *line 34*, which concludes one iteration of the outer loop.

The resulting list of merged chunks of size $2k$ is constructed using the variables *tail* (pointer to the tail of the list), e (pointer to current front element of either p-chunk or q-chunk) and *list* (pointer to the head of the list) on *lines* $28 - 32$. At *line 36* we `NULL` terminate the list of merged chunks. At this point, n/k chunks of length k are merged. This whole process is illustrated in Figure 5.3.

Complexity of the `mergeInner` function. Given that the outer loop iterates over the list in steps of size $2k$ while the first inner loop partitions the list into k-chunks and the second inner loop iterates over these chunks, the overall complexity of the example is linear in the size of the *list*. The example implements merge sort if we assume that `mergeInner` is consecutively called with $k = 1, 2, 4, \ldots, \frac{n}{2}$, where n is the size of *list*. `mergeInner` is thus called $\log_2(n) - 1$ times, and we obtain the well-known merge sort complexity of $n \log(n)$.

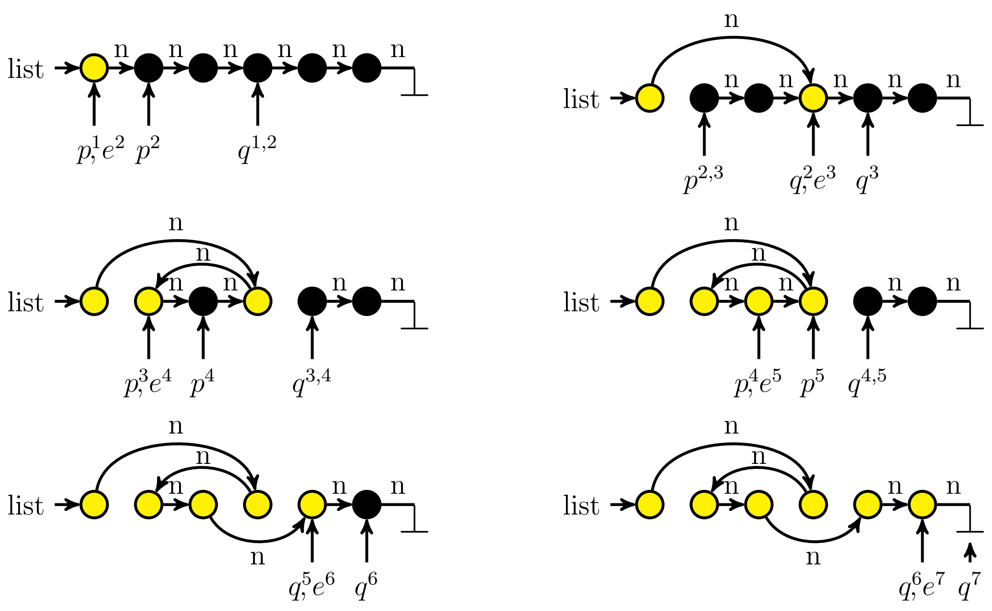


Figure 5.3.: Illustration of one run of the loop at *line 13* on a singly-linked list from the Figure 5.2. The loop iterates six times. In each iteration we denote already merged and not-yet-merged nodes by yellow and black colour respectively. We use upper index i to denote the state of the variable at the start of ith iteration of the cycle. Variables p and q points to the beginning of the p-chunk and q-chunk, and the variable e points to the currently merged node. We omit some of the variables from the illustration, such as the *pstop* variable, from the picture.

This example poses two main challenges to an automated bound analysis since (1) it requires precise shape analysis to track the interplay between the chunks of the list pointed to by variables p and q, as well as handle the in-place modification of the traversed list and (2) given an input singly-linked list of size n, in the worst case, k may be n and the inner loops at *line 8* and *line 13* may thus iterate n-times on a single execution of the outer loop, which itself can be iterated n-times for the case $k = 1$, nevertheless, the overall complexity is $\mathcal{O}(n)$. However, the naive reasoning would lead to imprecise $\mathcal{O}(n^2)$ complexity.

The tools APROVE and COSTA could not infer the linear complexity: COSTA times out after 5 minutes of computation and APROVE infers a quadratic complexity for the example. In the following we describe how our approach infers the linear complexity of the `mergeInner` function, as will discussed in the next sections.

Deriving of suitable norms. First, we derive the following tracked shape norms $\mathcal{N}_C$ from the code (as is outlined in Section 5.3.2): For the outer loop at *line 5*, we derive the single norm $p\langle\text{next}^*\rangle$`NULL`, for the first inner loop at *line 8*, we derive the norm $q\langle\text{next}^*\rangle$`NULL` and for the last inner loop at *line 13*, we derive the norms $q\langle\text{next}^*\rangle$`NULL`, $p\langle\text{next}^*\rangle pstop$, $pstop\langle\text{next}^*\rangle p$

Generation of integer program. We generate an integer program using the initial set of tracked norms $\mathcal{N}_c = \{p\langle\text{next}^*\rangle\texttt{NULL}, q\langle\text{next}^*\rangle\texttt{NULL}, p\langle\text{next}^*\rangle pstop, pstop\langle\text{next}^*\rangle p\}$[1]. Note that during the generation the initial set of norm candidates $\mathcal{N}_c$ is extended. As an example, consider the instruction $pstop = q$ in line 11. To model the effect on norms of this instruction we add the norm $p\langle\text{next}^*\rangle q$, derived from the norm $p\langle\text{next}^*\rangle pstop \in \mathcal{N}_c$ to $\mathcal{N}_c$, which is then tracked as well. At the end of the analysis we track set of norms $\mathcal{N}_c = \{p\langle\text{next}^*\rangle\texttt{NULL}, q\langle\text{next}^*\rangle\texttt{NULL}, p\langle\text{next}^*\rangle pstop, pstop\langle\text{next}^*\rangle p, p\langle\text{next}^*\rangle q, list\langle\text{next}^*\rangle\texttt{NULL}\}$. The resulting numerical instructions are shown in Algorithm 5.1, highlighted in red. We include the original program statements for better understanding. However, our approach runs the bound analyser only on the integer program which results from removing all pointer instructions from Algorithm 5.1[2].

Bounds Analysis. The integer program is analysed by the underlying bounds analyser (in our case, the Loopus tool), which computes the bounds on program loops. Due to the nature of integer program construction, an upper bound on the integer program is also an upper bound on the original program. We give a brief sketch on how Loopus infers a bound for our example (we refer the reader to [SZV17] for details on the used algorithm).

For the main loop at *line 5* we can use the norm $p\langle\text{next}^*\rangle\texttt{NULL}$ as the bound. The norm is first initialized to the value of the norm $list\langle\text{next}^*\rangle\texttt{NULL}$ and it is never increased (despite the swapping operations between p and q on *lines* 6 and 34) and it is decremented at least once at every iteration — since we assume that $k > 0$, then either in the inner loop at *line 8* the instruction at *line 9* is executed at least once (and hence $p\langle\text{next}^*\rangle\texttt{NULL}$ is decremented after execution of *line 34*) or the loop is not executed at all which implies that $q\langle\text{next}^*\rangle\texttt{NULL} == 0$ and hence $p\langle\text{next}^*\rangle\texttt{NULL} == 0$ will hold at *line 34*. Hence, the outer loop has linear complexity in size of the $list\langle\text{next}^*\rangle\texttt{NULL}$. Now a similar reasoning can be applied for the loop at *line 8*, for which we can use the norm $q\langle\text{next}^*\rangle\texttt{NULL}$ as the bound. $q\langle\text{next}^*\rangle\texttt{NULL}$ is initialized to the size of $p\langle\text{next}^*\rangle\texttt{NULL}$, hence the bound is linear in size of $list\langle\text{next}^*\rangle\texttt{NULL}$ as well.

However, for the inner loop at *line 13*, we have to use the combination of norms $p\langle\text{next}^*\rangle pstop$ and $q\langle\text{next}^*\rangle\texttt{NULL}$ in order to compute bounds. The size of $q\langle\text{next}^*\rangle\texttt{NULL}$ is bounded by the size of $list\langle\text{next}^*\rangle\texttt{NULL}$ and norm $p\langle\text{next}^*\rangle pstop$ is always set to 0 at *line 6* and is incremented on each execution of the instruction at *line 9*. This increment can be executed in maximum the same number of times as the loop at *line 8*. Thus, the number of increments by one of $p\langle\text{next}^*\rangle pstop$ can be bounded by the size of $list\langle\text{next}^*\rangle\texttt{NULL}$ and the overall bound on $p\langle\text{next}^*\rangle pstop$ is $0 + list\langle\text{next}^*\rangle\texttt{NULL} \cdot 1$, i.e. it is bounded by $list\langle\text{next}^*\rangle\texttt{NULL}$ as well. Thus, in worst case the loop at *line 13* is executed $(p\langle\text{next}^*\rangle pstop + q\langle\text{next}^*\rangle\texttt{NULL})$-times, hence the bound of the loop is $2 \cdot list\langle\text{next}^*\rangle\texttt{NULL}$. Since, all the loops are linearly bounded in the length of the list *list*, the overall complexity of `mergeInner` is linear in the length of *list*.

[1]Note that we can reduce the set $\mathcal{N}_c$ and remove the norm $pstop\langle\text{next}^*\rangle p$, since the position of the *pstop* variable is not changed during the algorithm.

[2]Note, however, that the original instructions would have no effect on our backed resource bounds analyser, Loopus, since it ignores these instructions

Algorithm 5.1: Numerical abstraction of the Algorithm 5.1.

```
Function void mergeInner(Node* list, int k)
    Node* p = list;
    $p\langle next^*\rangle NULL = list\langle next^*\rangle NULL$;
    Node* tail = NULL;
    Node* list = NULL;
    while $p\langle next^*\rangle NULL \neq 0 \wedge p \neq NULL \wedge k > 0$ do
        Node* q = p;
        $q\langle next^*\rangle NULL = p\langle next^*\rangle NULL$; $p\langle next^*\rangle q = 0$;
        int j = k;
        while $j > 0 \wedge q\langle next^*\rangle NULL \neq 0 \wedge q \neq NULL$ do
            q = q→next;
            $q\langle next^*\rangle NULL --$; $p\langle next^*\rangle q ++$;
            j−−;
        Node* pstop = q;
        $p\langle next^*\rangle pstop = p\langle next^*\rangle q$;
        qsize = k;
        while $(p\langle next^*\rangle pstop \neq 0 \wedge p \neq pstop) \vee (qsize > 0 \wedge q\langle next^*\rangle NULL \neq 0 \wedge q \neq NULL)$ do
            Node* e;
            if $(p\langle next^*\rangle pstop = 0 \vee p == pstop)$ then
                e = q;
                q = q→next;
                $q\langle next^*\rangle NULL --$; $p\langle next^*\rangle q ++$;
                qsize−−;
            else if $(qsize == 0 \vee q\langle next^*\rangle NULL = 0 \vee q == NULL)$ then
                e = p;
                p = p→next;
                $p\langle next^*\rangle NULL --$; $p\langle next^*\rangle pstop --$; $p\langle next^*\rangle q --$;
            else if $(*)$ then
                e = p;
                p = p→next;
                $p\langle next^*\rangle NULL --$; $p\langle next^*\rangle pstop --$; $p\langle next^*\rangle q --$;
            else
                e = q;
                q = q→next;
                $q\langle next^*\rangle NULL --$;$p\langle next^*\rangle q ++$;
                qsize−−;
            if $(tail \neq NULL)$ then
                tail→next = e;
            else
                list = e;
            tail = e;
        p = q;
        $p\langle next^*\rangle NULL = q\langle next^*\rangle NULL$; $p\langle next^*\rangle pstop = q\langle next^*\rangle NULL$; $p\langle next^*\rangle q = 0$;
    if $(tail == NULL)$ then
        return;
    else
        tail→next = NULL;
```

5.2. Preliminaries

First we introduce the basic notions, the program model, as well as reachable memory configurations, which we later use to infer changes of our new class of shape norms.

5.2.1. Program Model

For the rest of the chapter, we will use $\mathbb{V}_p$ to denote the set of *pointer variables*, $\mathbb{V}_i$ the set of *integer variables*, $\mathbb{S}_p$ the set of *pointer selectors* (or fields) of dynamic data structures, and $\mathbb{S}_i$ the set of *integer selectors* (or data fields). We assume all these sets to be finite and mutually disjoint. Let $\mathbb{V} = \mathbb{V}_p \cup \mathbb{V}_i$ be the set of all program variables and $\mathbb{S} = \mathbb{S}_p \cup \mathbb{S}_i$ be the set of all selectors. Finally, let `NULL` denote the *null* pointer and assume that $\texttt{NULL} \notin \mathbb{V} \cup \mathbb{S}$.

We consider *pointer manipulating program statements* from the set $\textsc{stmts}_p$ that are generated by the following grammar where $x, y \in \mathbb{V}_p$, $z \in \mathbb{V}_p \cup \{\texttt{NULL}\}$ and $sel \in \mathbb{S}_p$:

$$\begin{aligned}\texttt{stmt}_p ::= {} & x = z \mid x = y \to sel \mid x \to sel = z \mid \\ & x = \texttt{malloc()} \mid \texttt{free(}x\texttt{)} \mid x == z \mid x \neq z\end{aligned}$$

Further, we consider *integer manipulating program statements* from the set $\textsc{stmts}_i$ that are generated by the following grammar where $x \in \mathbb{V}_i$, $y \in \mathbb{V}_p$, $sel \in \mathbb{S}_i$, $c \in \mathbb{Z}$, and f is an integer operation (more complex statements could easily be added too):

$$\begin{aligned}\texttt{stmt}_i &::= x = op \mid x = f(op, op) \mid y \to sel = op \mid x == op \mid x \neq op \\ op &::= c \mid x \mid y \to sel\end{aligned}$$

Finally, we let $\textsc{stmts} = \textsc{stmts}_p \cup \textsc{stmts}_i$.

Control-flow graphs. A *control-flow graph* (CFG) is a tuple $G = (\textsc{Loc}, T, l_b, l_e)$ where $\textsc{Loc}$ is a finite set of program *locations*, $T \subseteq \textsc{Loc} \times \textsc{stmts} \times \textsc{Loc}$ is a finite set of *transitions* (sometimes also called *edges*), $l_b \in \textsc{Loc}$ is the *initial (starting) location*, and $l_e \in \textsc{Loc}$ is the *final location*.

Let $G = (\textsc{Loc}, T, l_b, l_e)$ be a CFG. A path in G of length $n \geq 0$ is a sequence of transitions $t_0 \ldots t_n = (l_0, st_0, l_1)(l_1, st_1, l_2) \ldots (l_n, st_n, l_{n+1})$ such that $t_i \in T$ for all $0 \leq i \leq n$. We denote the set of all such paths by Φ_G. For a given location l, we denote by Φ_G^l the set of paths where $l_0 = l$, i.e. all paths starting from the location l. Given locations $l_1, l_2 \in \textsc{Loc}$, we say l_1 dominates l_2 (and denote it by $l_1 \succ l_2$) iff all paths to l_2 in $\Phi_G^{l_b}$ lead through l_1. We call a transition $(l, st, h) \in T$ a *back-edge* iff $h \succ l$. We call the location h a *loop header* and denote the set of its back-edges as T_h. Back-edges are fundamental to resource bounds analysis, since we can reduce the computing the bounds on the number of iterations of a loop to computing the bounds on the number of firing of loop back-edges. Further, we denote the set of all loop headers as $\text{LH} \subseteq \textsc{Loc}$. Note that, for a loop header h_n of a loop nested in some outer loop with a loop header h_o, we have $h_o \succ h_n$.

Loops. Given a CFG $G = (\textsc{Loc}, T, l_b, l_e)$ with a set of loop headers LH, a loop L with a header $h_L \in \text{LH}$ is the sub-CFG $L' = (\textsc{Loc}', T_{|\textsc{Loc}'}, h_L, h_L)$ where $\textsc{Loc}' = \{l \in \textsc{Loc} \mid \exists n \geq 0\ \exists (l_0, st_0, l_1) \dots (l_n, st_n, l_{n+1}) \in \Phi_G : l_0 = l_{n+1} = h_L \wedge (\exists 0 \leq i \leq n : l = l_i) \wedge (\forall 1 \leq j \leq n : h_L \succ l_j)\}$, i.e., the set of locations on cyclic paths from h_L (but not crossing the header of any outer loop in which L might be nested), and $T_{|\textsc{Loc}'}$ is the restriction of transitions T to $\textsc{Loc}'$ only. We will denote the set of all program loops as $\mathcal{L}$. Note that the set of loop headers LH can be divided according to the nesting level of particular loops. By $\mathcal{LH}_0$ we denote the loop headers corresponding to the topmost loops and then for each loop header $l_h \in \mathcal{LH}_i$, we will denote by $\mathcal{LH}_{i+1}(l_h)$ all loop headers corresponding to the nested loops of the loop represented by header l_h.

5.2.2. Memory Configurations

Let $\mathbb{V}_p$, $\mathbb{V}_i$, $\mathbb{S}_p$, and $\mathbb{S}_i$ be sets of pointer variables, integer variables, pointer selectors, and integer selectors, respectively, as defined in the previous subsection. We define *memory configurations*, i.e., *shapes*, as triples $s = (M, \sigma, \nu)$ where (1) M is a finite set of memory locations (or cells), $\texttt{NULL} \notin M$, $M \cap \mathbb{Z} = \emptyset$, (2) $\sigma : (M \times \mathbb{S}_p \to M \cup \{\texttt{NULL}\}) \cup (M \times \mathbb{S}_i \to \mathbb{Z})$ is a function defining values of selectors, and (3) $\nu : (\mathbb{V}_p \to M \cup \{\texttt{NULL}\}) \cup (\mathbb{V}_i \to \mathbb{Z})$ is a function defining values of program variables. We denote the set of all such shapes by $\mathcal{S}$. Note that a shape is basically an oriented graph, also called a *shape graph*, with nodes from $M \cup \mathbb{Z} \cup \{\texttt{NULL}\}$, edges labeled by selectors, and some of the nodes referred to by the program variables. For simplicity, we do not explicitly deal with undefined values of pointers in what follows. For the purposes of resource bounds analysis, they can be considered equal to null values. We will later propose a resource bounds analysis that first runs the shape analysis on the given heap-manipulating program. So if the program may crash due to using them, we assume this to be revealed by the shape analysis phase.

Let us assume the shape analyser that works with a set $\mathcal{A}$ of *abstract shape representations* (ASRs), which can be automata, formulae, symbolic graphs, etc. This means that each ASR $A \in \mathcal{A}$ represents a (finite or infinite) set of shapes $[\![A]\!] \subseteq \mathcal{S}$. Allowing for disjunctive abstract representations, we assume that we can run the shape analyser and, as result, it will label each location of the CFG of a program by a set of ASRs overapproximating the set of shapes reachable at that location. Moreover, we assume that the shape analyser can infer a special successor relation between ASRs whenever they label locations linked by a transition s.t. the statement of the transition may be executed between some shapes encoded by the ASRs. The successor relation allows us to analyse the shape before and after the execution of a program statement. This leads to a notion of annotated CFGs defined below.

Annotated CFGs. An *annotated CFG* (ACFG) Γ is a triple $\Gamma = (G, \lambda, \rho)$ where $G = (\textsc{Loc}, T, l_b, l_e)$ is a CFG, $\lambda : \textsc{Loc} \to 2^{\mathcal{A}}$ is a function mapping locations to sets of ASRs generated by the underlying shape analyser for the particular locations, and $\rho \subseteq (\textsc{Loc} \times \mathcal{A}) \times (\textsc{Loc} \times \mathcal{A})$ is a *successor relation* on pairs of locations and ASRs where $((l_1, A_1), (l_2, A_2)) \in \rho$ iff $A_1 \in \lambda(l_1)$, $A_2 \in \lambda(l_2)$, and there is a transition $(l_1, st, l_2) \in T$ and shapes $s_1 \in [\![A_1]\!]$, $s_2 \in [\![A_2]\!]$ such that st transforms s_1 into s_2. We show an example of ACFG in Figure 5.4.

For the relation ρ, we will use ρ^* and ρ^+ as usual.

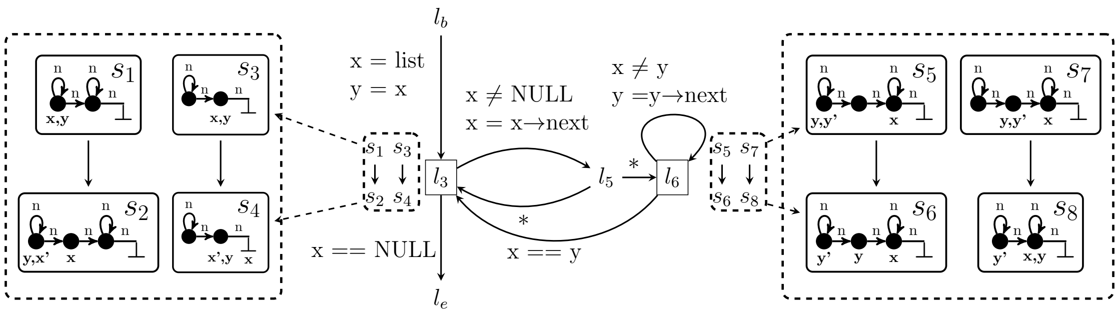


Figure 5.4.: An example of Annotated Control-Flow Graph for two nested loops that iterate over a singly-linked lists. Each location $l \in$ Loc is annotated with ASRs, which are shown in the dashed rectangles. Each ASR $s \in \mathcal{A}$ is depicted in simplified way in the full rectangles, where the self loops on the nodes represent abstraction, i.e. that the black node represents a segment of k nodes for some $k \in \mathbb{N}$. We denote that $((l_1, s_1), (l_2, s_2)) \in \rho$ as $s_1 \to s_2$, and, finally, we use x' and y' to denote the previous positions of x and y before executing the pointer statements.

5.3. Numerical Measures on Dynamic Data Structures

We now propose a notion of *shape norms* based on regular expressions that encode sets of selector paths between some memory locations. Intuitively, we assume that the program needs to traverse these paths and hence their length determines (or at least significantly contributes to) the complexity of the algorithm. Typically, one considers selector paths between two memory locations pointed by some pointer variables (e.g. when we are looking for some element in some part of some data structure) or between a location pointed by a variable and `NULL` (e.g. when we are traversing the structure from its beginning to the end). However, one can also use paths between a source location pointed by some variable and any location containing some specific data value.

For a concrete memory configuration, the numerical value of a shape norm corresponds to the *supremum* of the lengths of the paths represented by the regular expression of the norm. Indeed, in the worst case, the program may follow the longest (possibly cyclic and hence infinite) path in the memory. However, note that we will propose a resource bounds analysis that does usually not work with concrete values of shape norms but instead it will work with ASRs and hence we will need to reason about the values of a given norm over potentially infinite sets of shapes. So instead, we will track relative changes (i.e., increments, decrements) of the norms in a way consistent with all the shapes in a given ASR. An exception to this is the case where the value of a norm is equal to a constant for all shapes in the ASR (e.g. after the statement $y = x \rightarrow next$, the distance from x to y via *next* is always 1).

When analysing a program, we will first infer an initial set of *candidate norms* $\mathcal{N}_c$ (i.e., norms potentially useful for establishing resource bounds of the given program) from the CFG of the program—this set may later be extended if we realize some more norms may be useful. Subsequently, we derive as precisely as possible the effects (i.e., increments, decrements, or resets) that particular program statements have on the values of the candidate norms in shapes represented by the different ASRs obtained from shape analysis. We instead use the set of tracked norms $\mathcal{N}_c$ to generate a *integer program* simulating the original program, which allows us to leverage the strength of current resource bounds analysers for integer programs as well as to deal with resource bounds arguments combining heap and numerical measures.

5.3.1. Shape Norms

Let $\mathbb{S}_p$ be the set of selectors. In the rest of the thesis, we will use the set $\mathrm{RE}_{\mathbb{S}}$ of restricted regular expressions `re` over $\mathbb{S}_p$ defined as follows:

$$\texttt{re} ::= \texttt{ru}^* \quad \texttt{ru} ::= sel \mid \texttt{ru} + \texttt{ru} \mid \varepsilon \quad sel \in \mathbb{S}.$$

Below, the `ru` sub-expressions are called *regular units*, sometimes distinguishing *selector units* (*sel*) and *join units* ($\texttt{ru} + \texttt{ru}$). For $re \in \mathrm{RE}_{\mathbb{S}}$, we denote the language of selector paths described by re as $\mathcal{L}_{re}$. Intuitively, when we analyse the control-flow graph of a program for traversals through selectors, a join unit corresponds to a branching of the control-flow, and the star expression ($\texttt{re}^*$) to a whole loop.

Our notion of selector regular expressions can be extended with *concatenation units* ($\texttt{ru.ru}$) and *nested star units* ($\texttt{ru}^*$), corresponding to sequences of unit traversals and nested loop traversals, respectively. Concatenation units are supported in our resource bounds analyser. However, since they bring in many (quite technical) corner cases, in this thesis we limit ourselves to the join units to simplify the presentation. On the other hand, extending the techniques below by nested stars seems to be more complicated, and we leave it for future work. Nevertheless, note that we did not find it much useful in our experiments as it would correspond to using the same variable as the iterator of several nested loops (while usually different pointer variables are used as the iterators of the loops).

Let $\mathbb{V}_p$ be a set of pointer variables and $\mathbb{S}_i$ a set of data selectors. We use $\mathcal{P} = \mathbb{V}_p \cup \{\texttt{NULL}\} \cup \{[.data = k] \mid k \in \mathbb{Z}, data \in \mathbb{S}_i\}$ to refer to locations of memory configurations (shapes) of a program. While $x \in \mathbb{V}_p$ denotes the location that is pointed by the pointer variable x, and NULL denotes the special null location, $[.data = k]$ denotes any memory location whose selector *data* has the value $k \in \mathbb{Z}$. We note that these three cases are the most frequently used in the loop header conditions. A numerical measure μ on a memory configuration, i.e., a *shape norm*, is a triple $(x, \texttt{re}, y) \in \mathbb{V}_p \times \mathrm{RE}_{\mathbb{S}} \times \mathcal{P}$. We will use $\mathcal{N}$ to denote the set of all shape norms, and, further, we will use $x\langle\texttt{ru}^*\rangle y$ as a shorthand for the triple $(x, \texttt{ru}^*, y) \in \mathcal{N}$.

As we have already mentioned above, we are interested in evaluating norms over ASRs, not over concrete shapes. Moreover, up to the cases where a norm has the same constant value for all shapes in an ASR, we are not interested in absolute values of the norms at all, and we instead track changes of the values of the norms only. However, in order to be able to soundly speak about such changes, we need to first define the value of a norm for a shape.

We will define the value of norms in terms of graphs. For this, we first define the notion of the height of a pointed graph. Then we describe how to obtain a pointed graph for a pair of a shape graph and a norm.

Pointed graphs. A *pointed graph* $\mathcal{G} = (N, E, n)$ consists of a set of nodes N, a set of directed edges $E \subseteq N \times N$ and, a source node $n \in N$. A path π is a finite sequence of nodes $n_0, \cdots, n_l$ such that $(n_i, n_{i+1}) \in E$ for all $0 \leq i < l$. We call $|\pi| = l$ the *length* of the path. We say π starts in n if $n_0 = n$. We define the *height* of $\mathcal{G}$ by setting $|\mathcal{G}| = \sup\{|\pi| \mid \text{path } \pi \text{ starts in } n\}$ where we set $\sup(D) = \infty$ for an infinite set $D \subseteq \mathbb{N}$. We note that, for a finite graph $\mathcal{G} = (N, E, n)$, we have $|\mathcal{G}| = \infty$ iff there is a cycle reachable from n.

Pointed graphs associated to shape graphs and null-terminated norms. We first consider norms μ that end in NULL, i.e, $\mu = x\langle\texttt{ru}^*\rangle\texttt{NULL}$. For a shape $s = (M, \sigma, \nu) \in \mathcal{S}$, we define the associated pointed graph $\mathcal{G}_s^{x\langle\texttt{ru}^*\rangle\texttt{NULL}} = (M \cup \{\texttt{NULL}\}, E, \nu(x))$ where $E = \{(n_1, n_2) \in M \times (M \cup \{\texttt{NULL}\}) \mid \text{there is path from } n_1 \text{ to } n_2 \text{ in } s \text{ s.t. the string of selectors along the path matches the regular expression } \texttt{ru}\}$.

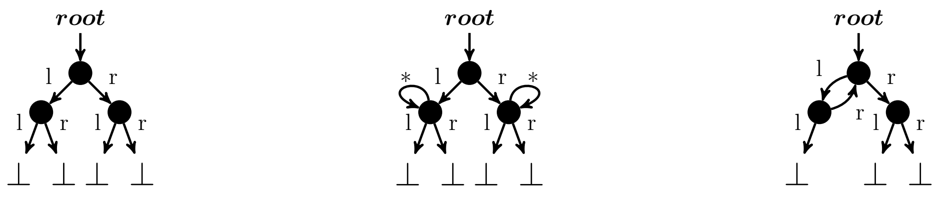
Pointed graphs associated to shape graphs and non-null-terminated norms. We now consider a norm $\mu = x\langle \mathtt{ru}^* \rangle y$ with $y \in \mathcal{P}\setminus\{\mathtt{NULL}\}$. For a shape $s = (M, \sigma, \nu) \in \mathcal{S}$, we set $s(y) = \{\nu(y)\}$ for $y \in \mathbb{V}_p$, and $s(y) = \{m \in M \mid \sigma(m, data) = k\}$ for $y = [.data = k]$. We define the shape $s[y/\mathtt{NULL}] = (M \setminus s(y), \sigma[y/\mathtt{NULL}], \nu[y/\mathtt{NULL}])$ where (1) $\sigma[y/\mathtt{NULL}](m, sel) = \sigma(m, sel)$ if $m \notin s(y)$ and $\sigma[y/\mathtt{NULL}](m, sel) = \mathtt{NULL}$ otherwise, and (2) $\nu[y/\mathtt{NULL}](x) = \nu(x)$ if $\nu(x) \notin s(y)$ and $\nu[y/\mathtt{NULL}](x) = \mathtt{NULL}$ otherwise.

We define the associated pointed graph as $\mathcal{G}_s^{x\langle \mathtt{ru}^* \rangle y} = \mathcal{G}_{s[y/\mathtt{NULL}]}^{x\langle \mathtt{ru}^* \rangle \mathtt{NULL}}$.

Values of shape norms. We can now define values of shape norms in shapes. In particular, the value of a norm $\mu \in \mathcal{N}$ in a shape $s \in \mathcal{S}$, denoted $\|\mu\|_s$, is a value from the set $\mathbb{N} \cup \{\infty\}$ defined such that $\|\mu\|_s = |\mathcal{G}_s^{\mu}|$. This is, the value of the norm μ in the shape s is defined as the height of the associated pointed graph.

The intuition behind the above definition is the following. The pointed graph associated to a norm $\mu = x\langle \mathtt{ru}^* \rangle y$ makes the instances of the regular expression `ru` explicit. The height of the pointed graph corresponds to the longest chain of instances of the expression `ru` in the given shape graph. The intuition behind replacing the targets of norms with `NULL` stems from the fact that one either reaches the replaced target (and hence program will terminate naturally) or reaches the `NULL`, dereferences it and thus crashes (hence terminating unnaturally). However, if we exploit the results of a preceding shape analysis, we can assume memory safety and exclude the case of termination by crash. In case there exists a cycle in the shape reachable from the source point x, the value of the norm is infinite. In such a case the norm is unusable for the later complexity analysis, and hints at the potential non-termination of the program under analysis.

We further generalize the notion of values of norms from particular shapes to ASRs. The value of a norm $\mu \in \mathcal{N}$ over a set of shapes given by an ASR $A \in \mathcal{A}$, denoted $\|\mu\|_A$, is a value from the set $\mathbb{N} \cup \{\infty, \omega\}$ defined such that $\|\mu\|_A = \sup_\omega\{\|\mu\|_s \mid s \in [\![A]\!]\}$ where (i) $\sup_\omega X = \omega$ iff $\infty \in X$ and (ii) $\sup_\omega(X) = \sup X$ otherwise. Intuitively, we need to distinguish the case when some of the represented shapes contains a cyclic selector path and the case where the ASR represents a set of shapes containing paths of finite but unbounded length (e.g. in the case when the ASR represents all acyclic lists of any length). Indeed, in the former case, the program may loop over the cyclic selector path while, in the latter case, it will terminate, but its running time cannot be precisely bounded by a constant (it is bounded, e.g. by the length of the encountered list). We illustrate all of the cases in Figure 5.5



(a) $root\langle \mathrm{l}+\mathrm{r}^* \rangle\mathtt{NULL} = 2$ (b) $root\langle \mathrm{l}+\mathrm{r}^* \rangle\mathtt{NULL} = \infty$ (c) $root\langle \mathrm{l}+\mathrm{r}^* \rangle\mathtt{NULL} = \omega$

Figure 5.5.: Illustration of the three cases of norms values: constant; finite, but unbounded; and infinite. We denote the shape abstraction relation using the $*$ symbol

5.3.2. Deriving the Set of Candidate Shape Norms

We propose a method that be used to infer a suitable initial set of norm candidates for a given heap-manipulating program. Note that this set is only an initial set of norm candidates that could be useful for inferring the bounds on the program loops. It is extended when tracking norm changes as discussed in Section 5.4. For each program loop L, we derive a set of norm candidates in the following three steps:

Inspection of Loop Conditions. We inspect conditions of the loop L which involve pointer variables w.r.t the program model of Section 5.2.1 and declare each variable that appears in such a condition as *relevant.* E.g., for the Algorithm 5.1 variables p and $pstop$ are declared as relevant due to the condition $p \neq pstop$ in line *line 13*.

Analysis of Control-flow We iterate over all simple loop paths of L (a loop path is any path which stays inside the loop L, and starts from and returns to the loop header; a loop path is simple if it does not visit any location twice except for the loop header) and derive a set of selectors $S_x \subseteq \mathbb{S}_p$ for each relevant variable x: Given a simple loop path *slp* and a relevant variable x, we perform a symbolic backward execution to compute the effect of *slp* on x, i.e., we derive an assignment $x = exp$ such that exp captures how x is changed when executing *slp*. For example, for our running example Algorithm 5.1 we infer $x = x \to next$, $y = y$ for both simple loop paths of the outer loop and $y = y \to next$, $x = x$ for the single simple loop path of the inner loop. In the case exp is of form $x \to sel$, i.e., the effect of the loop path is dereferencing variable x by some selector $sel \in \mathbb{S}_p$, we add sel to S_x. This basic approach can be easily extended to handle consecutive dereferences of the same pointer over different selectors: We can deal with expressions of the form $exp = x \to sel_1 \to sel_2$ by adding $sel_1.sel_2$ to S_x.

Inferring Candidates Finally, we consider all subsets $T \subseteq S_x$ and create norms for each $T = \{sel_1, ..., sel_l\}$ using the regular expression $join(T) = sel_1 + \ldots + sel_l$. The candidate norms $\mathcal{N}_L$ created for different forms of conditions of the loop L are given in the right column of Table 5.1. For example, for our running example in Sect. 5.1: we create norm $p\langle \text{next}^* \rangle \texttt{NULL}$ for the outer cycle; norm $q\langle \text{next}^* \rangle \texttt{NULL}$ for first inner cycle; and norms $p\langle \text{next}^* \rangle pstop$, $pstop\langle \text{next}^* \rangle p$, and $q\langle \text{next}^* \rangle \texttt{NULL}$ for the second inner loop.

Condition of L	**Candidate Norms** $\mathcal{N}_L$
$x \circ y$	$\{\ x\langle join(\mathrm{T})^* \rangle y \mid T \subseteq S_x\}$ $\cup\{\ y\langle join(\mathrm{T})^* \rangle x \mid T \subseteq S_x\}$
$x \circ \texttt{NULL}$	$\{\ x\langle join(\mathrm{T})^* \rangle \texttt{NULL} \mid T \subseteq S_x\}$
$x \to d \circ k$	$\{\ x\langle join(\mathrm{T})^* \rangle [.data = k] \mid T \subseteq S_x\}$

Table 5.1.: Norm candidates $\mathcal{N}_L$ for a loop L, $\circ \in \{=, \neq\}$

The overall set of tracked norm candidates $\mathcal{N}_c$ is then set to the union of norm candidates over all loops in the program, i.e. $\mathcal{N}_c = \bigcup_{L \in \mathcal{L}} \mathcal{N}_L$. For each norm from $\mathcal{N}_c$ we will track its size-changes, as we discuss in Section 5.4 in order to derive corresponding integer program for a given heap-manipulating one. Note that we can optimize the size of $\mathcal{N}_c$ by pruning irrelevant norms from the start, e.g. those norms that never decrease; the concrete heuristics are described in Section 5.5.3.

5.4. From Shapes to Norm Changes

In Section 5.3.2, we have shown how to automatically derive an initial set of candidate norms $\mathcal{N}_c$ that are likely to be useful for deriving bounds on the number of executions of the different program loops. This section describes the core of our procedure: how to derive numerical changes of the values of these norms, allowing us to consequently derive an integer program simulating the original heap-manipulating program from the point of view of its runtime complexity. During this process, new norms may be found as potentially useful, which leads to an extension of the set $\mathcal{N}_c$ and to a re-generation of the integer program such that the newly added norms are also tracked.

In the integer program, we introduce a new *numeric variable* for each candidate norm. By a slight abuse of the notation, we use the norms themselves to denote the corresponding numeric variables, so, e.g. we will write $x\langle \mathrm{u}^* \rangle \texttt{NULL} == 0$ to denote that the value of the numeric variable representing the norm $x\langle \mathrm{u}^* \rangle \texttt{NULL}$ is equal to zero. These variables store values from the set $\mathbb{N} \cup \{\omega\}$ with omega representing an infinite distance (due to a loop in a shape). In what follows, we assume that any increment or decrement of ω yields ω again and that ω is larger than any natural number. Note that we do not need a special value to represent ∞ for describing a finite distance without an explicit bound. For that, we will simply introduce a fresh variable constrained to be smaller than ω.

The *integer program* is constructed using the ACFG $\Gamma = (G, \lambda, \rho)$ built on the top of the CFG $G = (\textsc{Loc}, T, l_b, l_e)$ of the original program. The original control flow is preserved except that each location $l \in \textsc{Loc}$ is replaced by a separate copy for each ASR labelling it, i.e., it is replaced by locations (l, A) for each $A \in \lambda(l)$. Transitions between the new locations are obtained by copying the original transitions between those pairs of locations and ASRs that are related by the successor relation, i.e., a transition (l_1, st, l_2) is lifted to $((l_1, A_1), st, (l_2, A_2))$ whenever $((l_1, A_1), (l_2, A_2)) \in \rho$. Subsequently, each pointer-dependent condition labelling some edge in the extended CFG is translated to a condition on the numeric variables corresponding to the shape norms from $\mathcal{N}_c$. Likewise, each edge originally labeled by a pointer-manipulating statement is relabeled by numerical updates of the values of the concerned norm variables. Integer conditions and statements are left untouched.

Soundness of the abstraction. The translation of the pointer statements described below is done such that, for any path π in the CFG of a program and the shape s resulting from executing π, the values of the numeric norm variables obtained by executing the corresponding path in the integer program conservatively *over-approximate* the values of the norms over s. This is, if the numeric variable corresponding to some norm μ can reach a value $n \in \mathbb{N} \cup \{\omega\}$ through the path π with pointer statements replaced as described below, then $\|\mu\|_s \leq n$. *As a consequence, we get that every bound obtained for the integer abstraction is a bound of the original program.*

Translating pointer conditions. Given the above, the translation of *pointer conditions* is straightforward. We translate each condition `x == NULL` to a disjunction of tests $x\langle \mathrm{u}^*\rangle\mathtt{NULL} == 0$ over all regular units u such that $x\langle \mathrm{u}^*\rangle\mathtt{NULL} \in \mathcal{N}_c$. Likewise, every condition `x == y` is translated to a disjunction of conditions of the form $x\langle \mathrm{u}^*\rangle y == 0$ and $y\langle \mathrm{u}^*\rangle x == 0$ over all regular units u such that $x\langle \mathrm{u}^*\rangle y, y\langle \mathrm{u}^*\rangle x \in \mathcal{N}_c$. The intuition is that the condition holds when at least one of the norms is equal to zero and so the source variable x has reached the target `NULL` or y in the shape. Pointer inequalities are then translated to a negation of the conditions formed as above, leading to a conjunction of inequalities on numeric norm variables.

Handling *data-related pointer tests* of the form $\mathtt{x} \to \mathtt{data} == \mathtt{y}$ is more complex. Consider such a test on an edge starting from a location-ASR pair (l, A). Currently, we can handle the test in a non-trivial way only if y evaluates to the same constant value in all shapes represented by A, i.e., if there is some $k \in \mathbb{N}$ such that $\nu(y) = k$ for all shapes $(M, \sigma, \nu) \in [\![A]\!]$. In this case, the test is translated to a disjunction of conditions of the form $x\langle \mathrm{u}^*\rangle[.data = k] == 0$ over all regular units u such that $x\langle \mathrm{u}^*\rangle[.data = k] \in \mathcal{N}_c$. Otherwise, the test is left out—a better solution is an interesting issue for the future work, possibly requiring more advanced shape analysis and a tighter integration with it. Data-related pointer non-equalities can then again be treated by negation of the equality test (provided y evaluates to a constant value).

Finally, after a successful equality test (of any of the above kinds), all numeric norm variables that appeared in the disjunctive condition used are set to zero. All other variables (and all variables in general for an inequality test) keep their original value.

Translating pointer statements. Next, we show how we translate non-destructive, destructive, and data-related pointer statements other than tests. The translation can lead to decrements, resets, or increments of the numeric norm variables corresponding to the norms in $\mathcal{N}_c$. In case, we realize that we need some norm $\mu' \notin \mathcal{N}_c$ to describe the value of some current candidate norm $\mu \in \mathcal{N}_c$, we add μ' into $\mathcal{N}_c$ and restart the translation process (in practice, of course, the results of the previously performed translation steps can be reused). Such a situation can happen, e.g. when $\mathcal{N}_c = \{x\langle \mathrm{next}^*\rangle\mathtt{NULL}\}$ and we encounter an instruction `x = list`, which generates a reset of the norm $x\langle \mathrm{next}^*\rangle\mathtt{NULL}$ to the value of the norm $list\langle \mathrm{next}^*\rangle\mathtt{NULL}$. The latter norm is then added into $\mathcal{N}_c$.[3]

The rules for translating non-destructive, destructive, and data-related pointer updates to the corresponding updates on numeric variables are given in Figures 5.6, 5.9, and 5.12, respectively. Before commenting on them in more detail, we first make several general notes. First, values of norms of the form $x\langle \mathrm{u}^*\rangle x$ are always zero, and hence we do not consider them in the rules. Next, let $u = sel_1 + \ldots + sel_n$, $n \geq 1$, be a regular join unit. We will write $sel \in u$ iff $sel = sel_i$ for some $1 \leq i \leq n$. Rules that use some selector unit n (e.g. $x = y \to n$) have two distinct cases on norms of form $x\langle \mathrm{u}^*\rangle y$: (1) when $n \in u$ (the so-called unit case) and (2) when $n \notin u$ (the so-called non-unit case). We denote new values of norms using an overline, and the old values without an overline. The norms that are not mentioned in a given rule keep implicitly the same value.

[3]Alternatively, one could use a more complex initial static analysis that would cover, although maybe less precisely, even such dependencies among norms.

Finally, in rules describing how the value of a norm variable μ is changed by firing some statement between ASRs A_1 and A_2, we often use constructions of the form $\overline{\mu} \mathrel{\mathring{=}} expr$ where *expr* is an expression on norm variables. This construction constrains the new value of μ using the current values of norm variables or using directly the ASRs encountered, depending on which case is more precise. First, if μ has the same natural value in all shapes in $[\![A_2]\!]$, i.e., if $\|\mu\|_{A_2} \in \mathbb{N}$, then we let $\overline{\mu} = \|\mu\|_{A_2}$. Otherwise, if the value of μ is infinite in A_1 and unbounded but finite in A_2, i.e., if $\|\mu\|_{A_1} = \omega$ and $\|\mu\|_{A_2} = \infty$, we constrain the new value of μ by the constraint $\overline{\mu} = v \ \wedge \ v < \infty$ where v is a fresh numeric variable.[4] The same constraint with a fresh variable is used when $\|\mu\|_{A_2} = \infty$ and $expr = \omega$. Otherwise, we let $\overline{\mu} = expr$. At last, we assume that the preceding shape analysis will discover potential problems with a location being freed and re-allocated with some dangling pointers still pointing to it (the ABA problem).

The described translation allows for sound resource bounds analysis. Indeed, for each run of the original pointer program, there will exist one run in the derived integer program where the norms get exact or overapproximated values. Provided that the underlying bounds analyser is sound in that it returns worst-case bounds, the bounds obtained for the integer program will not be smaller than the bounds of the original program.

[4]Intuitively, this case is used, e.g. when $\mu = x\langle \mathrm{n}^* \rangle \texttt{NULL}$, and the encountered pointer statement cuts an ASR representing cyclic lists of any length pointed by x to an ASR representing acyclic `NULL`-terminated lists pointed by x. Naturally, when one subsequently starts a traversal of the list, it will terminate though in an unknown number of steps.

5.4.1. Non-Destructive Pointer Updates

We now comment more on the less obvious parts of the rules from Figure 5.6. Concerning the rule for $x = \texttt{NULL}$, Case 1 reflects the fact that we always consider all paths from x limited by either the designated target w or, implicitly, NULL. Hence, after $x = \texttt{NULL}$, the distance is always 0. Likewise, in Case 1 of $x = \mathit{malloc}()$, the distance is always 1 as we assume all fields of the newly allocated cell to be nullified, and so the paths consist of the newly allocated cell only. Case 2 of $x = \mathit{malloc}()$ is based on that we assume the newly allocated cell to be unreachable from other memory locations, and so any path taken from another memory location towards x will implicitly be bounded by NULL.

$$\frac{\texttt{[x = NULL]} \quad \forall w \in \mathcal{P},\ \forall z \in \mathbb{V}_p \setminus \{x\}}{\begin{array}{lcll}\overline{x\langle \text{u}^*\rangle w} & = & 0 & (1)\\ \overline{z\langle \text{u}^*\rangle x} & \mathrel{\hat{=}} & z\langle \text{u}^*\rangle\texttt{NULL} & (2)\end{array}}
\qquad
\frac{\texttt{[x = malloc()]} \quad \forall w \in \mathcal{P}\setminus\{x\},\ \forall z \in \mathbb{V}_p}{\begin{array}{lcll}\overline{x\langle \text{u}^*\rangle w} & = & 1 & (1)\\ \overline{z\langle \text{u}^*\rangle x} & \mathrel{\hat{=}} & z\langle \text{u}^*\rangle\texttt{NULL} & (2)\end{array}}$$

$$\frac{\begin{array}{c}\texttt{[x = y}\rightarrow\texttt{n (alias)]}\\ \exists v \in \texttt{AliasNext}(A_1, y, \texttt{n})\\ \forall w \in \mathcal{P}\ \forall z \in \mathbb{V}_p\end{array}}{\begin{array}{lcll}\overline{x\langle \text{u}^*\rangle w} & \mathrel{\hat{=}} & v\langle \text{u}^*\rangle w & (1)\\ \overline{z\langle \text{u}^*\rangle x} & \mathrel{\hat{=}} & z\langle \text{u}^*\rangle v & (2)\end{array}}
\qquad
\frac{\begin{array}{c}\texttt{[x = y}\rightarrow\texttt{n (non-unit)]}\\ \texttt{n} \notin u,\ \forall w \in \mathcal{P},\ \forall z \in \mathbb{V}_p\end{array}}{\begin{array}{lcll}\overline{x\langle \text{u}^*\rangle w} & \mathrel{\hat{=}} & \omega & (1)\\ \overline{z\langle \text{u}^*\rangle x} & \mathrel{\hat{=}} & \omega & (2)\end{array}}$$

$$\frac{\begin{array}{c}\texttt{[x = y]}\\ \forall z \in \mathbb{V}_p\ \forall w \in \mathcal{P}\end{array}}{\overline{x\langle \text{u}^*\rangle w} \mathrel{\hat{=}} y\langle \text{u}^*\rangle w \quad (1) \qquad \overline{z\langle \text{u}^*\rangle x} \mathrel{\hat{=}} z\langle \text{u}^*\rangle y \quad (2)}$$

$$\frac{\begin{array}{c}\texttt{[free(x)]}\\ \forall z \in \mathbb{V}_p,\ \forall w \in \mathcal{P}\end{array}}{\overline{z\langle \text{u}^*\rangle w} \mathrel{\hat{=}} \begin{cases} z\langle \text{u}^*\rangle x & \texttt{AllPaths}(A_1, u, z, w, x))\\ z\langle \text{u}^*\rangle w & \text{otherwise}\end{cases}}$$

$$\frac{\begin{array}{c}\texttt{[x = y}\rightarrow\texttt{n (unit)]}\\ \texttt{n} \in u,\ x \neq y,\ \forall t \in \texttt{Alias}(A_1, y)\\ \forall s \in \texttt{MayAlias}(A_1, y)\\ \forall w \in \mathcal{P} \setminus \texttt{MayAlias}(A_1, y)\\ \forall z \in \mathbb{V}_p \setminus \texttt{Alias}(A_1, y)\end{array}}{\begin{array}{lcll}\overline{t\langle \text{u}^*\rangle x} & \mathrel{\hat{=}} & t\langle \text{u}^*\rangle\texttt{NULL} & (1)\\ \overline{x\langle \text{u}^*\rangle s} & \mathrel{\hat{=}} & s\langle \text{u}^*\rangle\texttt{NULL} - 1 & (2)\\ \overline{x\langle \text{u}^*\rangle w} & \mathrel{\hat{=}} & y\langle \text{u}^*\rangle w - 1 & (3)\\ \overline{z\langle \text{u}^*\rangle x} & \mathrel{\hat{=}} & z\langle \text{u}^*\rangle y + \overline{y\langle \text{u}^*\rangle x} & (4)\end{array}}$$

Figure 5.6.: Translation rules for *non-destructive pointer updates*. The rules are assumed to be applied between location-ASR pairs (l_1, A_1) and (l_2, A_2) linked by an edge labelled by a non-destructive pointer update with $x, y \in \mathbb{V}_p$, $n \in \mathbb{S}_p$. For any case of any of the rules with the left-hand side of the form $\overline{a\langle \text{u}^*\rangle b}$, u ranges over all regular units such that $a\langle \text{u}^*\rangle b \in \mathcal{N}_c$. If the norms used on the right-hand side of any of the applied rules is not in $\mathcal{N}_c$, it is added into $\mathcal{N}_c$, and the analysis is re-run with the new $\mathcal{N}_c$.

Free. Concerning the rules for $free(x)$, the predicate $\texttt{AllPaths}(A, u, z, w, x)$ holds iff all paths over selector sequences matching u^* between the location z and the location w go through x in all shapes in $[\![A]\!]$. In this case, clearly, all paths from z to w are shrunk to paths to x by $free(x)$ as x becomes undefined (which we take as equal to `NULL` for our purposes[5]; we discussed the validity of this approach before.) Otherwise, we take the old value of the norm since it either stays the same or the paths perhaps get shorter but in some shapes only. However, using this reasoning will still get precise upper bounds on the norms values. We illustrate this rule in Figure 5.7.

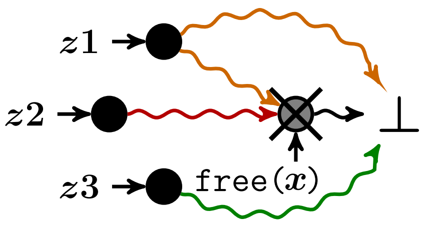


Figure 5.7.: Illustration of the `[free(x)]` rule. By freeing the variable x we can only shorten the norm $z_2\langle \mathrm{u}^*\rangle$`NULL`, since all of the red paths go through the variable x. On the other hand, we keep the values of norms $z_1\langle \mathrm{u}^*\rangle$`NULL` and $z_3\langle \mathrm{u}^*\rangle$`NULL`, because the green and orange paths may have been longer than those going through the x. Hence, in order to infer safe bounds, we keep the previous values.

Non-destructive dereference. Concerning the rules for $x = y \to n$, we first note that, if applicable, the "alias" rule has priority. It is applied when the n-successor of y is pointed by some variable v in all shapes in $[\![A_1]\!]$. Formally, $v \in \texttt{AliasNext}(A, y, \texttt{n})$ iff $\forall(M, \sigma, \nu) \in [\![A]\!] : \sigma(\nu(y), n) = \nu(v)$. Such an alias can be used to define norms based on x by copying those based on v. Of course, the distance from x to v after the update should be zero, which is assured by the $\overset{\circ}{=}$ operator. If there is no such v, and n does not match u, we can only limit the new value of the norm based on the ASR, which is again taken care by the $\overset{\circ}{=}$ operator (otherwise we take the worst possibility, i.e., ω).

The most complex rule is that for $x = y \to n$ when there is no alias for the n-successor of y, and the selector n matches u. First, note that the rule is provided for the case of x being a different variable than y only. We assume statements $x = x \to n$ to be transformed to a sequence $y = x; x = y \to n$; for a fresh pointer variable y. In the rules, we then use the following must- and may-alias sets: $\texttt{Alias}(A, y) = \{v \in \mathbb{V}_p \mid \forall(M, \sigma, \nu) \in [\![A]\!] : \nu(y) = \nu(v)\}$ and $\texttt{MayAlias}(A, y) = \{v \in \mathbb{V}_p \mid \exists(M, \sigma, \nu) \in [\![A]\!] : \nu(y) = \nu(v)\}$.

Concerning Case 1, note that u can be a join unit and u^* can match several paths from y that need not go to the new position of x at all (and hence can stop only when reaching `NULL`), or they can go there, but as there is no variable pointing already to the new position of x, we anyway have to approximate such paths by extending them up to `NULL`. The case when the only path to x is via n will then be solved by the $\overset{\circ}{=}$ operator. The must aliases of y can naturally be treated in an equal way as y in the above.

[5]Note that all outgoing norms from the variable x will be set according to the shape, which is ensured using the $\overset{\circ}{=}$ operator.

In Case 2, we start by considering paths from x to y. Since we have no alias of the n-successor of y that could help us define the value of the norm, we have to approximate the distance from x to y by extending the paths from x up until `NULL`. Further note that such paths are a subset of those from y to `NULL` (since the new position of x is a successor of y). We can thus use $y\langle \text{u}^* \rangle$`NULL` to approximate $\overline{x\langle \text{u}^* \rangle \texttt{NULL}}$[6]. However, we can constrain the latter distance to be smaller by one. Indeed, if the longest path from y to `NULL` does not go through the new position of x, the distance from x to `NULL` is at least by one smaller. On the other hand, if the longest path goes through the new position of x, then we save the step from y to the new position of x. The same reasoning then applies for any variable that may alias y—for those that cannot alias it, one can do better as expressed in the next case.

In Case 3, one can use a similar reasoning as in Case 2 as the paths from y to w include those from the new position of x to w. Note, however, that this reasoning cannot be applied when y may alias with w. In such a case, their distance may be zero, and the distance from the new position of x to w can be bigger, not smaller.[7] Finally, to see correctness of Case 4, note that should there be a longer path over u from z to the n-successor of y than going through y, this longer path will be included into the value of the norm for getting from z to y too since the norm takes into account all u paths either going to y or missing it and then going up until `NULL` (or looping). We illustrate the four cases of this rule in Figure 5.8.

The intuition behind the rule for $x = y$ is similar to the other statements.

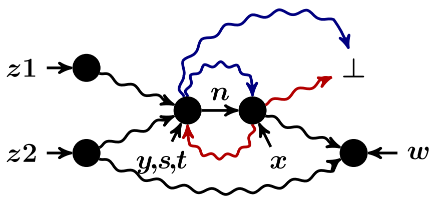


Figure 5.8.: Illustration of the `[x = y→n (unit)]` rule. Blue paths from the aliased variables t illustrate the case (1), when there may exist other paths that do not lead to the variable x (e.g. when $n = \mathit{right}$, $u = \mathit{left} + \mathit{right}$, and there exists a path from t to `NULL` through $t \to \mathit{right}$). Red paths illustrate the case (2), where $s\langle \text{u}^* \rangle$`NULL` $- 1$ is the best approximation of the norm $x\langle \text{u}^* \rangle s$. The rest of the paths illustrate the cases (3) and (4). Note that norms $z_1\langle \text{u}^* \rangle w$ and $z_2\langle \text{u}^* \rangle w$ are not change at all by this rule.

[6]Note that $y\langle \text{u}^* \rangle$`NULL` can naturally be equal to ω, in case there is a loop in the shape, i.e. one can get back to y from its successors.

[7]Note that while the use of `Alias` is, in fact, an optimization, which could be removed, the use of `MayAlias` is necessary.

5.4.2. Destructive Pointer Updates

We now proceed to the rules for destructive pointer statements shown in Figure 5.9. Contrary to previous set of rules, destructive updates can change and increase the length of the selector paths.

$$\frac{\begin{array}{c}\texttt{[x}\to\texttt{n = NULL (unit)]}\\ \texttt{n} \in u,\ \forall z \in \mathbb{V}_p,\ \forall w \in \mathcal{P}\end{array}}{\overline{z\langle \mathrm{u}^*\rangle w} \mathrel{\mathring{=}} \begin{cases} z\langle \mathrm{u}^*\rangle x + 1 & \texttt{AllPathsFld}(A_1, u, z, w, x, n) \\ z\langle \mathrm{u}^*\rangle w & \text{otherwise}\end{cases}}$$

$$\frac{\begin{array}{c}\texttt{[x}\to\texttt{n = y (unit)]}\\ \texttt{n} \in u,\ \forall s_1 \in \texttt{Alias}(A_1, x),\ \forall s_2 \in \texttt{MayAlias}(A_1, x)\\ \forall t_1 \in \texttt{Alias}(A_1, y),\ \forall t_2 \in \texttt{MayAlias}(A_1, y)\\ \forall w \in \mathcal{P} \setminus (\texttt{Alias}(A_1, x) \cup \texttt{Alias}(A_1, y))\\ \forall z \in \mathbb{V}_p \setminus (\texttt{MayAlias}(A_1, x) \cup \texttt{MayAlias}(A_1, y))\end{array}}{\begin{array}{rcll} \overline{s_1\langle \mathrm{u}^*\rangle t_1} & \mathring{=} & s_1\langle \mathrm{u}^*\rangle t_1 & (1)\\ \overline{s_2\langle \mathrm{u}^*\rangle w} & \mathring{=} & \begin{cases} \omega & \texttt{BadLoopClosed}(A_2, u, y, x, w)\\ \overline{s_2\langle \mathrm{u}^*\rangle y} + y\langle \mathrm{u}^*\rangle w & \text{otherwise}\end{cases} & (2)\\ \overline{t_2\langle \mathrm{u}^*\rangle w} & \mathring{=} & \begin{cases} \omega & \texttt{BadLoopClosed}(A_2, u, y, x, w)\\ t_2\langle \mathrm{u}^*\rangle w & \text{otherwise}\end{cases} & (3)\\ \overline{z\langle \mathrm{u}^*\rangle w} & \mathring{=} & \begin{cases} z\langle \mathrm{u}^*\rangle x + \overline{x\langle \mathrm{u}^*\rangle w} & \texttt{AllPaths}(A_2, u, z, w, x))\\ \max\big(z\langle \mathrm{u}^*\rangle x + \overline{x\langle \mathrm{u}^*\rangle w}, z\langle \mathrm{u}^*\rangle w\big) & \texttt{SomePaths}(A_2, u, z, w, x))\\ z\langle \mathrm{u}^*\rangle w & \text{otherwise}\end{cases} & (4)\end{array}}$$

Figure 5.9.: Translation rules for *destructive pointer updates*. The rules are assumed to be applied between location-ASR pairs (l_1, A_1) and (l_2, A_2) linked by an edge labelled by a destructive pointer update with $x, y \in \mathbb{V}_p$, $n \in \mathbb{S}_p$. The treatment of the regular units u is the same as in Fig. 5.6.

Destructive dereference update to `NULL`. We first start with the translation for the statement $x \to n =$ `NULL`, considering the case of n being a unit, i.e., $n \in u$. After this statement, the distance from any source memory location z to any target memory location w either stays the same or decreases. The latter happens when the changed n-selector of x influences the longest previously existing path from z to w. Identifying this case in general is difficult, but one can reasonably recognize it in common ASRs at least in the situation when all paths between z and w whose selector sequences match u^* go through the n-selector of the memory location marked by x in all shapes represented by the ASR A_1, i.e., $[\![A_1]\!]$. We denote this fact by the predicate $\texttt{AllPathsFld}(A_1, u, z, w, x, n)$. In this case, the new distance between z and w clearly corresponds to the old distance between z and x plus one (for the step from x to `NULL`). In all other cases, we conservatively keep the old value of the distance (up to it can be reduced by the $\mathring{=}$ operator as usual). We illustrate this rule in Figure 5.10.

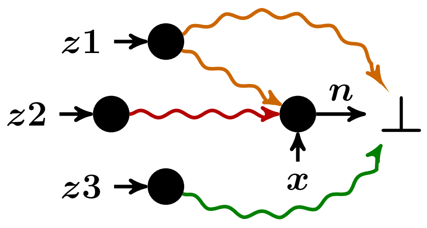


Figure 5.10.: Illustration of the `[x→n = NULL (unit)]` rule. The intuition is similar to the `[free(x)]` rule depicted in Figure 5.7. Only red paths from the z_2 are shortened, hence, we only modify the value of the norm $z_2\langle \mathrm{u}^* \rangle \mathtt{NULL}$.

Destructive dereference update to variable. Concerning the statement $x \to n = y$, the distance between x and y (and their aliases) can stay the same or get shortened. In Case 1 of the rule for this statement, the latter is reflected in the use of the $\mathrel{\mathring{=}}$ operator. In Case 2, we use the predicate $\mathtt{BadLoopClosed}(A_2, u, y, x, w)$ to denote a situation when the statement $x \to n = y$ closes a loop (over the u selectors) in at least some shape represented by A_2 such that w does not appear in between of y and x in the loop. Naturally, in such a case, the distance between x (or any of its may-aliases) and w is set to ω. Note that the may-alias is needed in this case since it is enough that this problematic situation arises even in one of the concerned shapes. As for correctness of the other variant of Case 2, if there are paths over u^* from x to w not passing through y, they will be covered by $\overline{x\langle \mathrm{u}^* \rangle y}$, which will consider such paths extended up until `NULL`. In Case 3, if the loop is not closed, then the paths from y to w are not influenced.

In Case 4, if all paths from z to w in the shapes represented by A_2 go through x, we can take the original distance of z and x, which does not change between A_1 and A_2 as the change happens after x, and then add the new distance from x to w. If no path from z to w passes x, the distance is not influenced by the statement. If some but not all of the paths pass x, we have to take the maximum of the two previous cases. We illustrate the cases in Figure 5.11.

As for non-unit cases of the above two statements, i.e., the case when $n \notin u$, the norms do not change since the paths over u^* do not pass the changed selector.

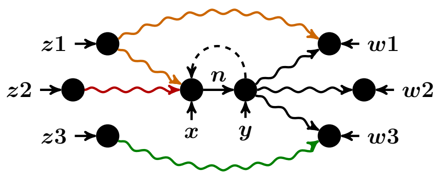


Figure 5.11.: Illustration of the `[x→n = y (unit)]` rule. The figure illustrates mainly the case (4). The new value of the norm $z_1\langle \mathrm{u}^* \rangle w_1$ must contain the maximum constraints, since we cannot guarantee that newly created paths are longer. The value of $z_2\langle \mathrm{u}^* \rangle w_2$, however, can be safely shortened to $z_2\langle \mathrm{u}^* \rangle x + \overline{x\langle \mathrm{u}^* \rangle w_2}$. At last the value of $z_3\langle \mathrm{u}^* \rangle w3$ does not change at all.

5.4.3. Data-Related Pointer Updates

$$\frac{\begin{array}{c}\texttt{[x}\rightarrow\texttt{d = y (data-const)]}\\ \exists k \in \mathbb{Z} : \texttt{ValIsConst}(A_1, y, k),\ \forall z \in \mathbb{V}_p \setminus \{x\},\ \forall l \in \mathbb{Z} \setminus \{k\}\end{array}}{\begin{array}{rcll}
\overline{x\langle\mathrm{u}^*\rangle[.d = k]} & = & 0 & (1)\\
\overline{x\langle\mathrm{u}^*\rangle[.d = l]} & \overset{\circ}{=} & \begin{cases} x\langle\mathrm{u}^*\rangle\texttt{NULL} & \texttt{ValMayBe}(A_1, x, d, l)\\ x\langle\mathrm{u}^*\rangle[.d = l] & \text{otherwise}\end{cases} & (2)\\
\overline{z\langle\mathrm{u}^*\rangle[.d = k]} & \overset{\circ}{=} & \begin{cases} z\langle\mathrm{u}^*\rangle x & \texttt{AllPaths}(A_1, u, z, [.d = k], x)\\ z\langle\mathrm{u}^*\rangle[.d = k] & \text{otherwise}\end{cases} & (3)\\
\overline{z\langle\mathrm{u}^*\rangle[.d = l]} & \overset{\circ}{=} & \begin{cases} z\langle\mathrm{u}^*\rangle\texttt{NULL} & \texttt{ValMayBe}(A_1, x, d, l)\\ z\langle\mathrm{u}^*\rangle[.d = l] & \text{otherwise}\end{cases} & (4)
\end{array}}$$

$$\frac{\begin{array}{c}\texttt{[x}\rightarrow\texttt{d = y (data-unknown)]}\\ (\neg\exists k \in \mathbb{Z} : \texttt{ValIsConst}(A_1, y, k)),\ \forall z \in \mathbb{V}_p,\ \forall l \in \mathbb{Z}\end{array}}{\overline{z\langle\mathrm{u}^*\rangle[.d = l]} \overset{\circ}{=} \begin{cases} z\langle\mathrm{u}^*\rangle[.d = l] & z\langle\mathrm{u}^*\rangle[.d = l] < z\langle\mathrm{u}^*\rangle x\\ z\langle\mathrm{u}^*\rangle[.d = l] & \neg\texttt{ValMayBe}(A_1, x, d, l)\\ z\langle\mathrm{u}^*\rangle\texttt{NULL} & \end{cases}}$$

Figure 5.12.: Translation rules for *data-related pointer updates*. The rules are assumed to be applied between location-ASR pairs (l_1, A_1) and (l_2, A_2) linked by an edge labelled by a data-related pointer update with $x \in \mathbb{V}_p$, $y \in \mathbb{V}_i$, $d \in \mathbb{S}_i$. The treatment of the regular units u is the same as in Fig. 5.6.

Our rules for translating data-related pointer updates are given in Figure 5.12. The first of them applies in case the value being written into the data field d of the memory location pointed by x is constant over all shapes represented by the ASR A_1, i.e., if there is some constant $k \in \mathbb{Z}$ such that $\forall(M, \sigma, \nu) \in [\![A]\!] : \nu(y) = k$. This fact is expressed by the $\texttt{ValIsConst}(A_1, y, k)$ predicate. In this case, after the statement $x \rightarrow d = y$, the distance from x to a data value k becomes clearly zero. Case 2 captures the fact that if the d-field of x may be l in at least one shape represented by A_1, i.e., if $\exists(M, \sigma, \nu) \in [\![A]\!] : \nu(y) = l$, which is expressed by the $\texttt{ValMayBe}(A_1, x, d, l)$ predicate, the new distance of x to a data value l is approximated by its distance to `NULL`. The reason is that the old data value is re-written, and one cannot say whether another data field with the value l may be reached before one gets to `NULL`. Otherwise, the norm keeps its original value. Case 3 covers the distance from a location z other than x to a data value k. This distance clearly stays the same or can get shorter after the statement. We are able to safely detect the second scenario when all paths from z to a data value k lead through x. In that case, the distance from z to a data value k shrinks to that from z to x. Otherwise, we conservatively keep the norm value unchanged. Finally, Case 4 is an analogy of Case 2.

In case the value being written through a data selector is not constant, which is covered by the second rule of Fig. 5.12, our approach is currently rather conservative. We keep the original value of the norms between z and a data value l if either this data value is always reached from z before x is reached (the norm takes into account the first occurrence of the data value) or if the re-written value of the data field d of x is not l in any of the shapes represented by A_1 (and hence the original value of the norm is not based on the distance to this particular field). In such a case, the distance between z and the data value l does surely not change. Otherwise, we conservatively approximate the new distance between z and the data value l by the distance over paths matching u^* from z up until `NULL`.

The stress on handling constant values of data may seem quite restricted, but it may still allow one to verify a lot of interesting programs. The reason is that often the programs use various important constants (like 0) to steer their control flow. Moreover, due to data-independence, it is often enough to let programs work with just a few constant values—c.f., e.g. [BHV04, AHH$^+$13, HHL$^+$15a] where just a few data values ("colors") are used when checking various advanced properties of dynamic data structures. We illustrate the data rule in Figure 5.13. Still a better support of data is an interesting issue for future work.

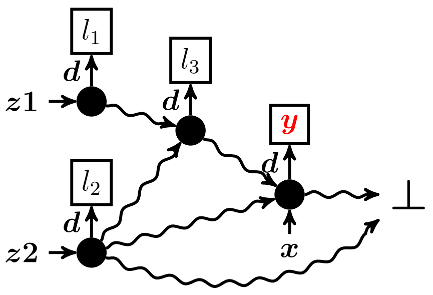


Figure 5.13.: Illustration of the `[x→d = y]` rule. Depending on the value of y, we have to safely set the new values of norms. In case the $x \to d$ was previously equal to l_3, we keep the value of the norm $z_1\langle \mathrm{u}^*\rangle[.d = l_3]$, but reset $z_2\langle \mathrm{u}^*\rangle[.d = l_3]$ to $z_2\langle \mathrm{u}^*\rangle$`NULL`. The similar intuition is applied to other cases.

5.5. Implementation and Experiments

We have implemented our method in a prototype tool called Ranger. The implementation is based on the *Forester* shape analyser [HŠRV13, HHL+15a], which represents sets of memory shapes using so-called *forest automata* (FAs). We use the *Loopus* tool [SZ10] as a back-end *resource bounds analyser* for the generated integer programs. Moreover, we use Z3 [Bjo08] as an underlying SMT solver for inferring concrete constant numerical changes of the norms and for several optimizations. We evaluated Ranger on a set of benchmarks including programs manipulating various complex data structures or requiring amortized reasoning for inferring precise bounds. The experimental results we obtained are quite encouraging and show that we were able to leverage both the precise shape analysis of complex data structure provided by Forester as well as the amortized analysis of loop bounds provided by Loopus and consequently we precisely analyse some challenging programs for the first time fully-automatically.

In the rest of the section, we first briefly introduce our extension of the Forester tool in some more detail and discuss how we implemented our approach on top of it. Next, we mention some further optimizations that we included into the implementation. At last, we present the experiments we performed and their results.

5.5.1. Requirements on Underlying Shape Analysis

In our approach, the underlying shape analysis is used as a parameter. This subsection lists our requirements on what it should provide. We mainly use the inferred shape invariants (i) to check the aliasing of variables, (ii) to compute the value of the norms for a given ASR, and (iii) to generate the annotated control-flow graph for the given program P as defined in Section 5.2.2. Contrary to, e.g. approaches of [MDR16] or [BCC+07], we lay lower requirements on the shape analysis. In particular we require the following:

1. After a successful shape analysis run, we require the underlying shape analyser to generate the annotated CFG (as defined in the Section 5.2.2), i.e. the analyser has to annotate each location with a set of shapes together with the successor relation of the shapes. We need these shapes and their relations (1) in the rules defined in Sections 5.4.1-5.4.3, and (2) in selected optimizations.

2. Given an ASR at a given location in an annotated CFG, the underlying shape analyser has to be able to return the value of any numerical measure (norm), which represents the lengths of paths between two points in the shape (like e.g. variables, `NULL`, etc.) denoted as a regular expression over selectors of dynamic data structures. In particular, the returned value either has to be a numeric constant c, a special symbol ∞ (to represent finite, but unbounded length of paths), or a special symbol ω (to represent that the given ASR contains cyclic selector path). Basically, the shape analyser has to implement the $\mathrel{\mathring{=}}$ operator.

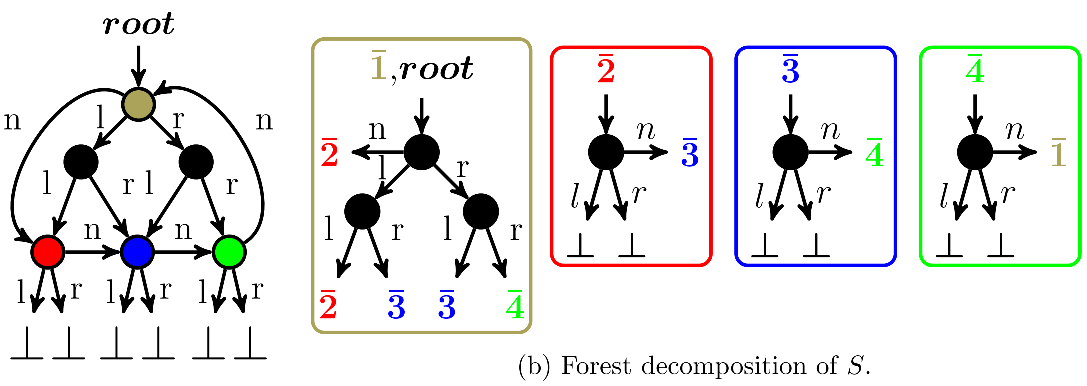
3. Given an ASR at a given location in an annotated CFG, the underlying shape analyser has to be able to traverse selector paths leading to (resp.from) a certain distinct point (`NULL`, variables, etc.). Basically, it has to be able to implement the predicates `AllPaths`, `SomePaths`, `AllPathsFld`, and `BadLoopClosed`.

4. For any given location and an ASR, the underlying shape analyser has to be able to compute the must- and may- aliasing of variables. In particular, it has to be able to compute `Alias` and `MayAlias` sets used in rules in Section 5.4.1 and 5.4.2.

5. For any given location and and ASR, the underlying shape analyser has to be able to return (or at least safely approximate) the values of data fields. In particular, it has to implement `ValMayBe` and `ValIsConst` predicates.

5.5.2. Implementation on Top of Forester

The Forester shape analyser represents particular shapes by decomposing them into *tuples* of *tree components*, and hence *forests*. In particular, each memory location that is `NULL`, pointed by a pointer variable, or that has multiple incoming pointers becomes a so-called *cut-point*. Shape graphs are then cut into tree components at the cut-points, and each cut-point becomes the root of one of the tree components. Leaves of the tree components may then refer back to the roots, which can be used to represent both loops in the shapes as well as multiple paths leading to the same location. Of course, Forester does not work with particular shapes but with sets of shapes. This leads to a need of dealing with tuples of sets of tree components, which are finitely represented using finite *tree automata* (TAs). A tuple of TAs then forms an forest automaton (FA), which we use as the ASR in our implementation. An example of FA is depicted in Figure 5.14.



(a) Concrete tree structure S

(b) Forest decomposition of S.

Figure 5.14.: Subfigure (a) shows a concrete tree structure with $left$ and $right$ successors (or childs) terminated with `NULL`. Moreover, the root node is singly-linked with the leaves of the tree. We highlight the four *cut-points* (i.e. the nodes with either multiple references or variable reference) using colours. Each cut-point is the root of the tree automaton depicted in subfigure (b). We denote references to ith node as $\bar{\imath}$.

Hence, we need to be able to implement all the operations used on ASRs in the previous section on FAs. Fortunately, it turns out that it is not at all difficult.[8] In particular, we can implement the various operations by searching through the particular TAs of an FA, following the transitions that match the relevant unit expressions.[9] We can then, e.g. easily see whether the distance between some memory locations is constant, finite but unbounded, or infinite. It is constant if the given memory locations are linked by paths in the structure of the involved automata that are of the given constant length. It is finite but unbounded if there is a loop in the TA in between of the concerned locations (allowing the TA to accept a sequence of any finite length). Finally, the distance is infinite if some path from the source leads—while not passing through the target location—to some of the roots, which is then in turn referenced back from some leaf node reachable from it. Likewise, one can easily implement checks whether all paths go through some location, whether some variables are aliased (in Forester, this simply corresponds to the variables being associated with the same root), or whether a loop is closed by some destructive update (which must create a reference from a leaf back to a loop).

5.5.3. Optimizations of the Basic Approach

In Ranger, we use several heuristic optimizations to reduce the size of the generated integer program. First, we do not translate each pointer statement in isolation to generate the set of norm changes as described in Section 5.4. Instead, we perform the translation per *basic blocks*. Basically, we take the blocks written in the static single assignment form, translate the statements in the blocks as described in Section 5.4, and then perform various standard simplifications of the generated numeric constraint (in particular, evaluation of constant expressions, copy propagation, elimination of variables) using the SMT solver Z3 [Bjo08]. Finally, at the end of the block, we append the inferred and simplified norm changes. In our experience, the size of the generated integer program can be significantly reduced this way.

y = x	y = x
$y\langle \text{next}^*\rangle \texttt{NULL}_1 = x\langle \text{next}^*\rangle \texttt{NULL}_0$	
x = y→next	x = y→next
$x\langle \text{next}^*\rangle \texttt{NULL}_1 = y\langle \text{next}^*\rangle \texttt{NULL}_1 - 1$	
y = x→next	y = x→next
$y\langle \text{next}^*\rangle \texttt{NULL}_2 = x\langle \text{next}^*\rangle \texttt{NULL}_1 - 1$	
x = y	x = y
$x\langle \text{next}^*\rangle \texttt{NULL}_2 = y\langle \text{next}^*\rangle \texttt{NULL}_2$	$x\langle \text{next}^*\rangle \texttt{NULL}_2 = x\langle \text{next}^*\rangle \texttt{NULL}_0 - 2$

Figure 5.15.: Comparison of translating pointer statements per instruction (on the left) and per basic block (on the right).

[8]Based on our experience with other representations of sets of shapes, such as separation logic or symbolic memory graphs, we believe it would not be difficult with other shape representations either.

[9]In Ranger, we support concatenation (at least to some degree), which requires us to look at sequences of TA transitions to match a single unit. But to simplify the presentation we omit these details.

Our second optimization aims at reducing the *number of tracked norms.* For that, we use a simple heuristic exploiting the underlying shape analysis and the principle of *variable seeding* [BCC+07]. Basically, for each pointer variable x used as a source/target of some norm in $\mathcal{N}_c$, we create a shadow variable x', and remember the position of x at the beginning of a loop by injecting a statement $x' = x$ before the loop. We then use our shape analyser on the extended code to see whether the given variable indeed moves towards the appropriate target location when the loop body is fired once. If we can clearly see that this is not the case due to, e.g. the variable stays at the same location, we remove it from $\mathcal{N}_c$. For illustration, in our example from Section 5.1, we generate two norms $p\langle\text{next}^*\rangle pstop$ and $pstop\langle\text{next}^*\rangle p$ for the loop at *line 13.* Using the above approach, we can see that *pstop* is never moved, $pstop\langle\text{next}^*\rangle p$ is never decreased, and so we can discard it. Moreover, we check which norms decrease at which loop branches (or, more precisely, that cannot be excluded to decrease) and prune away norms that decrease only when some other norm is decreased—we say that such a norm is *subsumed.*

E.g. lets have three norms $n\langle\text{left}^*\rangle$`NULL`, $n\langle\text{right}^*\rangle$`NULL`, $n\langle\text{left+right}^*\rangle$`NULL` and non-deterministic tree traversal. The non-deterministic traversal then leads to a butterfly loop with two branches (one for left and one for right traversal). Norms $n\langle\text{left}^*\rangle$`NULL` and $n\langle\text{right}^*\rangle$`NULL` are decreased only at left and right branches of the butterfly loop respectively. However, the norm $n\langle\text{left+right}^*\rangle$`NULL` is decreased at both branches (i.e. it subsumes decreases of both of the norms) and hence is the only one included in $\mathcal{N}_C$.

```
1 Function void butterfly(Node* x)
2     while x != NULL do
3         if * then
4             x = x→left;
5         else
6             x = x→right;
```

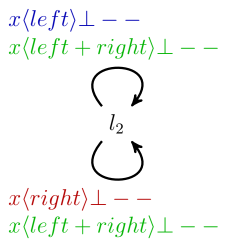

$x\langle left\rangle\bot --$
$x\langle left+right\rangle\bot --$
l_2
$x\langle right\rangle\bot --$
$x\langle left+right\rangle\bot --$

Figure 5.16.: The while cycle at *line 2* is translated into a butterfly loop. Our approach initially generates three candidate norms: $x\langle\text{left}^*\rangle$`NULL`, $x\langle\text{right}^*\rangle$`NULL` and $x\langle\text{left+right}^*\rangle$`NULL`. However, only norm $x\langle\text{left+right}^*\rangle$`NULL` decreases in both branches of the butterfly loop and so can be used as a loop bound. Since $x\langle\text{left+right}^*\rangle$`NULL` decreases every time $x\langle\text{left}^*\rangle$`NULL` or $x\langle\text{right}^*\rangle$`NULL` decreases, we say it subsumes these norms. The subsumed norms can be removed from tracking.

Finally, we reduce the size of the resulting integer program by taking into account only those changes (resets, increments, and decrements) of the norms whose effect can reach the loop for whose analysis the norm is relevant. For that, we use a slight adaptation of the reset graphs introduced in [SZV17].

5.5.4. Experimental Evaluation

Our experiments were performed on a machine with an Intel Core i7-2600@3.4 GHz processor and 32 GiB RAM running Debian GNU/Linux. We compared our prototype Ranger with two other tools: AProVE and Costa. These two tools are, to the best of our knowledge, the closest to Ranger and represent the most recent advancements in bounds analysis of heap-manipulating programs. However, note that both of the tools work over the Java bytecode, and thus we had to translate our benchmarks to Java. For our tool, we report three times: the running time of the shape analysis of Forester (**SA**), generation of the integer program (**IG**), and bounds analysis in Loopus (**BA**). For the other tools, we report times as reported by their web interface[10]. Further, from the outputs of the tools, we extracted the reported complexity of the main program loop, and, if needed, simplified the bounds to the big $\mathcal{O}$ notation. We remark that Costa uses path-based norms (i.e. a subset of our norms), so it is directly comparable with Ranger. AProVE, however, uses norms based on counting all reachable elements, and is therefore orthogonal to us. But, their norms are always bigger than our norms, thus if it reports an equal or bigger computational complexity we can meaningfully compare the results.

The results are summarized in Table 5.2. We use TIMEOUT(60s) if a time-out of 60 seconds was hit, ERROR if the tool failed to run the example[11], and Unknown if the tool could not bound the main loop of the example. We divided our benchmarks to three distinct categories. The Basic category consists of simple list structures — Singly-Linked Lists (SLL) and Circular Singly-Linked Lists (CSLL). In the Advanced Structures category, we infer bounds for programs on more complex structures — Binary Trees (BST), Doubly-Linked Lists (DLL), and even 2-level skip-lists (2-LVL SL). The last category Advanced Algorithms includes experiments with various more advanced algorithms, including show cases taken from related work.

In benchmarks marked with (*), AProVE returned an incorrect bound in our experiments. Further, in benchmarks marked with (**), we obtained different bounds from different runs of AProVE even though it was run in exactly the same way. In both cases, we were unable to find the reason.

The results confirm that our approach, conceived as highly parametric in the underlying shape and bounds analyses, allowed us to successfully combine an advanced shape analysis with a state-of-the-art implementation of amortized resource bounds analysis. Due to this, we were able to fully automatically derive tight complexity bounds even over data structures such as 2-level skip-lists, which are challenging even for safety analysis, and to get more precise and tight bounds for algorithms like Partitions or Func-Queue, which require amortized reasoning to get the precise bound. The most encouraging result is the fully automatically computed precise linear bound for the `mergeInner` method [Atk11]. While AProVE was able to process the example, it was still not able to infer the precise interplay between the traversals of the involved SLL partitions and numeric values needed to compute the precise linear bound.

[10]We could not directly compare the tools on the same machine due to the tool availability issues.
[11]However, we verified that all our examples are syntactically correct.

Of course, our path-based norms do have their limitations too. They are, e.g. not sufficient to verify algorithms like the Deutsch-Schorr-Waite tree traversal algorithm or tree destruction algorithms, which could be verified using norms, supported, e.g. by AProVE, based on counting all memory locations reachable from a given location. We thus see an approach combining such norms (perhaps with suitably bounded scope) with our norms as an interesting direction of future research along with a better support of norms based on data stored in dynamic data structures.

5.6. Conclusion and Future Directions

We have proposed a novel parametric class of shape norms that express the distance between two distinct points through selector paths. Further, we proposed a transformation of an input heap-manipulating program into a corresponding integer representation that can be followed by state-of-the-art resource bounds analysers for integer programs. Our approach managed to outperform the state-of-the-art methods on a series of programs either manipulating non-trivial data structures (such as trees, skip-lists or singly-linked lists) or requiring amortized reasoning for inferring precise resource bounds.

However, there are still many further optimizations possible. First, our class of norms is currently limited to simple selector paths only, and hence we would naturally like to extend the calculus to a richer variety of selector paths, in particular, concatenation and iteration of selector paths. Moreover, we could combine our path-based norms with size-based norms as defined, e.g. in AProVE [FG17]. At last, we would like to adapt bi-abduction techniques [CDOY11, LGQC14, LQC15] for resource bounds analysis that could allow us to analyse open programs and to be applied in the practice.

Table 5.2.: Experimental results.

Benchmark	Short description	Real bounds	RANGER				APROVE		COSTA	
			Bound	**SA**	**IG**	**BA**	**Time**	**Bound**	**Time**	**Bound**
BASIC										
SLL-CST	Constant-length SLL Traversal	$\mathcal{O}(1)$	$\mathcal{O}(1)$	0.002s	0.023s	0.011s	$\mathcal{O}(1)$	3.664s	$\mathcal{O}(n)$	0.251s
SLL	SLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.012s	0.087s	0.040s	$\mathcal{O}(n)$	6.434s	$\mathcal{O}(n)$	0.441s
SLL-NESTED	SLL with non-reset nested traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.027s	0.256s	0.057s	$\mathcal{O}(n)$	6.361s	$\mathcal{O}(n^2)$	1.582s
SLL-INT	SLL Traversal with int combination	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.037s	0.275s	0.057s	$\mathcal{O}(n)$	8.945s	$\mathcal{O}(n)$	0.921s
CSLL	CSLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.013s	0.086s	0.032s	ERROR		UNKNOWN	0.383s
CSLL-NT	Non-terminating CSLL Traversal	$\mathcal{O}(\infty)$	$\mathcal{O}(\infty)$	0.003s	0.001s	0.011s	ERROR		UNKNOWN	0.843s
ADVANCED STRUCTURES										
DLL-NEXT	Forward DLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.034s	0.518s	0.036s	$\mathcal{O}(n)$	5.954s	UNKNOWN	0.657s
DLL-PREV	Backward DLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.031s	0.181s	0.044s	$\mathcal{O}(n)$	6.459s	UNKNOWN	0.712s
DLL-NT	Non-terminating DLL Traversal	$\mathcal{O}(\infty)$	$\mathcal{O}(\infty)$	0.011s	0.004s	0.024s	ERROR		UNKNOWN	0.684s
DLL-INT	Forward DLL Traversal with int combination	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.044s	0.654s	0.044s	$\mathcal{O}(n)$	5.723s	UNKNOWN	0.946s
DLL-PAR	Parallel Forward and Backward DLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.058s	0.510s	0.069s	ERROR		UNKNOWN	0.668s
BUTTERFLY	Terminating Butterfly Loop	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.005s	0.054s	0.024s	$\mathcal{O}(n)$	7.389s	$\mathcal{O}(n)$	0.883s
BUTTERFLY-INT	Terminating Butterfly Loop with int combination	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	0.026s	0.198s	0.059s	$\mathcal{O}(n)^*$	3.513s	UNKNOWN	0.899s
BUTTERFLY-NT	Non-terminating Butterfly Loop	$\mathcal{O}(\infty)$	$\mathcal{O}(\infty)$	0.005s	0.090s	0.015s	$\mathcal{O}(n)^*$	7.768s	UNKNOWN	1.701s
BST-DOUBLE	Leftmost BST Traversal with nested Rightmost	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	25.147s	12.523s	0.203s	$\mathcal{O}(n^2)^{**}$	14.547s	UNKNOWN	3.004s
BST-LEFT	Leftmost BST Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	2.947s	7.321s	0.171s	$\mathcal{O}(n)^{**}$	13.335s	UNKNOWN	2.476s
BST-RIGHT	Rightmost BST Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	2.895s	5.779s	0.168s	$\mathcal{O}(n)^{**}$	13.007s	UNKNOWN	2.457s
BST-LR	Random BST Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	3.331s	7.010s	0.188s	$\mathcal{O}(n)^{**}$	14.488s	UNKNOWN	2.619s
2-LVL SL-L1	2-lvl Skip-list Traversal via lvl1	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.309s	0.837s	0.036s	ERROR		UNKNOWN	1.449s
2-LVL SL-L2	2-lvl Skip-list Traversal via lvl2	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.096s	0.526s	0.042s	ERROR		UNKNOWN	1.442s
ADVANCED ALGORITHMS										
FUNCQUEUE	Queue implemented by two SLLs	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.046s	0.519s	0.136s	$\mathcal{O}(n)$	8.222s	UNKNOWN	4.808s
PARTITIONS	SLL Partitioning	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.094s	0.729s	0.059s	$\mathcal{O}(n^2)$	8.526s	$\mathcal{O}(n^2)$	7.047s
INSERTSORT	Insert Sort on SLL	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	0.041s	0.288s	0.051s	$\mathcal{O}(n^2)$	6.453s	$\mathcal{O}(n^2)$	0.904s
MERGEINNER	Showcase example of Atkey [Atk11]	$\mathcal{O}(n)$	$\mathcal{O}(n)$	3.589s	14.080s	1.502s	$\mathcal{O}(n^2)$	57.935s	TIMEOUT(60s)	

6. Conclusion and Future Directions

The main goal of this thesis was to improve the state of the art of formal analysis and verification of systems with infinite state space and with the focus on techniques based on automata. In particular, we addressed this goal in two distinct parts.

In the first part, we focused on weak monadic second-order logic of one successor (WS1S): a highly expressive, yet decidable, theory that was successfully applied in several formal analyses and verification methods. First, we limited ourselves to formulae in the prenex normal form and proposed an antichain-based decision procedure. The procedure checks the validity of a formula by constructing a corresponding finite automaton for the matrix (i.e. the quantifier free sub-formula) of the given formula followed by processing the prefix of quantifiers by recursively computing the fixpoint of final resp. nonfinal states. Finally, the method concludes that the formula is valid if the intersection of initial and final states is non-empty. We further optimized this procedure by a generalization of the antichain-based universality checking which allows us to considerably reduce the explored state space. We demonstrated the efficiency of this method on a series of both artificial formulae and formulae describing invariants of programs manipulating with singly-linked lists beating the state-of-the-art methods.

We further generalized the procedure to arbitrary formulae. We proposed a systematic way to express the formulae as so-called language terms and check validity of the formulae using an on-the-fly algorithm. We optimized the basic algorithm by two main techniques: a antichain-based pruning of the state space and a lazy evaluation of the sub-terms. We evaluated the procedure on a series of formulae used for verification of programs manipulating singly-linked lists or arrays. Our second procedure outperformed both the state-of-the-art methods as well as our initial approach by several orders of magnitude.

In the second part of the thesis, we focused on resource bounds analysis of heap-manipulating programs. We proposed a novel parametric class of shape norms that express the distance between two distinct points through selector paths (such as the norm $x\langle \text{next}^* \rangle$`NULL` expressing the lengths of paths through the `next` selector from the variable x to null pointer). Based on this class of norms, we designed a method that transforms an input heap-manipulating program into a corresponding integer representation. We then analyse the resulting integer program using state-of-the-art resource bounds analysers for integer programs. In order to construct the integer representation efficiently, we propose to (1) derive the norms directly from the program, (2) use a calculus that infers changes of norms according to the results of the shape analysis, and (3) prune the set of tracked norms based on several heuristics. We evaluated our approach on a series of programs either manipulating non-trivial data structures or requiring amortized reasoning for inferring precise resource bounds. Our procedure managed to outperform the state-of-the-art methods both in terms of the speed and the precision of the bounds.

All of our contributions were implemented as tools. The first antichain based method was implemented as a prototype tool called dWiNA [FHLV14] and the second lazy method was implemented in a tool called Gaston [FHJ+16]. We implemented the novel resource bounds analyser Ranger [FHR+18b] on top of the Forester and Loopus tools.

6.1. Further Directions

In the introduction, we discussed that the current state of the art of performance analysis of complex data structures is still less developed than, e.g. analysis of integer programs. While our contributions have hopefully pushed the usability border of both performance analysis and WS1S logic, we still think that there is a lot of potential directions we could follow or enhance.

We mentioned that WS1S has many applications, e.g, as an underlying theory for specification of invariants of linear data structures. The next natural step is to extend our procedures to WS2S — weak monadic second-order logic of two successors — which would enable us to model properties of more complex data structures, such as trees. However, one will have to cope with a more complex type of automata, in particular, tree automata. For tree automata, however, even some basic operations, such as subsumption testing or the simulation relation, are more complex and more expensive. Moreover, while our procedures performed well on many formulae, e.g. describing properties of arrays [ZHW+14] or singly-linked lists [MQ11], they still failed on many other due to a state space explosion — an inherent property of WS1S. We believe that we could achieve further state space reduction by adapting, e.g. simulation-based techniques [Cé17] both on the generated automata as well as by weakening the term-subsumption relation. Another possible reduction could be achieved by integration of our approach with SAT and SMT techniques or by adapting some of the techniques proposed by the authors of the Mona tool [KMS02].

In the second part, we proposed a resource bounds analysis of the heap-manipulating programs based on a new class of norms, which model numerical measures such as lengths of lists by selector paths between distinct points. Our class of norms is currently limited to simple selector paths only, and so we would like to extend its calculus to a richer variety of selector paths. In particular, we would like to support concatenation (to support sequential traversals in the control flow) and iteration (to support nested cycles in the control flow) of selector paths. Moreover, we believe we could infer resource bounds for a wider class of programs if we combined our path-based norms with size-based norms as defined, e.g. in AProVE [FG17]. At last, we would like to extend our approach to analysis of open programs. We believe that adapting bi-abduction techniques [CDOY11, LGQC14, LQC15] for resource bounds analysis could allow us to scale better and to be applied in the practice.

In this thesis, we mainly researched possibilities of static analysis for performance analysis of programs. Another direction that we are currently exploring is dynamic analysis of programs. In particular, we propose to model the performance of programs based on real data captured from actual program runs. One can then model the performance of a program using, e.g. regression, non-parametric [WJ94, HTF09], or multivariate [Dev11] analyses. Based on these models, we believe we can automatically detect performance changes between two distinct versions of programs, i.e. detect from pairs of models for current and some baseline version of a program that the performance considerably degraded. Moreover, we wish to explore possibilities of using, e.g. fuzz-testing for triggering performance changes in program runs. Finally, we would like to develop an optimized collection of resource data from real runs by limiting the analysis only to subset of program units (functions, etc.) that impacts the program performance the most. Of course, a question is how to find this subset.

6.2. Publications Related to this Thesis

We developed two decision procedures for WS1S logic. The first one based on antichains was initially published in TACAS'15 [FHLV15]; its extended version with additional proofs and more thorough examples was published in the Acta Informatica journal [FHLV19]. Our follow-up work, which generalised the original decision procedure to arbitrary formulae based on lazy techniques and on-the-fly exploration of the state space was published in TACAS'17 [FHJ+17].

In the field of performance analysis of heap-manipulating programs, we built on efficient approaches of the Loopus and Forester tools and proposed a parametric framework for amortized resources bounds analysis published in VMCAI'18 [FHR+18a].

In summary, we developed three tools. The implementations of antichain based and lazy decision procedures for WS1S logic called dWiNA [FHLV14] and Gaston [FHJ+16] respectively. Further, we extended the Forester tool into a Ranger tool [FHR+18b] which translates input heap-manipulating programs into corresponding integer programs. At last, our ongoing work is currently developed as the Perun tool [FGL+18].

Bibliography

[AAG$^+$08] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: Design and implementation of a cost and termination analyzer for Java bytecode. In *Proc. of FMCO'07*, volume 5382 of *LNCS*, pages 113–132. Springer, 2008.

[AAG$^+$12] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. Automatic inference of resource consumption bounds. In *Proc. of LPAR'12*, volume 7180 of *LNCS*, pages 1–12. Springer, 2012.

[ABH$^+$08] Parosh Aziz Abdulla, Ahmed Bouajjani, Lukáš Holík, Lisa Kaati, and Tomáš Vojnar. Computing simulations over tree automata: Efficient techniques for reducing tree automata. In *Proc. of TACAS'08*, volume 4963 of *LNCS*, pages 93–108. Springer, 2008.

[ACH$^+$10] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. When simulation meets antichains (on checking language inclusion of NFAs). In *Proc. of TACAS'10*, volume 6015 of *LNCS*, pages 158–174. Springer, 2010.

[AFHG15] Thomas Ströde Cornelius Aschermann, Florian Frohn, Jera Hensel, and Jürgen Giesl. AProVE: Termination and memory safety of C programs (competition contribution). In *Proc. of TACAS'15*, LNCS, pages 417–419. Springer, 2015.

[AHH$^+$13] Parosh A. Abdulla, Frédéric Haziza, Lukáš Holík, Bength Jonsson, and Ahmed Rezine. An integrated specification and verification technique for highly concurrent data structures. In *Proc. of TACAS'13*, volume 7795 of *LNCS*, pages 324–338. Springer, 2013.

[Atk11] Robert Atkey. Amortised resource analysis with separation logic. *Logical Methods in Computer Science*, 7(2), 2011.

[BBH$^+$11] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomáš Vojnar. Programs with lists are counter automata. *Formal Methods in System Design*, 38(2), 2011.

[BCC$^+$07] Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter O'Hearn. Variance analyses from invariance analyses. In *Proc. of POPL'07*, pages 211–224. ACM, 2007.

[BG00] Doron Bustan and Orna Grumberg. Simulation based minimization. In *Proc. of CADE'00*, volume 1831 of *LNCS*, pages 255–270. Springer, 2000.

[BHH$^+$08] Ahmed Bouajjani, Peter Habermehl, Lukáš Holík, Tayssir Touili, and Tomáš Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In *Proc. of CIAA'08*, volume 5148 of *LNCS*, pages 57–67. Springer, 2008.

[BHV04] Ahmed Bouajjani, Peter Habermehl, and Tomáš Vojnar. Abstract Regular Model Checking. In *Proc. of CAV'04*, volume 3114 of *LNCS*, pages 372–386. Springer, 2004.

[Bjo08] Nikola Bjorner. The Z3 Theorem Prover, 2008. Available from: `https://github.com/Z3Prover/z3/`.

[Büc59] Julius Richard Büchi. Weak second-order arithmetic and finite automata. Technical report, The University of Michigan, 1959. Available from `http://hdl.handle.net/2027.42/3930`.

[CDG$^+$08] Hubert Comon, Max Dauchet, Remi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications.* 2008.

[CDNQ12] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006–1036, 2012.

[CDOY11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6):26:1–26:66, December 2011.

[Cé17] G. Cécé. Foundation for a series of efficient simulation algorithms. In *In Proc. of LICS'17*, pages 1–12. IEEE, June 2017.

[DBCO06] Dino Distefano, Josh Berdine, Byron Cook, and Peter W. O'Hearn. Automatic Termination Proofs for Programs with Shape-shifting Heaps. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[Dev11] Jay L. Devore. *Probability and Statistics for Engineering and the Sciences.* Brooks/Cole, 8th edition, January 2011. ISBN-13: 978-0-538-73352-6.

[DR10] Laurent Doyen and Jean-François Raskin. Antichain algorithms for finite automata. In *Proc. of TACAS'10*, volume 6015 of *LNCS*, pages 2–22. Springer, 2010.

[DV14] Loris D'Antoni and Margus Veanes. Minimization of symbolic automata. In *In Proc. of POPL'14*, volume 49, pages 541–554. ACM, 2014.

[EKM98] Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: new techniques for WS1S and WS2S. In *Proc. of CAV'98*, volume 1427 of *LNCS*, pages 516–520. Springer, 1998.

[FG17] Florian Frohn and Jürgen Giesl. Complexity analysis for Java with AProVE. In *Proc of IFM'17*, pages 85–101. Springer, 2017.

[FGL$^+$18] Tomáš Fiedor, Martina Grzybowská, Matuš Liščinský, Jiří Pavela, Radim Podola, and Šimon Stupinský. Perun: Performance version system, 2018. Available from `https://github.com/tfiedor/perun`.

[FHJ$^+$16] Tomáš Fiedor, Lukás Holík, Petr Janků, Ondřej Lengál, and Tomáš Vojnar. GASTON, 2016. Available from `http://www.fit.vutbr.cz/research/groups/verifit/tools/gaston/`.

[FHJ$^+$17] Tomáš Fiedor, Lukáš Holík, Petr Janků, Ondřej Lengál, and Tomáš Vojnar. Lazy automata techniques for WS1S. In *Proc. of TACAS'17*, number 10205, pages 407–425. Springer Verlag, 2017.

[FHLV14] Tomáš Fiedor, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. `dWiNA`, 2014. Available from `http://www.fit.vutbr.cz/research/groups/verifit/tools/dWiNA/`.

[FHLV15] Tomáš Fiedor, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. Nested antichains for WS1S. In *Proc. of TACAS'15*, volume 9035 of *LNCS*. Springer, 2015.

[FHLV19] Tomáš Fiedor, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. Nested antichains for WS1S. *Acta Informatica*, 2019.

[FHR$^+$18a] Tomáš Fiedor, Lukáš Holík, Adam Rogalewicz, Moritz Sinn, Tomáš Vojnar, and Florian Zuleger. From shapes to amortized complexity. In *Proc. of VMCAI'18*, number 10747 in LNCS, pages 205–225. Springer, 2018.

[FHR$^+$18b] Tomáš Fiedor, Lukáš Holík, Adam Rogalewicz, Moritz Sinn, Tomáš Vojnar, and Florian Zuleger. RANGER, 2018. Available from `http://www.fit.vutbr.cz/research/groups/verifit/tools/ranger/`.

[GK10] Tobias Ganzow and Lukasz Kaiser. New algorithm for weak monadic second-order logic on inductive structures. In *Proc. of CSL'10*, volume 6247 of *LNCS*, pages 366–380. Springer, 2010.

[GLS09] Sumit Gulwani, Tal Lev-Ami, and Mooly Sagiv. A combination framework for tracking partition sizes. In *Proc. of POPL'09*, pages 239–251, 2009.

[HHL$^+$15a] Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Counterexample Validation and Interpolation-Based Refinement for Forest Automata. In *Proc. of VMCAI'17*, volume 10145 of *LNCS*. Springer, 2015.

[HHL$^+$15b] Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Forester: Shape analysis using tree automata (competition contribution). In *Proc. of TACAS'15*, volume 9035 of *LNCS*. Springer, 2015.

[HHR$^+$12] Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jirí Šimáček, and Tomáš Vojnar. Forest automata for verification of heap manipulation. *Formal Methods in System Design*, 41(1):83–106, 2012.

[HIRV07] Peter Habermehl, Radu Iosif, Adam Rogalewicz, and Tomáš Vojnar. Proving Termination of Tree Manipulating Programs. In *Proc. of ATVA'07*, volume 4762 of *LNCS*. Springer, 2007.

[HJK10] Jad Hamza, Barbara Jobstmann, and Viktor Kuncak. Synthesis for regular specifications over unbounded domains. In *Proc. of FMCAD'10*, pages 101–109. IEEE, 2010.

[HR13] Martin Hofmann and Dulma Rodriguez. Automatic type inference for amortised heap-space analysis. In *Proc. of ESOP'13*, number 7792 in LNCS. Springer, 2013.

[HŠRV13] Lukáš Holík, Jiří Šimáček, Adam Rogalewicz, and Tomáš Vojnar. Fully Automated Shape Analysis Based on Forest Automata. In *Proc. of CAV'13*, LNCS. Springer, 2013.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, 2nd edition, 2009.

[IRŠ13] Radu Iosif, Adam Rogalewicz, and Jirí Šimáček. The tree width of separation logic with recursive definitions. In *Proc. of CADE'13*, volume 7898 of *LNCS*, pages 21–38. Springer, 2013.

[JSS$^+$12] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proc. of PLDI'12*, pages 77–88, 2012.

[Kla99] Nils Klarlund. A theory of restrictions for logics and automata. In *Proc. of CAV'99*, volume 1633 of *LNCS*, pages 406–417. Springer, 1999.

[KMS02] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002.

[KS93] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proc. of POPL'13*. ACM, 1993.

[LGQC14] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In *Proc. of CAV'14*, pages 52–68. Springer, 2014.

[LQ06] Shuvendu .K. Lahiri and Shaz Qadeer. Verifying Properties of Well-Founded Linked Lists. In *Proc. of POPL'06*. ACM, 2006.

[LQC15] Ton Chanh Le, Shengchao Qin, and Wei Ngan Chin. Termination and non-termination specification inference. In *Proc. of PLDI'15*, LNCS. ACM, 2015.

[LŠV12] Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. VATA: A library for efficient manipulation of non-deterministic tree automata. In *Proc. of TACAS'12*, volume 7214 of *LNCS*, pages 79–94. Springer, 2012.

[MDR16] Roman Manevich, Boris Dogadov, and Noam Rinetzky. From shape analysis to termination analysis in linear time. In *Proc. of CAV'16*. Springer, 2016.

[Mey72] Albert R. Meyer. Weak monadic second order theory of successor is not elementary-recursive. In *Proc. of Logic Colloquium*, volume 453 of *LNM*, pages 132–154. Springer, 1972.

[MPQ] Parthasarathy Madhusudan, Gennaro Parlato, and Xiaokang Qiu. Strand benchmark. `http://web.engr.illinois.edu/~qiu2/strand/`. Accessed: 2014-01-29.

[MPQ11] Parthasarathy Madhusudan, Gennaro Parlato, and Xiaokang Qiu. Decidable logics combining heap structures and data. In *Proc. of POPL'11*, pages 611–622. ACM, 2011.

[MQ11] Parthasarathy Madhusudan and Xiaokang Qiu. Efficient decision procedures for heaps using STRAND. In *Proc. of SAS'11*, volume 6887 of *LNCS*, pages 43–59. Springer, 2011.

[MST10] Tiziana Margaria, Bernhard Steffen, and Christian Topnik. Second-order value numbering. In *Proc. of GraMoT 2010*, volume 30 of *ECEASST*, pages 1–15. EASST, 2010.

[MTLT08] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Thor: A tool for reasoning about shape and arithmetic. In *Proc. of CAV'08*, volume 5123 of *LNCS*. Springer, 2008.

[MTLT10] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proc. of POPL'10*. ACM, 2010.

[NJT13] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *Proc. of MSR'13*, pages 237–246. IEEE, 2013.

[Rey02] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. IEEE CS Press, 2002.

[Rug04] Radu Rugina. Shape analysis quantitative shape analysis. In *Proc. of SAS'04*, volume 3148 of *LNCS*. Springer, 2004.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric Shape Analysis via 3-valued Logic. *Proc. of TOPLAS'02*, 24(3), 2002.

[SZ10] Moritz Sinn and Florian Zuleger. LOOPUS: A tool for computing loop bounds for C programs. In *Proc. of WING'10*, 2010.

[SZV14] Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. pages 745–761, 2014.

[SZV17] Moritz Sinn, Florian Zuleger, and Helmut Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of Automated Reasoning*, 59(1):3–45, 2017.

[Tra15] Dmitriy Traytel. A coalgebraic decision procedure for WS1S. In *CSL'15*, volume 41 of *LIPIcs*, pages 487–503. Schloss Dagstuhl, 2015.

[TWMS06] Christian Topnik, Eva Wilhelm, Tiziana Margaria, and Bernhard Steffen. jMosel: A stand-alone tool and jABC plugin for M2L(Str). In *Proc. of SPIN'06*, volume 3925 of *LNCS*, pages 293–298. Springer, 2006.

[WDHR06] Martin De Wulf, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[WJ94] Matt Wand and Michael Chris Jones. *Kernel smoothing*, volume 60. CRC Press, 1994.

[WMK11] Thomas Wies, Marco Muñiz, and Viktor Kuncak. An efficient decision procedure for imperative tree data structures. In *Proc. of CADE'11*, volume 6803 of *LNCS*, pages 476–491. Springer, 2011.

[YB02] Tuba Yavuz-Kahveci and Tevfik Bultan. Automated verification of concurrent linked lists with counters. In *Proc. of SAS'02*, volume 2477 of *LNCS*, pages 69–84. Springer, 2002.

[ZGSV11] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *Proc. of SAS'11*, 2011.

[ZHW$^+$14] Min Zhou, Fei He, Bow-Yaw Wang, Ming Gu, and Jiaguang Sun. Array theory of bounded elements and its applications. *Journal of Automated Reasoning*, 52(4):379–405, 2014.

[ZKR08] Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In *Proc. of POPL'08*. ACM, 2008.