VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

AUTOMATA IN INFINITE-STATE FORMAL VERIFICATION

AUTOMATY V NEKONEČNĚ STAVOVÉ FORMÁLNÍ VERIFIKACI

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR AUTHOR	Ing. ONDŘEJ LENGÁL
ŠKOLITEL SUPERVISOR	prof. Ing. TOMÁŠ VOJNAR, Ph.D.
ŠKOLITEL SPECIALISTA CO-SUPERVISOR	Mgr. LUKÁŠ HOLÍK, Ph.D.

BRNO 2015

Abstract

The work presented in this thesis focuses on finite state automata over finite words and finite trees, and the use of such automata in formal verification of infinite-state systems. First, we focus on extensions of a previously introduced framework for verification of heap-manipulating programs—in particular programs with complex dynamic data structures—based on tree automata. We propose several extensions to the framework, such as making it fully automated or extending it to consider ordering over data values. Further, we also propose novel decision procedures for two logics that are often used in formal verification: separation logic and weak monadic second order logic of one successor. These decision procedures are based on a translation of the problem into the domain of automata and subsequent manipulation in the target domain. Finally, we have also developed new approaches for efficient manipulation with tree automata, mainly for testing language inclusion and for handling automata with large alphabets, and implemented them in a library for general use. The developed algorithms are used as the key technology to make the above mentioned techniques feasible in practice.

Keywords

Antichains, binary decision diagrams, finite automata, heaps, language inclusion, monadic logic, nondeterminism, regular tree model checking, second-order logic, separation logic, shape analysis, simulation, tree automata, formal verification.

Abstrakt

Tato práce se zaměřuje na konečné automaty nad konečnými slovy a konečnými stromy, a použití těchto automatů při formální verifikaci nekonečně stavových systémů. Práce se nejdříve věnuje rozšíření existujícího přístupu pro verifikaci programů které manipulují s haldou (konkrétně programů s dynamickými datovými strukturami), jenž je založen na stromových automatech. V práci je navrženo několik rozšíření tohoto přístupu, jako například jeho plná automatizace či jeho rozšíření o podporu uspořádaných dat. V práci jsou popsány nové rozhodovací procedury pro dvě logiky, které jsou často používány ve formální verifikaci: pro separační logiku a pro slabou monadickou druhořádovou logiku s následníkem. Obě tyto rozhodovací procedury jsou založeny na převodu jejich problému do automatové domény a následné manipulaci v této cílové doméně. Posledním přínosem této práce je vývoj nových algoritmů k efektivní manipulaci se stromovými automaty, s důrazem na testování inkluze jazyků těchto automatů a manipulaci s automaty s velkými abecedami, a implementace těchto algoritmů v knihovně pro obecné použití. Tyto vyvinuté algoritmy jsou použity jako klíčová technologie, která umožňuje použití výše uvedených technik v praxi.

Klíčová slova

Antiřetězce, analýza tvaru, binární rozhodovací diagramy, druhořádová logika, haldy, jazyková inkluze, konečný automat, monadická logika, nedeterminismus, regulární stromový model checking, separační logika, simulace, stromový automat, formální verifikace.

Citace

Ondřej Lengál, Automata in Infinite-State Formal Verification, disertační práce, Brno, FIT VUT v Brně, 2015

Automata in Infinite-State Formal Verification

Prohlášení

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

. . . .
Ondřej Lengál
9. března 2015

Acknowledgements

First and foremost, I thank my supervisors Tomáš Vojnar and Lukáš Holík for dragging me kicking and screaming into the beautiful magical land of theoretical computer science and showing me what doing research is about. The adventure has been amazing, their guidance essential, and fighting the beasts—the problems—dwelling in this land rewarding. Moreover, I am also grateful to them for introducing me to other knights that fearlessly roam this realm and teaching me to appreciate their wisdom and cunning. I thank my co-authors, listed in the alphabetical order: Parosh Aziz Abdulla, Constantin Enea, Tomáš Fiedor, Bengt Jonsson, Adam Rogalewicz, Mihaela Sighireanu, Jiří Šimáček, and Cong Quy Trinh. Further, I am grateful to the other past and present members of the VeriFIT research group for creating a friendly research environment. I thank the people who hosted me at their institutions during the years: Parosh Aziz Abdulla (Uppsala University), Yu-Fang Chen and Bow-Yaw Wang (Academia Sinica), and Constantin Enea, Peter Habermehl, and Mihaela Sighireanu (LIAFA, Université Paris Diderot). From those not already mentioned above, I thank Anthony Widjaja Lin and members of the Division of Computer Systems of Uppsala University for interesting discussions. I am thankful to my friends that live outside the realm, who helped me find a sanctuary in my times of need. I am also grateful to my family for supporting me and letting me pursue my dreams.

The work presented in this thesis was supported by the Czech Ministry of Education, Youth, and Sports (under the long-term institutional project MSM 0021630528 and the project OC10009), the Czech Science Foundation (under the projects 202/13/37876P, P103/10/0306, 102/09/H042, 13-37876P, and 14-11384S), the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, the internal BUT FIT projects FIT-S-12-1 and FIT-S-14-2486, and the action IC0901 of the COST association.

Contents

1. Introduction

Computer-based systems and technologies keep penetrating still deeper into human lives. The importance of their uninterrupted and correct operation thus keeps growing. Today, computer systems are widely used in the automotive industry (currently, there are over 30 microcontrollers in an average car), aerospace industry, telecommunication, bank sector, military, etc. An incorrect behaviour of a computer system in some of these environments may cause substantial loses of money, resources, or, in the worst case, even human lives. Even in cases of programs that are not *safety-critical*, errors are often the cause of a negative user experience, which can lead to frustration, and, in an extreme case, even to damage to hardware.

Verification is a process that checks whether a given system is *correct* with respect to a provided *specification.* There are two main approaches to verification: the so-called bug hunting and formal verification. *Bug hunting* methods focus on finding as many errors as possible in the verified system. This approach includes testing of programs using random inputs while observing their behaviour, dynamic analysis (extrapolation of program's dynamic behaviour), some forms of static analysis (such as detection of errors that match some patterns in the source code), bounded model checking (systematic search of the state space of the verified program to a limited depth), etc. Bug hunting methods usually cannot guarantee a program's correctness and often find only easily reachable errors.

1.1. Formal Verification

Formal verification is, as opposed to bug-hunting, a technique that attempts to *formally* prove that the verified system is error-free, i.e. formal verification can guarantee that if it does not find an error, there are indeed no errors present in the system. Although the formal verification problem is in general *undecidable*, there are currently various formal verification methods that work well for a large range of classes of programs.

Several properties are often required from formal verification methods. Perhaps the most important of these properties is *soundness.* A method is said to be sound in case it never pronounces a system error-free when the system contains a behaviour that violates the specification. On the other hand, a method is said to be *complete* if it does not produce *spurious counterexamples*, i.e. counterexamples that in fact can never occur in the real system. A desired property of formal verification algorithms is also *termination*, i.e. that the algorithms always converge.

1.2. Shape Analysis of Programs Manipulating Heap

One particular class of errors are the ones relating to *memory safety* in programs that use dynamic memory allocation, such as programs manipulating different flavours of lists (e.g. singly/doubly linked, circular, with skip pointers) and trees (e.g. binary trees, trees with root/parent pointers). The area that investigates techniques for dealing with them is called *shape analysis*. Examples of the considered errors are invalid pointer dereference (which may cause a corruption of data values or an abnormal termination of the program) or occurrence of garbage (which may cause the program to deplete the memory available and even affect other programs running on the computer). Dynamic memory is utilised (either directly or indirectly via library calls) in a vast portion of currently produced software. Among the most critical applications that extensively use dynamic memory are kernels of operating system (e.g. Linux) and various standard libraries (e.g. the GNU C library `glibc` or the C++ standard library).

Because programs manipulating heap are usually infinite-state, a sound analysis technique needs to represent the heap *symbolically*, i.e. represent sets of heaps by different means than enumerating all of their elements. Currently, there are several competing approaches for symbolic heap representation. The first approach is based on the use of formulae of various logics to describe sets of heap configurations. The logics used are e.g. separation logic [Rey02, MTLT10, BCC$^+$07, GVA07, CDNQ12a, CRN07, YLB$^+$08, CDOY09, DPV13, CDNQ12b, LGQC14], monadic second-order logic [MS01, JJSK97, MPQ11, MQ11], or other [SRW02, ZKR08, BR06]. Another approach is based on the use of automata. In this approach, elements of languages of the automata describe configurations of the heap [BHRV06, BBH$^+$11, DEG06]. The last approach that we will mention is based on graph grammars describing (sets of) heap graphs [HNR10, Wei12]. The presented approaches differ in their degree of specialisation for a particular class of data structures, their efficiency, and their level of dependence on user assistance (such as definition of loop invariants or inductive predicates for the considered data structures).

The works that build on *separation logic*, such as [BCC$^+$07, YLB$^+$08, LGQC14], are among the more efficient ones, thanks to the support for local reasoning provided by the *separating conjunction* (separating conjunction effectively decomposes the heap into disjoint components so that each can be handled independently of the others, without the need to consider all possible aliasings of their elements). However, most of the techniques based on separation logic are either specialised for some particular data structure—such as singly/doubly linked lists—and even a slight change in the data structure can make the technique unusable (as e.g. in [BCC$^+$07, YLB$^+$08, DPV13]), or they need the user to provide inductive definitions of the used data structures. Moreover, when testing for a fixpoint (which is done to detect whether a newly obtained symbolic representation is subsumed by some already existing one), the analysis needs to check entailment of a pair of separation logic formulae. Entailment procedures have so far been either for considerably limited classes of data structures (e.g. singly linked lists), or quite ad-hoc, based on folding/unfolding inductive predicates in the formulae and trying to obtain a syntactic proof of the entailment. Obviously, this often came with no completeness guarantee. Only recently have there appeared more systematic approaches [IRŠ13, IRV14].

The shape analysis techniques based on automata can address this issue by exploiting the generality of the automata-based representation. Finite tree automata, for instance, have been shown to provide a good balance between efficiency and ability to express complex data structures. In particular, the so-called *abstract regular tree model checking* (ARTMC) of heap-manipulating programs [BHRV12] uses a finite tree automaton to describe a set of heaps positioned on a tree backbone (non-tree edges of the heap are represented using regular "routing" expressions describing how the target can be reached from the source using tree edges). Manipulation with the heap is represented using a finite tree transducer and the set of reachable configurations is computed by iteratively applying the transducer on the initial configuration, until a fixpoint is reached. At each step, the obtained symbolic configuration is safely over-approximated using abstraction—which collapses certain states of the automaton—and a fixpoint is detected by standard automata language inclusion testing. The abstraction used is derived automatically during the run of the analysis, using the so-called *counterexample guided abstraction refinement* (CEGAR) technique, which uses spurious counterexamples to refine the abstraction. This formalism is able to fully automatically verify even as complex data structures as binary trees with linked leaves, however, it suffers from the inefficiency of the monolithic encoding of the sets of heaps and the transition relation.

Recently, a technique borrowing the best from the worlds of separation logic and ARTMC emerged. This technique, introduced in [HHR$^+$12], is based on the so-called *forest automata*, which are essentially tuples of tree automata where leaves of the trees accepted by one of the tree automata can reference roots of the trees accepted by the other tree automata (or by itself). This "non-monolithic" encoding gives a support for local reasoning because heap manipulating operations are executed as simple operations locally on a particular tree automaton and not affecting the other tree automata in the forest automaton. Each root of a tree corresponds to a *cut-point* (a node with multiple incoming edges) in the heap graph. Some data structures have an unbounded number of cut-points, e.g. doubly linked lists wherein every internal node is a cut-point. Data structures of this kind cannot be represented in a finite way using this basic formalism; the number of tree components of the forest automata in the analysis would keep growing. The approach therefore uses *hierarchical encoding*, which uses special symbols—called *boxes*—to encode sets of subgraphs that contain a cut-point. Boxes are, again, represented using forest automata. The technique uses automata abstraction from ARTMC to obtain a sound over-approximation of the set of reachable configurations and accelerate obtaining a fixpoint of the analysis.

1.2.1. Selected Problems in Shape Analysis

One issue of the techniques described in the previous is that they often ignore the data component of the represented data structure. This is not always feasible because several data structures, such as binary search trees or skip lists, depend on the data stored inside—in a binary search tree, for example, if a new value is inserted, the ordering relation between the inserted value and the data stored in the root of the tree determines whether the new value is inserted into the left or the right subtree. Examples of works also considering data stored in data structures are [MPQ11, MQ11, QGSM13].

Another interesting problem emerging in the frameworks for shape analysis is the problem of detecting whether the analysis of symbolic executions of a loop has reached a fixpoint. A symbolic execution is an abstract execution of the program that uses the symbolic representation of the program's memory (there may be a potentially unbounded number of them, the same as for real program executions). In this case, the fixpoint is a *closed* representation of the set of reachable configurations of the heap, with *closed* meaning that any new iteration over the body of the loop cannot add anything new to the set. A fixpoint is detected by testing inclusion of the symbolically represented sets of states before and after one more execution of the loop. The analyses based on separation logic perform such a test by checking *entailment* of a pair of formulae describing the heap configurations. On the other hand, in the analyses based on automata, this test corresponds to checking inclusion of languages of a pair of automata. Also note that both of these problems are general and used in other settings, such as in deductive verification when deducing whether a precondition of a statement and its semantics imply its postcondition (for entailment), or testing containment of a pair of XML schemas (for tree automata language inclusion), among many others. These problems are theoretically very hard with a discouraging worst case complexity, yet good heuristics can often solve an average case in reasonable time.

An example of such a heuristic is the technique of the so-called *antichains* for checking language inclusion of a pair of nondeterministic finite state automata (over finite words or trees). The technique [WDHR06, DR10, BHH$^+$08, ACH$^+$10] avoids explicit determinisation of the automata by performing an on-the-fly exploration of the state space. During the exploration, it prunes parts of the state space using a subsumption relation on sets of states of the original automaton (the simplest form of the relation, introduced in [WDHR06], is simple set inclusion). Although language inclusion of a pair of nondeterministic automata has a forbidding worst case complexity—it is a **PSPACE**-complete problem for finite word automata and, even worse, **EXPTIME**-complete problem for finite tree automata—the technique works well for many practical examples.

1.3. Goals of the Thesis

The main goal of this thesis is an improvement of current state of the art in shape analysis. This goal consists of the following three subgoals. The first subgoal is the development of extensions to the shape analysis technique proposed in [HHR$^+$12] that would extend its degree of automation and class of programs it can handle, with a particular focus on data-dependent programs. The second subgoal is an extension and development of new efficient algorithms for testing entailment and validity of selected logics that are used in shape analysis, in particular separation logic and monadic second-order logic. For both of the logics, there exist fragments for which there have been developed efficient translations of decision problems in the logics into finite (tree) automata; such fragments are the particular focus of our attention. For separation logic, we consider the fragment where higher-order inductive predicates correspond to linked lists of many different kinds (singly and doubly linked, circular, nested, . . .), and for monadic second-order logic, we consider its weak fragment of one successor (the so-called weak monadic second-order logic of 1 successor—WS1S). The third subgoal of this thesis is development of techniques for efficient manipulation with finite tree automata, which underlie the previous two subgoals. In particular, the emphasis is placed on the development of algorithms for efficient testing of inclusion over nondeterministic tree automata, and on techniques for manipulating tree automata with large alphabets.

1.4. An Overview of the Achieved Results

This section summarises the contributions to the particular areas exposed in the previous section as goals of this thesis.

Fully Automated Shape Analysis with Forest Automata. The original paper on forest automata-based shape analysis [HHR$^+$12] relied on the user to provide together with the verified program also the needed boxes—i.e. the forest automata describing subgraphs of the heap to be *enclosed* into higher-level symbols. The first contribution of this thesis is the development of a fully automated method for discovering suitable boxes directly during the run of the analysis. The proposed method is based on selecting a suitable subgraph of the heap, isolating it as a box, and removing a cut-point by *folding* the selected subgraph into a single hyper-edge that is labelled with the box descriptor.

The challenging part is identifying which subgraphs to fold. In general, these need to be subgraphs that decrease the number of cut-points in the heap. However, some more complex conditions need to be met when the method is applied in the analysis. Firstly, the considered subgraph needs to be *small* enough so that the created box that represents it is reusable and the widening operator can make a loop in some tree automaton over the box. Secondly, on the contrary to the previous point, the subgraph needs to be *large* enough so that the box effectively helps to remove a cut-point from the heap graph. The second condition is needed because folding a finite number of input edges of a node with an unbounded in-degree into a box may be sometimes harmful and may even prevent the algorithm to find a more suitable subgraph and terminate.

We developed an algorithm that searches the heap graph for basic subgraphs (called *knots*) that match the particular conditions. The search starts from the knots smallest in the number of cut-points and proceeds to larger ones. During the search, knots are saturated in order to avoid the problems of too small subgraphs mentioned earlier; on the other hand, the algorithm keeps them as small as possible to allow the created boxes to be reused. The procedure developed in this contribution allowed us to fully automatically verify programs with such complex dynamic data structures as various flavours of singly/doubly linked (circular and/or nested) lists, trees, as well as skip lists (after the addition of data mentioned below).

Extending Forest Automata with Support for Data. A further contribution of this thesis is an extension of the forest automata-based framework with a support of programs with ordered data. In this extension, forest automata are augmented with constraints that can relate values stored in the nodes of the represented data structure. There are two types of constraints: (i) *local constraints*, which are used in tree automata transitions and relate data values occurring in the parent node to data values occurring in the subtrees of children nodes, and (ii) *global constraints*, which are used to relate two tree automata with respect to the data values that occur in the trees they can generate. The addition of constraints required further extension of the abstract transformers, which need to introduce new constraints where implied by the performed operation, and remove constraints that do not hold any more. Furthermore, to transform forest automata into the canonical form to make testing language inclusion possible, we devised a saturation procedure that traverses a forest automaton and infers new constraints from the existing ones. Using this extension, we were able to fully automatically verify programs with binary search trees and a full implementation of a 3-level skip list [Pug90], which is, to the best of our knowledge, the first time anyone has achieved this.

A Decision Procedure for Separation Logic with List Predicates. A further contribution of this thesis is the development of a decision procedure for the problem of testing entailment of a pair of formulae in a fragment of separation logic. The considered fragment supports a wide range of higher-order inductive predicates that describe various flavours of singly and doubly linked lists, including nested lists and skip lists. The developed decision procedure is based on finding a homomorphism between the symbolic heaps represented by the separation logic formulae, splitting the heaps into subgraphs according to this homomorphism and component-wise translating the separation logic formulae describing the subgraphs into trees and tree automata and checking membership of the trees in the languages of the tree automata.

An Antichain-based Technique for Deciding WS1S Formulae. As the penultimate contribution, we propose a decision procedure for the WS1S logic (the weak monadic second-order logic of 1 successor). The decision procedure checks, for a WS1S formula φ, whether φ is valid or not. The standard procedure is based on constructing a finite automaton for φ, starting by creating finite automata for the atoms of φ and then going upwards alongside the syntax tree of φ and performing finite automata operations corresponding to the logical operators, eventually creating a finite automaton representing φ, and checking whether its language is non-empty. The drawback of this procedure is that each negation and quantifier alternation yields complementation of an automaton, for which there is no known algorithm that avoids exponential explosion in the number of states (because it includes determinisation of the automaton). The exponential construction induced by complementation makes the procedure infeasible for larger formulae. We propose a method that avoids explicit complementation of the automata but exploits a technique that generalises the *antichains* principle used in algorithms for efficient testing of language inclusion over nondeterministic automata. Note that the multiple-exponential worst case complexity is unavoidable, because the inherent theoretical complexity of the addressed problem is **NONELEMENTARY**, i.e. it cannot be solved by a k-**EXPTIME** algorithm for any fixed k.

Efficient Algorithms for Nondeterministic Tree Automata. Finally, in order to make the previously described contributions usable in practical settings, they need the support of an efficient implementation of operations for manipulating the underlying automata representation. The majority of the previously existing automata-based techniques were based on the use of deterministic automata and suffered from the state explosion that comes with determinisation. The state explosion prevents the use of deterministic automata for larger systems. To avoid the state explosion, we always use nondeterministic automata and techniques that manipulate directly those. We never determinise them, even for such operations as testing language inclusion (which is usually done by determinising and complementing one of the automata and testing emptiness of the intersection with the other automaton). This (and also other problems from the wide area of applications of tree automata) poses the requirement for techniques that can efficiently execute operations directly on nondeterministic tree automata. Concretely, in addition to standard automata operations, such as constructing a union or an intersection of a pair of automata, there is also the requirement for efficient techniques for testing language inclusion. Although testing language inclusion of nondeterministic tree automata has an extreme worst case complexity (being an **EXPTIME**-complete problem), using clever heuristics—which avoid explicit determinisation of a tree automaton used by textbook algorithms causing an exponential state explosion—this can be done efficiently in many practical cases.

We propose a downward inclusion testing algorithm for nondeterministic tree automata, which, in contrast to already existing algorithms, traverses the automata top-down rather than bottom-up. In addition, it uses antichains and the simulation relation to prune parts of the search space that are subsumed by the already explored ones. In our experiments, this algorithm was in the majority of cases the fastest algorithm for testing language inclusion over tree automata. We also developed efficient algorithms for manipulating semi-symbolic representations of nondeterministic tree automata, which can be advantageously used for tree automata with large alphabets—such as those the emerge in the proposed decision procedure for WS1S—also including algorithms for efficient testing of inclusion or computation of simulation relations. We implemented the proposed algorithms in the Vata library, which has since been used by quite a few researchers around the world, who have used it as an efficient underlying library for handling nondeterministic automata for their own techniques.

1.5. Plan of the Thesis

Chapter 2 contains preliminaries on graphs, trees, and tree automata. Part I contains the following three chapters describing our contributions to the forest automata-based shape analysis. Chapter 3 introduces forest automata. Chapter 4 describes the approach taken to make the analysis based on forest automata fully automated using box learning. Chapter 5 describes the extension of the forest automata framework to support reasoning about heap-manipulating programs that depend on ordered data stored in the heap. Part II is dedicated to the description of the decision procedures for two logics: separation logic (Chapter 6) and WS1S (Chapter 7). Finally, Part III focuses on efficient techniques for manipulation of nondeterministic tree automata. In particular, Chapter 8 describes the proposed downward inclusion checking technique for nondeterministic tree automata, Chapter 9 proposes a symbolic encoding of nondeterministic tree automata with large alphabets, and Chapter 10 covers the design and implementation of an efficient tree automata library. The last chapter, Chapter 11, concludes the thesis.

2. Preliminaries

This section formally introduces concepts that will be used in the rest of the thesis, in particular graphs, trees, and tree automata.

2.1. Graphs and Trees

Given a word $\alpha = a_1 \ldots a_n$, where $n \geq 1$, we write α_i to denote its i-th symbol a_i. We use the symbol ϵ for the empty word. For a total map $f : A \to B$, we use $dom(f)$ to denote its domain A and $img(f)$ to denote its image in B.

A *ranked alphabet* is a (potentially infinite) set of symbols Σ associated with a mapping $\# : \Sigma \to \mathbb{N}_0$ that assigns ranks to symbols. A (directed, ordered, labelled) *graph* over Σ is a total map $g : V \to \Sigma \times V^*$ which assigns to every *node* $v \in V$ (1) a *label* from Σ, denoted as $\ell_g(v)$, and (2) a sequence of *successors* from V^*, denoted as $S_g(v)$, such that $\#\ell_g(v) = |S_g(v)|$. We drop the subscript g if no confusion may arise. Nodes v with $S(v) = \epsilon$ are called *leaves*. For any $v \in V$ such that $g(v) = (a, v_1 \cdots v_n)$, we call the pair $v \mapsto (a, v_1 \cdots v_n)$ an *edge* of g. The *in-degree* of a node in V is the overall number of its occurrences in $g(v)$ across all nodes $v \in V$. The nodes of a graph g with an in-degree larger than one are called *joins* of g.

A *path* from v to v' in g is a sequence $p = v_0, i_1, v_1, \ldots, i_n, v_n$ where $v_0 = v$, $v_n = v'$, and for each j such that $1 \leq j \leq n$, v_j is the i_j-th successor of v_{j-1}. The path is *empty* if $n = 0$. The path is *acyclic* if none of nodes $v_0, \ldots, v_n$ appears twice in it. The nodes $v_1, \ldots, v_{n-1}$ are called the *inner nodes* of p. The *length* of p is defined as $length(p) = n$. The path is a cycle if $v_0 = v_n$, and it is a *simple cycle (or loop)* if it is a cycle and no node except $v_0 = v_n$ appears twice in it. An acyclic path has defined the *cost* as the sequence $i_1, \ldots, i_n$. We say that p is *cheaper* than another path p' iff the cost of p is lexicographically smaller than that of p'. A node u is *reachable/accessible* from a node v iff there is a path from v to u in g (including the case when the path is empty, i.e. $u = v$). A node v that reaches all nodes of g is called the *root* of g. If such a node exists in the graph g, we say that g is *rooted (in v)*. A *tree* is a graph t that has exactly one root r and each of its nodes except r is a successor of at most one node v of t. We use $root(t)$ to denote the root of t and T_Σ to denote the set of all trees over Σ.

2.2. Tree Automata

A (finite, nondeterministic) *tree automaton* (TA) is a quadruple $\mathcal{A}$ defined as $\mathcal{A} = (Q, \Sigma, \Delta, R)$ where Q is a finite set of *states*, Σ is a ranked alphabet, Δ is a finite set of *transitions*, and $R \subseteq Q$ is a set of *root states*. Each transition is a triple of the form $(q, a, q_1 \cdots q_n)$ where $n \geq 0$, $q, q_1, \ldots, q_n \in Q$, $a \in \Sigma$, and $\#a = n$. We often use interchangeably $q \to a(q_1, \ldots, q_n)$ and $a(q_1, \ldots, q_n) \to q$ to denote $(q, a, q_1 \cdots q_n)$, depending whether we wish to emphasise the downward or the upward direction of the transition. In the special case where $n = 0$, we speak about the so-called *leaf transitions*. We use $Q^{\#}$ to denote the set of all tuples of states from Q with up to the maximum arity that some symbol in Σ has, i.e. if $n = \max_{a \in \Sigma} \#a$, then $Q^{\#} = \bigcup_{0 \leq i \leq n} Q^i$.

For $q \in Q$ and $a \in \Sigma$, we use $down_a(q)$ to denote the set of tuples accessible from q over a in the top-down manner; formally, $down_a(q) = \{(q_1, \ldots, q_n) \mid q \to a(q_1, \ldots, q_n)\}$. For $a \in \Sigma$ and $(q_1, \ldots, q_n) \in Q^{\#a}$, we denote by $up_a((q_1, \ldots, q_n))$ the set of states accessible from $(q_1, \ldots, q_n)$ over the symbol a in the bottom-up manner; formally, $up_a((q_1, \ldots, q_n)) = \{q \mid a(q_1, \ldots, q_n) \to q\}$. We also extend these notions to sets in the usual way, i.e. for $a \in \Sigma$, $P \subseteq Q$, and $S \subseteq Q^{\#a}$, $down_a(P) = \bigcup_{p \in P} down_a(p)$ and $up_a(S) = \bigcup_{(s_1, \ldots, s_n) \in S} up_a((s_1, \ldots, s_n))$.

A *run* of $\mathcal{A}$ over a tree t over Σ is a mapping $\rho : dom(t) \to Q$ s.t. for each node $v \in dom(t)$ where $q = \rho(v)$, if $q_i = \rho(S(v)_i)$ for $1 \leq i \leq |S(v)|$, then Δ has a transition $q \to \ell(v)(q_1, \ldots, q_{|S(v)|})$. We write $t \Longrightarrow_\rho q$ to denote that ρ is a run of $\mathcal{A}$ over t s.t. $\rho(root(t)) = q$. We use $t \Longrightarrow q$ to denote that $t \Longrightarrow_\rho q$ for some run ρ. The *language* of a state q is defined by $L(q) = \{t \mid t \Longrightarrow q\}$, and the *language* of $\mathcal{A}$ is defined by $L(\mathcal{A}) = \bigcup_{q \in R} L(q)$. We extend the notion of a language to a tuple of states $(q_1, \ldots, q_n) \in Q^n$ by letting $L((q_1, \ldots, q_n)) = L(q_1) \times \cdots \times L(q_n)$. The language of a set of n-tuples of sets of states $S \subseteq (2^Q)^n$ is the union of languages of elements of S, the set $L(S) = \bigcup_{E \in S} L(E)$.

Simulations. A *downward simulation* on a TA $\mathcal{A} = (Q, \Sigma, \Delta, R)$ is a preorder relation $\preceq_D \subseteq Q \times Q$ such that if $q \preceq_D p$ and $q \to a(q_1, \ldots, q_n)$, then there are states $p_1, \ldots, p_n$ such that $p \to a(p_1, \ldots, p_n)$ and $q_i \preceq_D p_i$ for each $1 \leq i \leq n$. Given a TA $\mathcal{A} = (Q, \Sigma, \Delta, R)$ and a downward simulation $\preceq_D$, an *upward simulation* $\preceq_U \subseteq Q \times Q$ induced by $\preceq_D$ is a relation such that if $q \preceq_U p$ and $a(q_1, \ldots, q_n) \to q'$ with $q_i = q$, $1 \leq i \leq n$, then there are states $p_1, \ldots, p_n, p'$ such that $a(p_1, \ldots, p_n) \to p'$ where $p_i = p$, $q' \preceq_U p'$, and $q_j \preceq_D p_j$ for each j such that $1 \leq j \neq i \leq n$.

2.3. Structured Labels

Sometimes, we will work with alphabets where symbols, called *structured labels*, have an inner structure. Let Γ be a ranked alphabet of *sub-labels*, ordered by a total ordering $\sqsubset_\Gamma$. We will work with graphs over the alphabet 2^Γ where for every symbol $A \subseteq \Gamma$, its arity is $\#A = \sum_{a \in A} \#a$. Let $e = v \mapsto (\{a_1, \ldots, a_m\}, v_1 \cdots v_n)$ be an edge of a graph g where $n = \sum_{1 \leq i \leq m} \#a_i$ and $a_1 \sqsubset_\Gamma a_2 \sqsubset_\Gamma \cdots \sqsubset_\Gamma a_m$. We decompose e into a sequence of m *sub-edges* $e\langle 1 \rangle = v \to (a_1, v_1 \cdots v_{\#a_1}), \ldots, e\langle m \rangle = v \to (a_m, v_{n-\#a_m+1} \cdots v_n)$. We call $e\langle i \rangle = v \to (a_i, v_k \cdots v_l)$ from the sequence the i-th sub-edge of e in g, for $1 \leq i \leq m$. We use $SE(g)$ to denote the set of all sub-edges of g, and $SE(g, v)$ for the subset of $SE(g)$ where v is the origin.. We say that a node v of a graph is *isolated* if it does not appear within any sub-edge, neither as an origin (i.e. $\ell(v) = \emptyset$) nor as a target. A graph g without isolated nodes is unambiguously determined by $SE(g)$ and vice versa (due to the total ordering $\sqsubset_\Gamma$ and since g has no isolated nodes).

A counterpart of the notion of sub-edges in the context of transitions of TAs is the notion of sub-terms, defined as follows: Given a transition $\delta = q \to \{a_1, \dots, a_m\}(q_1, \dots, q_n)$ of a TA over the alphabet of structured labels 2^Γ, *sub-terms* of δ are the terms $\delta\langle 1\rangle = a_1(q_1, \dots, q_{\#a_1}), \dots, \delta\langle m\rangle = a_m(q_{n-\#a_m+1}, \dots, q_n)$ where $\delta\langle i\rangle$, for $1 \leq i \leq m$, is called the *i-th sub-term of* δ.

Part I.

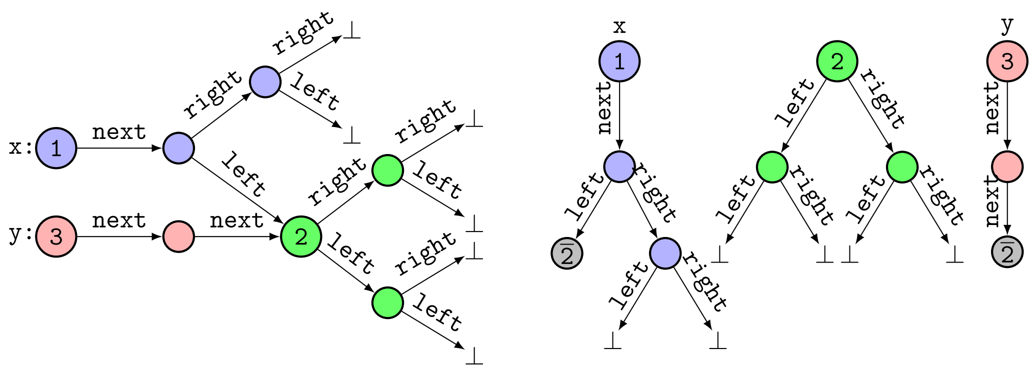
Forest Automata-Based Formal Verification of Programs

3. Shape Analysis with Forest Automata

In this chapter, as the starting point for our own work presented in Chapters 4 and 5, we will briefly describe the forest automata-based shape analysis framework for verification of programs manipulating complex dynamic data structures, as introduced by Habermehl *et al* in [HHR^{+}12]. The main concept of the symbolic representation used in the framework is the so-called *forest decomposition* of a heap graph, which is performed as follows: First, the *cut-points* of the graph are identified; a cut-point is a node that is either referenced by a program variable or is a target of multiple edges. Every cut-point is then taken as the root of a (cut-point-free) tree component whose leaves are either nodes with no outgoing edges, or other cut-points. The heap graph is split into the tree components. The tree components are then canonically ordered according to the order in which their roots were visited in a depth-first search (DFS) through the graph, when starting from program variables. In the tree components, any leaf that corresponds to a cut-point numbered with c during the DFS is changed into an explicit reference to the cut-point number c, written as $\bar{c}$. See Figure 3.1 for an illustration of the forest decomposition of a heap graph.

To represent a *set* of (potentially infinite) heaps $H = \{h_1, h_2, \dots\}$ with the same number n of cut-points, we decompose all heaps of H into forests and for every position $1 \leq i \leq n$, we then collect the i-th components of all forests into the set $H[i] = \{h_1[i], h_2[i], \dots\}$. The set $H[i]$ can be represented using a tree automaton (TA) $\mathcal{A}[i]$ and the whole set of heaps H can be represented by a tuple of TAs $\mathcal{A}[1], \dots, \mathcal{A}[n]$, called a *forest automaton* (FA). (Note that the previous decomposition of a set of heaps can be performed only in the case the set of forests F_H of H is *convex*. Convexity of F_H denotes the fact that we can take any forest $h[1], \dots, h[n]$ from F_H, substitute $h[j]$ with $h'[j]$ for any $1 \leq j \leq n$ and $h' \in H$, and the result will still be a member F_H. Non-convex sets of forests are represented as unions of convex sets. Our analysis also guarantees that all $H[i]$ are regular tree languages.)

An FA of the simple structure presented above cannot be used as a representation of data structures that have an unbounded number of cut-points—such as doubly linked lists (DLLs) or trees with parent pointers, where every internal node is a cut-point—and the analysis would need an infinite number of FAs to represent a set of all instances of these data structures. In order to be able to represent them using finitary means, the forest automata framework allows the use of the so-called *boxes*. Boxes are FAs that are used as symbols of another, *higher level* FA. In this FA, they represent a (complex) subgraph using a single symbol. Intuitively, the task of boxes is to decrease the in-degree of cut-points in a graph—when the in-degree of a node drops to one (and the node is not referenced by a program variable), the node is no longer a cut-point and can be represented by an ordinary state in a TA. In this way, it is possible to represent an



(a) A graph

(b) Its forest representation

Figure 3.1.: A graph and its forest representation

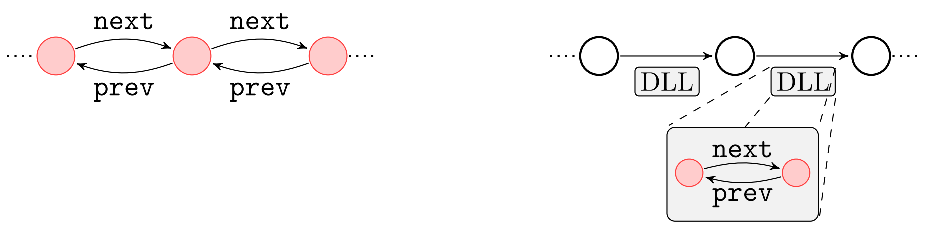
over-approximation of all reachable configurations of a program using forest automata with a bounded number of tree components. See Figure 3.2 for an example of a use of a box in an encoding of a DLL.

Alongside the notion of FAs, [HHR$^+$12] also proposed a shape analysis that uses FAs and is based on the framework of *abstract interpretation* [CC77]. For each program line, a set of forest automata is used to represent the set of memory configurations reachable at a given line. The program is symbolically executed on this representation in such a way that each program statement is mapped to an *abstract transformer* that simulates execution of the statement on the symbolic representation (and also checks whether an error has been encountered). The symbolic execution examines all branches of the program until no new symbolic states can be found on the branches and a fixpoint is obtained (this is detected by testing language inclusion of FAs, see [HHR$^+$12] for more details). Because, as mentioned earlier, programs manipulating heap are usually infinite-state, the *widening* operator is used to provide a sound over-approximation of the set of reachable configurations. This operator is based on automata *abstraction* borrowed from *abstract regular tree model checking* (ARTMC). For a given forest automaton, abstraction collapses some states of the TAs of the FA (for each TA separately), trying to introduce loops into the TAs to obtain TAs accepting an infinite (regular) tree language that over-approximates the original one and, in turn, a forest automaton representing an infinite set of heaps, again over-approximating the original one.

Outline. Section 3.1 of this short chapter introduces the formalism of forest automata and Section 3.2 describes the forest automata-based framework for shape analysis.

3.1. Forest Automata

Forests. Let Σ be a ranked alphabet disjoint from $\mathbb{N}$, i.e. $\Sigma \cap \mathbb{N} = \emptyset$. A Σ-labelled *forest* is a sequence of trees $t_1 \cdots t_n$ over the alphabet $(\Sigma \cup \{1, \ldots, n\})$ where for all $1 \leq i \leq n$, the arity of i is $\#i = 0$. Leaves labelled by $i \in \mathbb{N}$ are called *root references*.



(a) A DLL (b) Its hierarchical encoding

Figure 3.2.: A DLL and its hierarchical encoding

The forest $t_1 \cdots t_n$ represents the graph $\otimes t_1 \cdots t_n$ obtained by uniting the trees of $t_1 \cdots t_n$, assuming w.l.o.g. that their sets of nodes are disjoint, and interconnecting their roots with the corresponding root references. Formally, $\otimes t_1 \cdots t_n$ contains an edge of the form $v \mapsto (a, v_1 \cdots v_m)$ iff there is an edge $v \mapsto (a, v'_1 \cdots v'_m)$ of some tree t_i, for $1 \leq i \leq n$, such that for all $1 \leq j \leq m$, the following holds: if v'_j is a root reference with $\ell(v'_j) = k$ then $v_j = root(t_k)$, otherwise $v_j = v'_j$.

Graphs and forests with ports. We will further work with graphs with designated input and output nodes. An *io-graph* is a pair (g, ϕ), abbreviated as g_ϕ, where g is a graph and $\phi \in dom(g)^+$ a sequence of *ports* in which ϕ_1 is the *input port* and $\phi_2 \cdots \phi_{|\phi|}$ is a sequence of *output ports* such that the occurrence of ports in ϕ is unique. Ports and joins (i.e. nodes with multiple incoming edges) of g are called *cut-points* of g_ϕ. We use $cps(g_\phi)$ to denote all cut-points of g_ϕ. We say that g_ϕ is *accessible* if it is rooted in the input port ϕ_1. We sometimes abuse notation for graphs and use it also for io-graphs, e.g. we may write $dom(g_\phi)$ to denote $dom(g)$.

An io-forest is a pair $f = (t_1 \cdots t_n, \pi)$ such that $n \geq 1$ and $\pi \in \{1, \ldots, n\}^+$ is a sequence of port indices, π_1 is the *input index*, and $\pi_2 \ldots \pi_{|\pi|}$ is a sequence of *output indices*, with no repetitions of indices in π. An io-forest encodes the io-graph $\otimes f$ where the ports of $\otimes t_1 \cdots t_n$ are roots of the trees defined by π, i.e. $\otimes f = (\otimes t_1 \cdots t_n, root(t_{\pi_1}) \cdots root(t_{\pi_n}))$.

Forest automata. A *forest automaton* (FA) over the alphabet Σ is defined as a pair $F = (\mathcal{A}_1 \cdots \mathcal{A}_n, \pi)$ where $n \geq 1$, $\mathcal{A}_1 \cdots \mathcal{A}_n$ is a sequence of tree automata over the alphabet $(\Sigma \cup \{1, \ldots, n\})$, and $\pi \in \{1, \ldots, n\}^+$ is a sequence of port indices as defined for io-forests. The *forest language* of F is the set of io-forests $L_f(F) = L(\mathcal{A}_1) \times \cdots \times L(\mathcal{A}_n) \times \{\pi\}$, and the *graph language* of F is the set of io-graphs $L(F) = \{\otimes h \mid h \in L_f(F)\}$.

Forest automata of a higher level. We let Γ_1 be the set of all forest automata over the alphabet of structured labels 2^Γ and call its elements forest automata over Γ *of level 1*. For $i > 1$, we define Γ_i as the set of all forest automata over ranked alphabets $2^{\Gamma \cup \Delta}$ where $\Delta \subseteq \Gamma_{i-1}$ is any nonempty finite set of FAs of level $i-1$. We denote elements of Γ_i as forest automata over Γ *of level* i. The rank $\#F$ of an FA F in these alphabets is the number of its output port indices. When used in an FA F over $2^{\Gamma \cup \Delta}$, the forest automata from Δ are called *boxes* of F. We write Γ_* to denote $\bigcup_{i \geq 0} \Gamma_i$ and assume w.l.o.g. that Γ_* is ordered by some total ordering $\sqsubset_{\Gamma_*}$.

An FA F of a higher level over Γ accepts graphs where forest automata of lower levels appear as sub-labels. To define the semantics of F as a set of graphs over Γ, we need the following operation of *sub-edge replacement* where a sub-edge of a graph is substituted by another graph. Intuitively, the sub-edge is removed, and its origin and targets are identified with the input and output ports of the substituted graph respectively.

Formally, let g be a graph with an edge $e \in g$ and its i-th sub-edge $e\langle i\rangle = v_1 \to (a, v_2 \cdots v_n)$. Let g'_ϕ be an io-graph with $|\phi| = n$. Assume w.l.o.g. that $dom(g) \cap dom(g'_\phi) = \emptyset$. The sub-edge $e\langle i\rangle$ can be replaced by g' provided that for all $1 \leq j \leq n$ it is that $\ell_g(v_j) \cap \ell_{g'}(\phi_j) = \emptyset$, which means that the node $v_j \in dom(g)$ and the corresponding port $\phi_j \in dom(g'_\phi)$ do not have successors reachable over the same sub-label. If the replacement can be done, the result, denoted $g[g'_\phi/e\langle i\rangle]$, is the graph g_n in the sequence $g_0, \ldots, g_n$ of graphs obtained as follows: The graph g_0 is defined using sub-edges as $SE(g_0) = (SE(g) \setminus \{e\langle i\rangle\}) \cup SE(g')$, and for each $1 \leq j \leq n$, the graph g_j arises from g_{j-1} by (1) deriving a graph h_j by replacing the origin of the sub-edges of the j-th port ϕ_j of g'_ϕ by v_j, (2) redirecting edges leading to ϕ_j to lead to v_j, i.e. replacing all occurrences of ϕ_j in $img(h)_j$ by v_j, obtaining the graph h'_j, and (3) removing ϕ_j. Intuitively, we start by removing $e\langle i\rangle$ from g, proceed by adding g' to the graph and then, one by one, reconnecting edges leading to and leaving the ports of g'_ϕ with the nodes incident with $e\langle i\rangle$ in g. Figure 3.3 shows the sub-edge replacement step including the intermediate graphs.

If the symbol a of the sub-edge $e\langle i\rangle$ in the previous paragraph is an FA and $g'_\phi \in L(a)$, we say that $h = g[g'_\phi/e\langle i\rangle]$ is an *unfolding* of g, written $g \prec h$. Conversely, we say that g arises from h by *folding* g'_ϕ into $e\langle i\rangle$. Let $\prec^*$ be the reflexive transitive closure of $\prec$. The Γ-*semantics* of g is then the set of graphs g' over Γ such that $g \prec^* g'$, denoted $[\![g]\!]_\Gamma$, or just $[\![g]\!]$ if no confusion may arise. For an FA F of a higher level over Γ, we let $[\![F]\!] = \bigcup_{g_\phi \in L(F)}([\![g]\!] \times \{\phi\})$.

Canonicity. We call an io-forest $f = (t_1 \cdots t_n, \pi)$ *minimal* iff the roots of the trees $t_1 \cdots t_n$ are the cut-points of $\otimes f$. A minimal forest representation of a graph is unique up to reordering of $t_1 \cdots t_n$. Let the *canonical ordering* of cut-points of $\otimes f$ be defined by the cost of the cheapest paths leading from the input port to them. We say that f is *canonical* iff it is minimal, $\otimes f$ is accessible, and the trees within $t_1 \cdots t_n$ are ordered by the canonical ordering of their roots (which are cut-points of $\otimes f$). A canonical forest is thus a unique representation of an accessible io-graph. We say that an FA *respects canonicity* iff all forests from its forest language are canonical. (Note that we do not consider *canonical* FAs, due to the reason that there would have to be some canonicity restriction on the component TAs. As for a set of nondeterministic TAs with the same language L, there is no known natural canonical TA accepting L, and even if there were, the cost of conversion to this TA might be too high.) Respecting canonicity makes it possible to efficiently test FA language inclusion by testing TA language inclusion of the respective components of a pair of FAs. This method is precise for FAs of level 1 and sound (not always complete) for FAs of a higher level, see [HHR+12] for more details.

In practice, we keep automata in the so called *state uniform* form, which simplifies maintaining the canonicity respecting form (and it is also useful when abstracting and

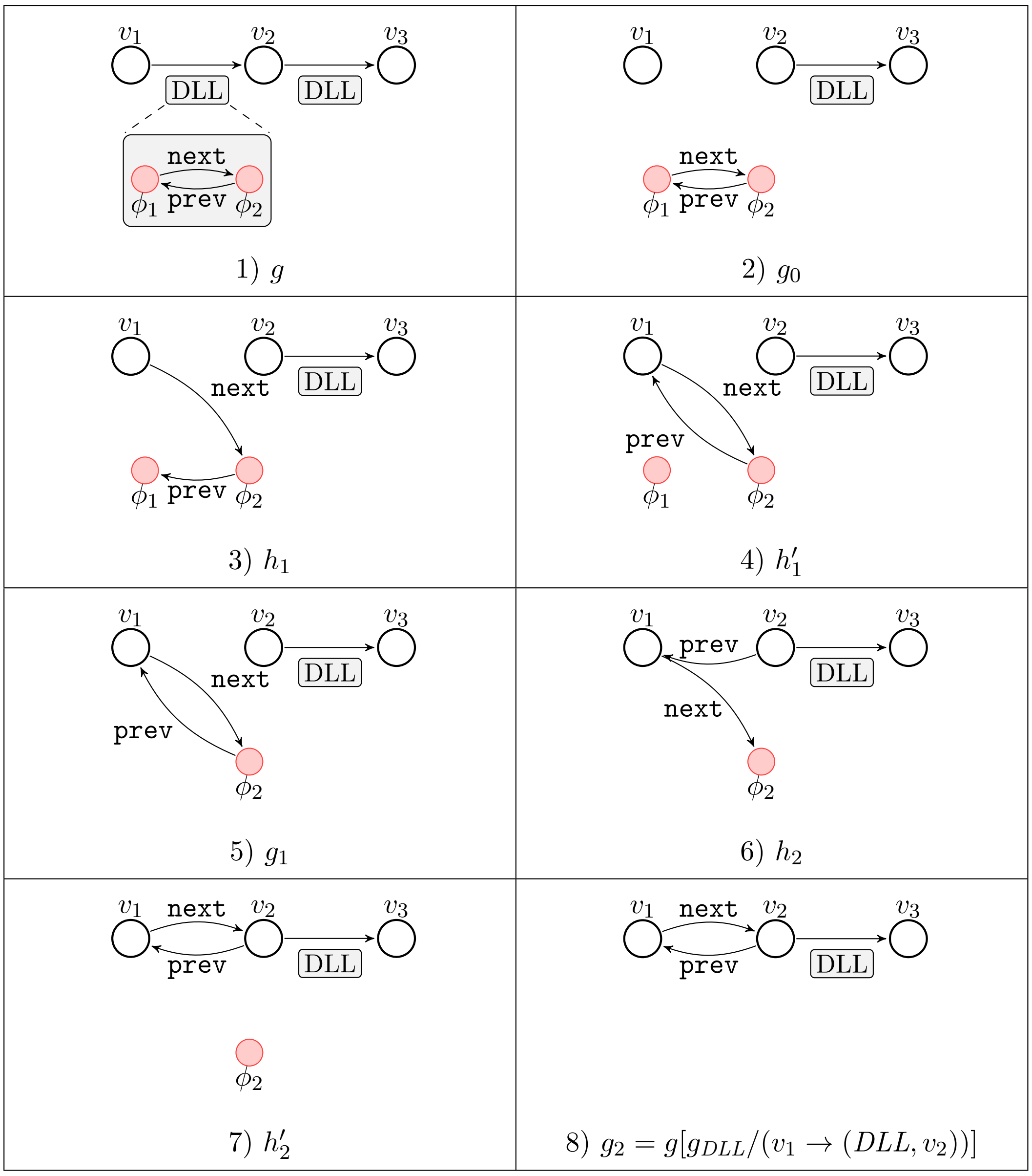


Figure 3.3.: Steps taken in the sub-edge replacement of an edge labelled by the DLL box

"folding", as discussed in Section 4.1.2). It is defined as follows. Given a node v of a tree t in an io-forest, we define its *span* as the pair (α, V) where $\alpha \in \mathbb{N}^*$ is the sequence of labels of root references reachable from the node v ordered according to the cost of the cheapest path to them, and $V \subseteq \mathbb{N}$ is the set of labels of references that occur more than once in the subtree of t rooted in v. The state uniform form then requires that all nodes of forests from $L_f(F)$ that are labelled by the same state q in some accepting run of F have the same span, which we denote by *span*(q).

3.2. FA-based Shape Analysis

We now provide a high-level overview of the main loop of the shape analysis based on forest automata. The analysis automatically discovers memory safety errors (such as invalid dereferences of `null` or undefined pointers, double frees, or memory leaks) and provides an FA-represented over-approximation of the sets of heap configurations reachable at each program line. The framework considers sequential non-recursive C programs manipulating the heap. Each heap cell may have several *pointer selectors* and *data selectors* from some finite data domain (below, *PSel* denotes the set of pointer selectors, *DSel* denotes the set of data selectors, and $\mathbb{D}$ denotes the data domain). Although the implementation of the approach in the Forester tool can handle limited pointer arithmetic and type casting, for the sake of simplicity we do not consider these features in the following description. The analysis can also provide as an output an FA-represented over-approximation of the sets of heap configurations reachable at each program line.

Heap Representation

A single heap configuration is encoded as an io-graph $g_{\texttt{sf}}$ (we describe the input port `sf` later in the text) over the ranked alphabet of structured labels 2^Γ with sub-labels from the ranked alphabet $\Gamma = \mathit{PSel} \cup (\mathit{DSel} \times \mathbb{D})$ with the ranking function that assigns each pointer selector 1 and each data selector 0. In this graph, an allocated memory cell is represented by a node v, and its internal structure of selectors is given by a label $\ell_g(v) \in 2^\Gamma$. Values of data selectors are stored directly in the structured label of a node as sub-labels from $\mathit{DSel} \times \mathbb{D}$, so e.g. a singly linked list cell with the data value 42 and the successor node $x_{\texttt{next}}$ may be represented by a node x such that $\ell_g(x) = \{\texttt{next}(x_{\texttt{next}}), (\texttt{data}, 42)(\epsilon)\}$. Selectors with undefined values are represented in such a way that the corresponding sub-labels are not in $\ell_g(x)$. The null value is modelled as the special node `null` such that $\ell_g(\texttt{null}) = \emptyset$. The input port `sf` represents a special node that contains the *stack frame* of the analysed function, i.e. a structure where selectors correspond to variables of the function.

In order to represent (infinite) *sets* of heap configurations, we use state uniform FAs of a higher level to represent sets of canonical io-forests representing the heap configurations. The FAs used as boxes, i.e. symbols of FAs of a higher level, are provided by the user.

Symbolic Execution

The verification procedure is based on abstract interpretation [CC77] with the abstract domain consisting of sets of state uniform FAs (a single FA does not suffice as FAs are in general not closed under union) representing sets of heap configurations at particular program locations. The computation starts from the initial heap configuration given by an FA for the io-graph $g_{\texttt{sf}}$ where g comprises two nodes: `null` and `sf` where $\ell_g(\texttt{sf}) = \emptyset$ (i.e. the values of all local variables are undefined). The computation then executes abstract transformers corresponding to program statements until the sets of FAs held at program locations stabilise. We note that abstract transformers corresponding to pointer manipulating statements are the most precise transformers. For each operation `op` in the intermediate representation of the analysed program, the semantics of C implies existence of a function $f_{\texttt{op}}$ that, when applied to the io-graph $g_{\texttt{sf}}$, gives the io-graph $f_{\texttt{op}}(g_{\texttt{sf}})$ representing the heap after executing `op`. Based on $f_{\texttt{op}}$, we define for each operation `op` the corresponding abstract transformer $\tau_{\texttt{op}}$ with the property that when $\tau_{\texttt{op}}$ is applied to the set of FAs $\mathcal{S}$, the result is the set of FAs $\mathcal{S}' = \tau_{\texttt{op}}(\mathcal{S})$ such that

$$\bigcup_{F' \in \mathcal{S}'} [\![F']\!] = \{f_{\texttt{op}}(g_{\texttt{sf}}) \mid g_{\texttt{sf}} \in [\![F]\!] \wedge F \in \mathcal{S}\}. \tag{3.1}$$

Executing the abstract transformer $\tau_{\texttt{op}}$ over a set of FAs $\mathcal{S}$ is performed separately for every $F \in \mathcal{S}$. In the first step, we perform *materialisation* during which we *unfold* (i.e. substitute by the corresponding FA) lower-level boxes until the heap nodes being accessed by the given operation are uncovered. Then we perform the actual update—which amounts to manipulation of states in the neighbourhood of a root state, which is quite close to the corresponding manipulation of concrete heap graphs—as described in the following paragraph.

Let us fix the set of stack frame sub-edges $S = SE(g, \texttt{sf})$. Pointer updates of the form $\texttt{x} := \texttt{y}$, $\texttt{x} := \texttt{y->sl}$, or $\texttt{x} := \texttt{null}$ replace the sub-edge $\texttt{sf} \to (\texttt{x}, v_{\texttt{x}})$ in S with the sub-edge $\texttt{sf} \to (\texttt{x}, v'_{\texttt{x}})$, where $v'_{\texttt{x}}$ is obtained according to the type of the update:

(i) For the assignment $\texttt{x} := \texttt{y}$, $v'_{\texttt{x}}$ is a node such that there is a sub-edge $\texttt{sf} \to (\texttt{y}, v'_{\texttt{x}})$ in S. In the case there is no such a sub-edge in S, the sub-edge $\texttt{sf} \to (\texttt{x}, v_{\texttt{x}})$ is removed from S and `x` is left undefined.

(ii) For the assignment $\texttt{x} := \texttt{y->sl}$, $v'_{\texttt{x}}$ is a node such that there is a node $v_{\texttt{y}}$ pointed by `y`, i.e. $\texttt{sf} \to (\texttt{y}, v_{\texttt{y}}) \in S$, where $v_{\texttt{y}}$ points to $v'_{\texttt{x}}$ over `sl`, $v_{\texttt{y}} \to (\texttt{sl}, v'_{\texttt{x}}) \in SE(g)$. In case there is no sub-edge $\texttt{sf} \to (\texttt{y}, v_{\texttt{y}})$ in S or $v_{\texttt{y}} = \texttt{null}$, i.e. `y` is undefined or `null` respectively, the analysis reports an invalid memory access error. On the other hand, if such a sub-edge exists but there is no sub-edge $\texttt{sf} \to (\texttt{y}, v_{\texttt{y}})$, the sub-edge $\texttt{sf} \to (\texttt{x}, v_{\texttt{x}})$ is, again, removed from S and `x` is left undefined.

(iii) Finally, for the assignment $\texttt{x} := \texttt{null}$, $v'_{\texttt{x}}$ is the node `null`.

Updates of the form `x->sl := y` replace the sub-edge $v_{\texttt{x}} \to (\texttt{sl}, z)$ with the sub-edge $v_{\texttt{x}} \to (\texttt{sl}, v_{\texttt{y}})$, where $\texttt{sf} \to (\texttt{y}, v_{\texttt{y}}) \in S$ (or remove $v_{\texttt{x}} \to (\texttt{sl}, z)$ if there is no sub-edge $\texttt{sf} \to (\texttt{y}, v_{\texttt{y}})$ in S). Note that in the case that either `x` is undefined or $v_{\texttt{x}}$ is the `null` node, the analysis reports invalid memory access. Further, symbolic execution of the operation `malloc(x)` replaces the sub-edge $\texttt{sf} \to (\texttt{x}, z)$ with the sub-edge $\texttt{sf} \to (\texttt{x}, v_{new})$, where v_{new} is a newly created node, $v_{new} \notin dom(g)$, where $\ell(v_{new}) = \emptyset$. The call `free(x)` removes the node $v_{\texttt{x}}$ such that $\texttt{sf} \to (\texttt{x}, v_{\texttt{x}}) \in S$ from g, and also removes all sub-edges $v \to (sel, v_{\texttt{x}}) \in SE(g)$, thus making all selectors pointing to $v_{\texttt{x}}$ undefined. Data updates $\texttt{x->data} := \texttt{d}_{new}$ replace the sub-edge $v_{\texttt{x}} \to ((\texttt{data}, \texttt{d}_{old}), \epsilon) \in S$ with the sub-edge $v_{\texttt{x}} \to ((\texttt{data}, \texttt{d}_{new}), \epsilon)$, where $v_{\texttt{x}}$ is a node such that $\texttt{sf} \to (\texttt{x}, v_{\texttt{x}})$ is in S. During these operations, dereferences of `null` and undefined selectors are detected, as well as emergence of garbage (detected when $f_{\texttt{op}}(g_{\texttt{sf}})$ is not accessible). Evaluating a guard on an io-graph $g_{\texttt{sf}}$ amounts to a test of equality of nodes, or equality or inequality of data fields of nodes.

Folding and abstraction. As we have already discussed at the beginning of this chapter, in order to be able to represent infinite sets of configurations of some data structures (in particular those with an unbounded number of cut-points), the analysis needs to use the so-called boxes, which are FAs of a lower level. In the context of the work that introduced forest automata-based shape analysis [HHR$^+$12], the user is required to provide as the input of the analysis a database of boxes; these boxes are then used by the analysis for *folding* of subgraphs.

The folding is performed after an update of the symbolic execution is completed. It takes the database of boxes and for every box, the procedure attempts to find in the FA that represents the current abstract state all substructures matching the structure of the box. Every such substructure is substituted by a sub-term labelled with the box name. This is done repeatedly until nothing more can be folded. The folding step is followed by transformation of the FA into the canonicity respecting form.

At junctions of program paths, the analysis computes unions of sets of FAs. At loop points (junctions at the beginning of a loop), the union is followed by widening. The widening is performed by applying *abstraction* on each FA from the set of FAs obtained at the loop point. The abstraction used is a modification of the abstraction based on tree languages of a finite height—the so-called *finite height abstraction*—from ARTMC [BHRV12], which is applied independently on every component TA in the FA. The finite height abstraction is parameterised by a height k, and it collapses those states of a TA whose tree languages of the height up-to k match.

3.3. Discussion

The results that were presented in the original work on forest automata-based shape analysis [HHR$^+$12] give evidence of the viability of the approach, in the senses of both the expressivity of the underlying formalism (forest automata can indeed represent various singly/doubly linked lists, skip lists, trees, and their (finitely nested) combinations) and in the scalability (thanks to the decomposition of the heap into a tuple of trees and manipulating each of them independently).

The data structures that are unsupported by the forest automata-based analysis in the proposed setting are either data structures that are not hierarchically structured (such as general graphs) or hierarchical data structures with an unbounded level of nesting (such as trees with linked leaves or skip lists of an arbitrary level). Some of the hierarchical data structures of an unbounded level of nesting could be represented by an extension of the formalism that would allow an FA to recursively contain itself (recall that self-reference is forbidden now); however, it is yet not clear how all steps of an analysis based on this extended formalism would be carried out. To give examples of other data structures that the analysis cannot handle, let us mention data structures with complex invariants (such as AVL trees, which rely on balancedness of a tree) or data structures that perform suballocation of their assigned memory (such as the data structures used in memory allocators).

One weak point of the presented analysis lies in the need for the user to provide the boxes for the substructures that the analysis might come upon during its run. As we strive for fully-automated analysis, the next chapter, Chapter 4, presents an approach that addresses this issue and provides a way for the analysis to infer the boxes itself, during its run. Moreover, Chapter 5 extends the formalism and augments the analysis based on it by taking into account ordering relations between the data stored in a data structure. This extension allows us to verify programs with data structures where the invariant depends on the ordering between the data values stored inside memory nodes, which is the case for various sorting algorithms, binary search trees, or procedures for manipulating skip lists.

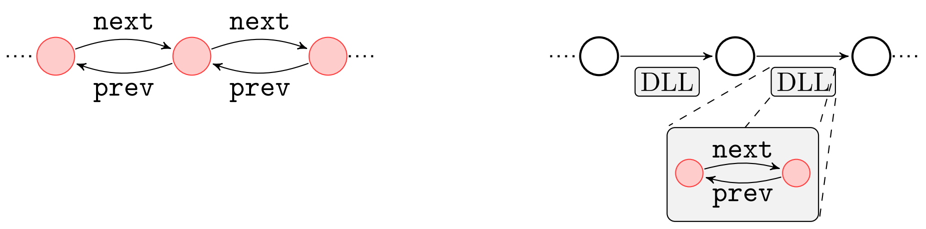
4. Learning Boxes for Forest Automata

The work presented in this chapter is an extension of the shape analysis based on forest automata, as described in Chapter 3. Recall that the described shape analysis relied on the user to provide a suitable set of boxes (the subgraphs to be folded into automata symbols). This means that the user needed to provide the analysis with a forest automata-based description of those data structures used in the program that have an unbounded number of cut-points. As we strive for a *push-button* analysis—an analysis that would run without user interaction, which, we believe and our cooperation with industry partners confirms, is the only kind of analysis that can work for large-scale ever-evolving systems—such an approach is naturally not feasible. To address this issue, we propose an extension of the approach where the boxes are inferred automatically during a run of the analysis using a technique that we call *box learning.*

The basic principle of box learning stems from the reason for which boxes were originally introduced into FAs. In particular, FAs must have a separate component TA for each node (called a *join*) that has multiple incoming edges in the represented graphs. If the number of joins is unbounded (as e.g. in doubly linked lists, abbreviated as DLLs below), unboundedly many component TAs are needed in flat FAs. However, when some of the edges are hidden in a box (as e.g. the `prev` and `next` links of DLLs in Figure 4.1) and replaced by a single box-labelled edge, a finite number of component TAs may suffice. Therefore, the basic idea of our learning is to identify subgraphs of the FA-represented graphs that contain at least one join, and when they are enclosed—or, as we say later on, *folded*—into a box, the in-degree of the join decreases.

There are, of course, many ways to select the above mentioned subgraphs to be used as boxes. To choose among them, we propose several criteria that we found useful in a number of experiments. Most importantly, the boxes must be *reusable* in order to allow eliminating as many joins as possible. The general strategy here is to choose boxes that are *simple* and *small* since these are more likely to correspond to graph patterns that appear repeatedly in typical data structures. For instance, in the already mentioned case of DLLs in Figure 4.1, it is enough to use a box enclosing a single pair of `next/prev` links. On the other hand, as also discussed below, too simple boxes are sometimes not useful either.

Further, we propose a way how box learning can be efficiently integrated into the main analysis loop. In particular, we do not use the perhaps obvious approach of incrementally building a *database of boxes* whose instances would be sought in the generated FAs. We found this approach inefficient due to the costly operation of finding instances of different boxes in FA-represented graphs. Instead, we always try to identify which subgraphs of the graphs represented by a given FA could be folded into a box, followed by looking into the so-far built database of boxes whether such a box has already been introduced or not.



(a) A DLL (b) Its hierarchical encoding

Figure 4.1.: A DLL and its hierarchical encoding

Moreover, this approach has the advantage that it allows one to use simple language inclusion checks for *approximate box folding* which substitutes a subgraph with a box from the database that has a *larger* language, thus over-approximating the set of graphs represented by a given FA. This approach sometimes greatly accelerates the computation. Finally, to further improve the efficiency, we interleave the process of box learning with the *automata abstraction* into a single iterative process. In addition, we propose an FA-specific improvement of the basic automata abstraction which *accelerates the abstraction* of an FA using components of other FAs. Intuitively, it lets the abstraction synthesise an invariant faster by allowing it to combine information coming from different branches of the symbolic computation.

We have prototyped the proposed techniques in Forester and evaluated it on a number of challenging case studies. The results show that the obtained approach is both quite general as well as efficient. For example, we were the first to fully-automatically analyse programs with a data-independent modification of 2- and 3-level skip lists (a modification where the shape invariant of a skip list does not rely on the fact that the list is ordered—our extension to the standard data-dependent skip lists is described in Chapter 5). On the other hand, our implementation achieves performance comparable and sometimes even better than that of Predator [DPV13] (a winner of multiple heap analysis-related awards in several years of the competition on software verification SV-COMP) on list manipulating programs despite being able to handle much more general classes of heap graphs.

Related work. From the point of view of efficiency and degree of automation, the main alternative to our approach are the methods that fully-automatically use separation logic with inductive list predicates as implemented in Space Invader [YLB$^+$08] or SLAyer [BCI11]. These approaches are, however, much less general than our approach since they are restricted to programs over certain classes of linked lists (and cannot handle even structures such as linked lists with data pointers pointing either inside the list nodes or optionally outside of them, which we can easily handle as discussed later on). A similar comparison applies to the Predator tool inspired by separation logic but using purely graph-based algorithms [DPV13]. The work [LYP11] on overlaid data structures mentions an extension of Space Invader to trees, but this extension is of a limited generality and requires some manual help.

In [GVA07], an approach for synthesising inductive predicates in separation logic is proposed. This approach is shown to handle even tree-like structures with additional pointers. One of these structures, namely, the so-called mcf trees implementing trees whose nodes have an arbitrary number of successors linked in a DLL, is even more general than what can in principle be described by hierarchically nested FAs (to describe mcf trees, recursively nested FAs or FAs based on hedge automata would be needed). On the other hand, the approach of [GVA07] seems quite dependent on exploiting the fact that the encountered data structures are built in a "nice" way conforming to the structure of the predicate to be learnt (meaning e.g. that lists are built by adding elements at the end only), which is close to providing an inductive definition of the data structure.

A novel technique based on the so-called second-order bi-abduction was presented in [LGQC14]. This technique tries to infer the most general pre- and post-conditions of functions, expressed in the form of higher-order inductive predicates of separation logic, such that they imply that the analysed program is memory-safe. First, pre- and post-conditions that use unknown predicates (second-order variables) are inferred from the code. Then, the analysis tries to synthesise the most general shape predicates for the unknown predicates such that when the synthesised predicates substitute the unknown predicates in the pre- and post-conditions, the result is consistent. The issue of this approach is that in the analysed program is not memory safe, the analysis cannot give a direct reason why it is so. Instead, the user will just see that the inferred pre- and post-conditions are trivial. Moreover, as in the previous work, this analysis also relies on the way how the data structure is created.

The work [MTLT10] proposes an approach which uses separation logic for generating numerical abstractions of heap manipulating programs allowing for checking both their safety as well as termination. The described experiments include even verification of programs with 2-level skip lists. However, the work still expects the user to manually provide an inductive definition of skip lists in advance. Likewise, the work [CRN07] based on the so-called separating shape graphs reports on verification of programs with 2-level skip lists, but it also requires the user to come up with summary edges to be used for summarizing skip list segments, hence basically with an inductive definition of skip lists. Compared to [MTLT10, CRN07], we did not have to provide any manual aid whatsoever to our technique when dealing with 2-level as well as 3-level skip lists in our experiments.

Finally, from the world of graph grammars, a concept of inferring graph grammar rules for the heap abstraction proposed in [HNR10] has recently appeared in [Wei12]. However, the proposed technique can so far only handle much less general structures than in our case.

Outline. The structure of this chapter is the following. First, Section 4.1 describes how we select the parts of the forest automata to be fold and how the very folding is carried out. Then, in Section 4.2, we talk about the abstraction that is used in the analysis. Afterward, Section 4.3 reports on the experimental results and, finally, Section 4.4 concludes the chapter.

4.1. Learning of Boxes

Sets of graphs with an unbounded number of joins can only be described by FAs with the help of boxes. In particular, boxes allow one to replace (multiple) incoming sub-edges of a join by a single sub-edge, and hence lower the in-degree of the join. Decreasing the in-degree to 1 turns the join into an ordinary node. When a box is then used in a cycle of an FA, it effectively generates an unbounded number of joins.

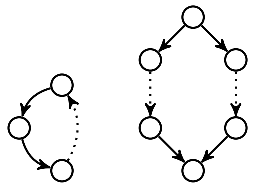
The boxes are introduced by the operation of *folding* of an FA F which transforms F into an FA F' and a box B used in F' such that $[\![F]\!] = [\![F']\!]$. However, the graphs in $L(F')$ may contain less joins since some of them are hidden in the box B, which encodes a set of subgraphs containing a join and appearing repeatedly in the graphs of $L(F)$. Before we explain folding, we give a characterisation of subgraphs of graphs of $L(F)$ which we want to fold into a box B. Our choice of the subgraphs to be folded is a compromise between two high-level requirements. On the one hand, the folded subgraphs should contain incoming edges of joins and be as simple as possible in order to be reusable. On the other hand, the subgraphs should not be too small in order not to have to be subsequently folded within other boxes (in the worst case, leading to generation of unboundedly nested boxes). Ideally, the hierarchical structuring of boxes should respect the natural hierarchical structuring of the data structures being handled since if this is not the case, unboundedly many boxes may again be needed.

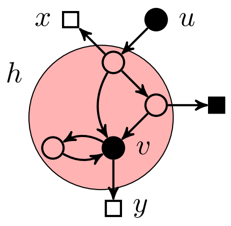
4.1.1. Knots of Graphs

We use $i = g \uplus g'$ to denote a graph i such that $SE(i) = SE(g) \cup SE(g')$. A graph h is a *subgraph* of a graph g iff $SE(h) \subseteq SE(g)$. The *border* of h in g is the subset of the set $dom(h)$ of nodes of h that are incident with sub-edges in $SE(g) \setminus SE(h)$. A *trace* from a node u to a node v in a graph g is a set of sub-edges $t = \{e_1, \ldots, e_n\} \subseteq SE(g)$ such that $n \geq 1$, the sub-edge e_1 is outgoing from u, the sub-edge e_n is entering v, the origin of e_i is one of the targets of e_{i-1} for all $1 \leq i \leq n$, and no two sub-edges in t have the same origin. We call the origins of $e_2, \ldots, e_n$ the *inner nodes* of the trace. A trace from u to v is *straight* iff none of its inner nodes is a cut-point. A *cycle* is a trace from a node v to v. A *confluence* of g_ϕ is either a cycle of g_ϕ or it is the union of two disjoint traces starting at a node u, called the *base*, and ending in the node v, called the *tip* (for a cycle, the base and the tip coincide)—cf. Figure 4.2a.

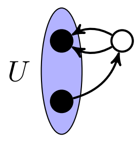
Given an io-graph g_ϕ, the *signature* of a sub-graph h of g is the minimum subset $sig(h)$ of $cps(g_\phi)$ that (1) it contains $cps(g_\phi) \cap dom(h)$ and (2) all nodes of h, except the nodes of $sig(h)$ themselves, are reachable by straight traces from $sig(h)$. Intuitively, $sig(h)$ contains all cut-points of h plus the cut-points of g_ϕ closest to h which lie outside of h but which are needed so that all nodes of h are reachable from the signature. Consider the example of the graph g_u in Figure 4.2b in which cut-points are represented by $\bullet$. The signature of g_u is the set $\{u, v\}$. The signature of the highlighted subgraph h is also equal to $\{u, v\}$.

Given a set $U \subseteq cps(g_\phi)$, a *confluence of* U is a confluence of g_ϕ with the signature in U. Intuitively, the confluence of a set of cut-points U is a confluence whose cut-points


(a) Two types of confluences



(b) Example of a closure



(c) Confluence of a set of cut-points

Figure 4.2.: Notions of confluence and closure

belong to U plus in case the base is not a cut-point, then the closest cut-point from which the base is reachable is also from U (cf. Figure 4.2c).

Finally, for a set $U \subseteq cps(g_\phi)$, we define the *closure* of U (denoted as $cl(U)$) as the smallest subgraph h of g_ϕ such that (1) it contains all confluences of U and (2) for every inner node v of a straight trace of h, it contains all straight traces from v to leaves of g. The closure of the signature $\{u, v\}$ of the graph g_u in Figure 4.2b is the highlighted subgraph h. Intuitively, Point 1 of the requirements on a closure includes into the closure all nodes and sub-edges that appear on straight traces between nodes of U apart from those that do not lie on any confluence (such as node u in Figure 4.2b). Note that nodes x and y in Figure 4.2b, which are leaves of g_u, are not in the closure as they are not reachable from an inner node of any straight trace of h. The *closure of a subgraph* h of g_ϕ is the closure of its signature, and h is *closed* if it equals its closure. In the following, we sometimes use $clsig(\cdot)$ to denote $cl(sig(\cdot))$.

Knots. For the rest of Section 4.1.1, let us fix an io-graph $g_\phi \in L(F)$. We now introduce the notion of a knot which summarises the desired properties of a subgraph k of g that is to be folded into a box. A *knot* k of g_ϕ with *weight* n is a subgraph of g where one of the following holds:

1. k is a confluence such that $n = |sig(k)|$,
2. $k = k' \uplus k''$ where k' and k'' share a sub-edge and their maximum weight is n, or
3. k is the closure of a knot of the weight n.

The weight of k therefore corresponds to the maximum from the numbers of cut-points of all confluences that were used to build up k. Note that it is possible that k may be constructed using different sequences of operations with confluences of potentially different weights. To address this issue, we further define the *complexity* of a knot k as the minimum weight over all possible constructions of k.

An optimal knot of complexity n is a maximal knot of complexity n which has a (possibly more than one) source, and at least one source is reachable from the input port of g_ϕ by a trace that does not intersect with sub-edges of the optimal knot.

The following lemma states some properties of a closure of a knot.

Lemma 4.1. *Given knots k and k' of g_ϕ and their respective closures $clsig(k)$ and $clsig(k')$, and sets of cut-points of g_ϕ α and β, the following properties hold:*

a) $SE(k) \subseteq SE(k') \implies sig(k) \subseteq sig(k')$,

b) $\alpha \subseteq \beta \implies cl(\alpha) \subseteq cl(\beta)$,

c) union preserves signature: for a knot $k'' = k \uplus k'$, it holds $sig(k'') = sig(k) \cup sig(k')$,

d) closure preserves signature: $sig(k) = sig(clsig(k))$,

e) monotonicity: $SE(k) \subseteq SE(k') \implies SE(clsig(k)) \subseteq SE(clsig(k'))$,

f) idempotence: $clsig(k) = clsig(clsig(k))$, and

g) extensivity: $SE(k) \subseteq SE(clsig(k))$.

Proof.

a) Suppose the contrary. Then it must hold that there is an isolated node in k. However, recall that we consider only accessible graphs that do not contain isolated nodes.

b) This clearly holds because $cl(\beta)$ can be computed by computing $cl(\alpha)$ and then adding more sub-edges.

c) We prove this by a simple observation that $sig(k'')$ contains all cut-points in k and k', and that all nodes of k'' are accessible from $sig(k) \cup sig(k')$, in particular the nodes originating from k are accessible from $sig(k)$ and the nodes originating from k' are accessible from $sig(k')$.

d) We first prove that $sig(k) \supseteq sig(clsig(k))$ and then prove $sig(k) \subseteq sig(clsig(k))$.

   1) $sig(k) \supseteq sig(clsig(k))$: To prove this direction we first observe that due to Point 1 of the definition of a closure, the cut-points on confluences of $clsig(k)$ are only those from $sig(k)$. Second, it is easy to see that Point 2 adds no new cut-points to the closure.

   2) $sig(k) \subseteq sig(clsig(k))$: We prove this direction using induction on the structure of k. For the base case when k is a confluence (Point 1 of the definition of a knot), from the definition of a closure, because closure contains all confluences of the set of nodes, it follows that $clsig(k)$ contains k, formally $SE(k) \subseteq SE(clsig(k))$. From Lemma 4.1a it follows that $sig(k) \subseteq sig(clsig(k))$.

   For the case when $k = k' \uplus k''$ for some knots k' and k'' (Point 2), we introduce the induction hypotheses $sig(k') \subseteq sig(clsig(k'))$ and $sig(k'') \subseteq sig(clsig(k''))$. Obviously, the following pair of inclusions holds:

$$\begin{aligned} sig(k') \subseteq sig(clsig(k')) \cup sig(clsig(k'')), \\ sig(k'') \subseteq sig(clsig(k')) \cup sig(clsig(k'')), \end{aligned} \tag{4.1}$$

and, therefore,

$$sig(k') \cup sig(k'') \subseteq sig(clsig(k')) \cup sig(clsig(k'')). \tag{4.2}$$

Further, it is easy to see that $sig(cl(\gamma_1)) \cup sig(cl(\gamma_2)) = sig(cl(\gamma_1) \uplus cl(\gamma_2))$, so we obtain

$$sig(k') \cup sig(k'') \subseteq sig(clsig(k') \uplus clsig(k'')). \tag{4.3}$$

Next, from

$$\begin{aligned} k' &\subseteq k' \uplus k'', \\ k'' &\subseteq k' \uplus k'' \end{aligned} \tag{4.4}$$

we obtain, using Lemma 4.1a and Lemma 4.1b that

$$\begin{array}{c} sig(k') \subseteq sig(k' \uplus k'') \\ sig(k'') \subseteq sig(k' \uplus k'') \end{array} \quad \text{and} \quad \begin{array}{c} clsig(k') \subseteq clsig(k' \uplus k'') \\ clsig(k'') \subseteq clsig(k' \uplus k'') \end{array}. \tag{4.5}$$

From this, it follows that

$$clsig(k') \uplus clsig(k'') \subseteq clsig(k' \uplus k''), \tag{4.6}$$

which implies, again using Lemma 4.1a, that

$$sig(clsig(k') \uplus clsig(k'')) \subseteq sig(clsig(k' \uplus k'')). \tag{4.7}$$

Combining Equations 4.3 and 4.7, we obtain the following inclusion:

$$sig(k') \cup sig(k'') \subseteq sig(clsig(k' \uplus k'')). \tag{4.8}$$

We infer that $sig(k' \uplus k'') \subseteq sig(clsig(k' \uplus k''))$ and conclude with $sig(k) \subseteq sig(clsig(k))$.

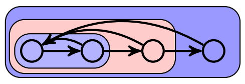
For the last case when $k = clsig(k')$ for a knot k' (Point 3), we use the induction hypothesis $sig(k') \subseteq sig(clsig(k'))$. From the induction hypothesis, Lemma 4.1a and Lemma 4.1b, we conclude that $sig(k') \subseteq sig(clsig(k')) \implies sig(clsig(k')) \subseteq sig(clsig(clsig(k')))$. From this, it follows that $sig(k) \subseteq sig(clsig(k))$.

e) Follows from Lemma 4.1a and Lemma 4.1b.

f) Follows from Lemma 4.1d.

g) We prove this part using induction on the structure of k. In the base case when k is a confluence (Point 1 of the definition of a knot), k is a confluence of $sig(k)$ and will therefore be contained in $clsig(k)$, therefore $SE(k) \subseteq SE(clsig(k))$.

For the case when $k = k' \uplus k''$ where k' and k'' are knots (Point 2 of the definition of a knot), we use the induction hypotheses $SE(k') \subseteq SE(clsig(k'))$ and $SE(k'') \subseteq SE(clsig(k''))$. It holds that $SE(k') \subseteq SE(k)$ and so $SE(clsig(k')) \subseteq SE(clsig(k))$ (Lemma 4.1e). Therefore, $SE(k') \subseteq SE(clsig(k')) \subseteq SE(clsig(k))$, so it holds


(a) A list with head pointers

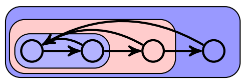

(b) A doubly linked list

Figure 4.3.: Knots in graphs

that $SE(k') \subseteq SE(clsig(k))$. The same holds for k'', therefore we conclude that $SE(k') \cup SE(k'') \subseteq SE(clsig(k))$.

If k is the closure of a knot k', $k = clsig(k')$, (Point 3) then Lemma 4.1f claims that $clsig(k') = clsig(clsig(k'))$ and so $SE(k) = SE(clsig(k))$, which proves the lemma. □

Note that the properties 4.1e, 4.1f, and 4.1g are the typical properties of a standard closure operator. Lemma 4.2 implies that optimal knots are uniquely identified by their signatures, which is crucial for the folding algorithm presented later.

Lemma 4.2. *An optimal knot is closed.*

Proof. This follows from Lemma 4.1g and the maximality of an optimal knot. □

Next, we explain what is the motivation behind the notion of an optimal knot:

Confluences. As mentioned above, in order to allow one to eliminate a join, a knot must contain some join v together with at least one incoming sub-edge in case the knot is based on a loop and at least two sub-edges otherwise. Since g_ϕ is accessible (meaning that there do not exist any traces that cannot be extended to start from the same node), the edge must belong to some confluence α of g_ϕ. If the folding operation does not fold the entire α, then a new join is created on the border of the introduced box: one of its incoming sub-edges is labelled by the box that replaces the folded knot, another one is the last edge of one of the traces of α. Confluences are therefore the smallest subgraphs that can be folded in a meaningful way.

Uniting knots. If two different confluences α and α' share an edge, then after folding α, the resulting edge shares with α' two nodes (at least one being a target node), and thus α' contains a join of g_ϕ. To eliminate this join too, both confluences must be folded together. A similar reasoning may be repeated with knots in general. Usefulness of this rule may be illustrated by an example of the set of all singly linked lists of an unbounded length with head pointers. Without uniting, every list would generate a hierarchy of knots of the same depth as the length of the list, as illustrated in Figure 4.3a for the list of length four. This is clearly impractical since the entire set of all lists of an unbounded length could not be represented using finitely many boxes of this type. Rule 2 unites all knots into one that contains the entire list, and the set of all such knots can then be represented by a single FA (containing a loop accepting the inner nodes of the lists).

Complexity of knots. The notion of complexity is introduced to limit the effect of Rule 2 of the definition of a knot, i.e. the rule which unites knots that share a sub-edge, and to hopefully make it follow the natural hierarchical structuring of data structures. Consider, for instance, the case of singly linked lists (SLLs) of cyclic doubly linked lists (DLLs). Recall that every node in a DLL is a cut-point. In this case, it is natural to first fold the particular segments of the DLLs (denoted as doubly linked segments—DLSs—below), i.e. to introduce a box for a single pair of `next` and `prev` pointers. This way, one effectively obtains SLLs of cyclic SLLs, where the latter are over the DLS box and each contains a single cut-point at the point where the cycle connects. Subsequently, one can fold the cyclic SLLs into a higher-level box. However, uniting all knots with a common sub-edge would create knots that contain entire cyclic DLLs (requiring unboundedly many joins inside the box). The reason is that in addition to the confluences corresponding to DLSs, there are confluences which traverse the entire cyclic DLLs and that share sub-edges with all DLSs (this is in particular the case of the two circular sequences consisting solely of `next` and `prev` pointers respectively). To avoid the undesirable folding, we exploit the notion of complexity and fold graphs in successive rounds. In each round we fold all optimal knots with the smallest complexity (as described later in Section 4.1.2), which should correspond to the currently most nested, not yet folded, sub-structures. In the previous example, the algorithm starts by folding DLSs of complexity 2, because the complexity of the confluences in cyclic DLLs is given by the number of the DLSs they traverse.

Closure of knots. The closure is introduced for practical reasons. It allows one to identify optimal knots by their signatures, which is then used to simplify automata constructions that implement folding on the level of FAs (cf. Section 4.1.2).

Root of an optimal knot. The requirement for an optimal knot k to have a root is to guarantee that if an io-graph g'_ψ containing a box B representing k is accessible, then the io-graph $g'_\psi[k/B]$ emerging by substituting k for a sub-edge labelled with B is accessible, and vice versa. It is also a necessary condition for the existence of a canonical forest representation of the knot itself (since one needs to order the cut-points w.r.t. the costs of the paths leading to them from the input port of the knot).

4.1.2. Folding in the Abstraction Loop

In this section, we describe the operation of folding together with the main abstraction loop of which folding is an integral part. The pseudo-code of the main abstraction loop is shown in Algorithm 4.1. The algorithm modifies a set of FAs until it reaches a fixpoint. Folding on line 5 is a sub-procedure of the algorithm which looks for substructures of FAs that accept optimal knots, and replaces these substructures by boxes that represent the corresponding optimal knots. The operation of folding is itself composed of four consecutive steps: *Identifying indices*, *Splitting*, *Constructing boxes*, and *Applying boxes*. The steps are described in the following paragraphs.

Algorithm 4.1: Abstraction Loop

```
1 Unfold solitaire boxes;
2 repeat
3     Normalise;
4     Abstract;
5     Fold;
6 until fixpoint;
```

Unfolding of solitaire boxes. Folding is in practice applied on FAs that accept partially folded graphs (only some of the optimal knots are folded). This may lead the algorithm to hierarchically fold data structures that are not hierarchical, causing the symbolic execution not to terminate. For example, consider a program that creates a DLL of an arbitrary length. Whenever a new DLS is attached, the folding algorithm would enclose it into a box together with the tail which was folded previously. This would lead to creation of a hierarchical structure of an unbounded depth (see Figure 4.3b), which would cause the symbolic execution to never reach a fixpoint. Intuitively, this is a situation when a repetition of subgraphs may be expressed by an automaton loop that iterates a box, but it is instead misinterpreted as a recursive nesting of graphs. This situation may happen when a newly created box contains another box that cannot be iterated by a cycle in an automaton (e.g. in Figure 4.3b there is always one occurrence of a box encoding a shorter DLL fragment inside a higher-level box). This issue is addressed in the presented algorithm by first unfolding all occurrences of boxes that cannot be iterated by automata loops before folding is started.

Normalising. We define the *index* of a cut-point $u \in cps(g_\phi)$ as its position in the canonical ordering of cut-points of g_ϕ, and the *index* of a closed subgraph h of g_ϕ as the set of indices of the cut-points in $sig(h)$. The folding algorithm expects the input FA F to satisfy the property that all io-graphs of $L(F)$ have the same indices of closed knots. The reason is that folding starts by identifying the index of an optimal knot of an arbitrary io-graph from $L(F)$, and then it creates a box which accepts all closed subgraphs of the io-graphs from g_ϕ with the same index. We need a guarantee that *all* these subgraphs are indeed optimal knots. This guarantee can be achieved if the io-graphs from $L(F)$ have equivalent interconnections of cut-points, as defined below.

We define the relation $\sim_{g_\phi} \subseteq 2^{\mathbb{N}} \times 2^{\mathbb{N}}$ between indices of closed knots of g_ϕ such that $N \sim_{g_\phi} N'$ iff there is a closed knot k of g_ϕ with the index N and a closed knot k' with the index N' such that k and k' have intersecting sets of sub-edges. We say that two io-graphs g_ϕ and g'_ψ are *interconnection equivalent* iff $\sim_{g_\phi} = \sim_{g'_\psi}$ and for every two cut-points $u \in dom(g)$ and $v \in dom(g')$ with the same index, the sets of indices of cut-points that are reachable from them by straight traces are the same (note the latter requirement is more general than saying that u and v have the same indices of their spans). Notice that the relation $\sim$ is reflexive and therefore $\sim_{g_\phi} = \sim_{g'_\psi}$ implies that a knot k with the index N is in g_ϕ iff a knot k' with the same index N is in g'_ψ.

Lemma 4.3. *Given two interconnection equivalent io-graphs g_ϕ and g'_ψ, $N \subseteq \mathbb{N}$ is the index of an optimal knot in g_ϕ iff it is the index of an optimal knot in g'_ψ.*

Proof. First, we prove that for two interconnection equivalent io-graphs g_ϕ and g'_ψ, the index of a signature of a knot of complexity n of one of the io-graphs is also the index of a signature of a knot of complexity n of the other io-graph. Let I be the index of a knot k of g_ϕ of complexity n. According to the inductive definition of a knot, k can be viewed as a term t_k that consists of literals (which correspond to confluences of g of complexity at most n according to Point 1 of the definition), occurrences of the $\uplus$ binary operator (Point 2) and occurrences of the $cl \circ sig$ unary operator (Point 3), such that the weight of k constructed in this way is n. An example of such a term may be the term

$$t_k = clsig(a \uplus clsig(b \uplus (c \uplus d))) \tag{4.9}$$

where a, b, c, and d are confluences.

Due to Lemmas 4.1d and 4.1c, the signature of a knot k is the same as the signature of the knot k' such that the term $t_{k'}$ is a modification of t_k where each union is preceded by applying closure on its arguments. Note that k' really is a knot because making a closure is allowed by Point 3 of the definition, and the application of each $\uplus$ operator in $t_{k'}$ by Point 2 is still justified since closure of a knot is extensive (according to Lemma 4.1g, if two knots share a sub-edge, their closures share the same sub-edge too). For the previous example, we would obtain

$$t_{k'} = clsig(clsig(a) \uplus clsig(clsig(b) \uplus clsig(clsig(c) \uplus clsig(d)))). \tag{4.10}$$

Let k'' be a subgraph of g'_ψ such that $t_{k''}$ emerges from $t_{k'}$ by substituting each occurrence of a confluence (literal) c' in $t_{k'}$ by a knot c'' of g'_ψ with the same index J of its signature. First, we observe that k'' is indeed a knot of complexity $m \leq n$ (k'' is either a confluence with the index of its signature J, a union of two other knots with indices of their signatures being subsets of J, or the closure of a knot with the index of its signature J). Second, we refute the possibility of $m < n$ using contradiction.

Let us suppose that the statement $m < n$ holds and consider the consequences. It must then hold that c'' is built by uniting confluences C'' with numbers of cut-points smaller than n along the way. Then, because g_ϕ and g'_ψ are interconnection equivalent, it holds that there needs to exist a set of knots C' which have the same indices of signatures as the confluences in C''. However, this means that it is possible to construct c' such that its weight is at most $m < n$, which is a contradiction to the assumption that the complexity of c' is n, and therefore $m = n$.

Next, we show that if I is the index of an *optimal* knot k of complexity n in g_ϕ, then it is the index of an optimal knot of the same complexity in g'_ψ (and *vice versa*). From the above, we know that I is the index of a knot k' of complexity n in g_ϕ. We may assume that k' is closed.

For the maximality condition of an optimal knot, if k' is not a maximal knot of complexity n, then it can be united with another knot k'' to obtain a bigger knot with the same complexity. However, because g_ϕ and g'_ψ are interconnection equivalent, there must exist a knot with the same index as k'' in g_ϕ with the complexity at most n that intersects with k. This is a contradiction because k would not be maximal in this case. Therefore, k' is maximal.

To prove the existence of a source of k' reachable from the input port, we use the second point of the definition of interconnection equivalence, which says that cut-points of g_ϕ and g'_ψ with the same indices must reach by straight traces sets of cut-points with the same indices. This means that if i is the index of a source cut-point of g_ϕ that reaches all cut-points of k' and that is reachable from the input of g_ϕ by a trace that does not traverse k, then i must also be the index of a source cut-point of g'_ψ such that it reaches all cut-points of k' and is reachable from the input port of g'_ψ by a path that does not traverse k'. Both k and k' thus have a source required by the definition of an optimal knot, which is reachable by a straight trace from the i-th cut-point. This means that k' is also optimal. □

Interconnection equivalence of all io-graphs in the language of an FA F is achieved by transforming F to the *interconnection respecting form.* This form requires that the language of every TA of the FA consists of interconnection equivalent trees (when viewing root references and roots as cut-points with corresponding indices). The normalisation step also includes a transformation into the state uniform and canonicity respecting form.

Abstraction. We use abstraction described in Section 4.2 that preserves the canonicity respecting form of TAs as well as their state uniformity. It may break interconnection uniformity, in which case it is followed by another round of normalisation. Abstraction is included into each round of folding for the reason that it leads to learning more general boxes. For instance, an FA encoding a cyclic list of one particular length is first abstracted into an FA encoding a set of cyclic lists of all lengths, and the entire set is then folded into a single box.

Identifying indices. For every FA F entering this sub-procedure, we pick an arbitrary io-graph $g_\phi \in L(F)$, find all its optimal knots of the smallest possible complexity n, and extract their indices. By Lemma 4.3 and since F is normalised, indices of the optimal knots are the same for all io-graphs in $L(F)$. For every found index, the following steps fold all optimal knots with that index at once. Optimal knots of complexity n do not share sub-edges (they would be united otherwise), the order in which they are folded is therefore not important.

Splitting. For an FA $F = (\mathcal{A}_1 \cdots \mathcal{A}_n, \pi)$ and an index I of an optimal knot found in the previous step, splitting transforms F into a (set of) new FA(s) with the same language. The nodes of the borders of I-indexed optimal knots of io-graphs from $L(F)$ become roots of trees of io-forests accepted by the new FA(s). Let $s \in I$ be a position in F such that the s-indexed cut-points of io-graphs from $L(F)$ reach all the other I-indexed cut-points. The index s exists since an optimal knot has a root. Due to the definition of the closure, the border contains all I-indexed cut-points, with the possible exception of s.

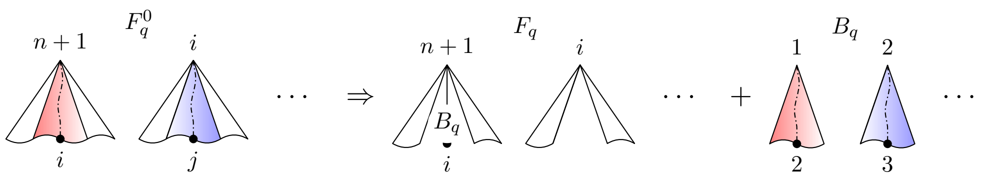


Figure 4.4.: Creation of F_q and B_q from F_q^0. The subtrees that contain references $i, j \in J$ are moved into B_q, and replaced by the B_q-labelled sub-term in F_q.

The s-th cut-point may be replaced in the border of the I-indexed optimal knot by the base e of the I-indexed confluence that is the first one reached from the s-th cut-point via a straight path. We call e the *entry*. The entry e is a root of the optimal knot, and the s-th cut-point is the only I-indexed cut-point that might be outside the knot. If e is indeed different from the s-th cut-point, then the s-th tree of forests accepted by F must be split into two TAs in the new FA: The subtree rooted at the entry is replaced by a reference to a new tree. The new tree then equals the subtree of the original s-th tree rooted at the entry.

The construction is carried out as follows. We find all states and all of their transitions that label entry nodes in accepting runs. We denote such states and transitions as entry states and transitions. For every entry state q, we create a new FA F_q^0 which is a copy of F but with the s-th TA $\mathcal{A}_s$ split to a new s-th TA $\mathcal{A}'_s$ and a new $(n+1)$-th TA $\mathcal{A}_{n+1}$. The TA $\mathcal{A}'_s$ is obtained from $\mathcal{A}_s$ by changing the entry transitions of q to accept just a reference to the new $(n+1)$-th root and by removing entry transitions of all other entry states (the entry states are processed separately in order to preserve possibly different contexts of entry nodes accepted at different states). The new TA $\mathcal{A}_{n+1}$ is a copy of $\mathcal{A}_s$ but with the only accepting state being q. Note that the construction is justified since due to state uniformity, each node that is accepted by an entry transition and that does not appear in a run below a node that is also accepted by an entry transition is an entry node. In the result, the set $J = (I \setminus \{s\}) \cup \{n+1\}$ contains the positions of the trees of forests of F_q^0 rooted at the nodes of the borders of I-indexed optimal knots.

Constructing boxes. For every F_q^0 and J being the result of splitting F according to an index I, a box B_q is constructed from F_q^0. We transform TAs of F_q^0 indexed by the elements of J. The resulting TAs will accept the original trees modified in such a way that their roots are stripped from the children that cannot reach a reference to J. To turn these TAs into an FA accepting optimal knots with the index I, it remains to order the obtained TAs and define port indices. Roughly, the input index of the box will be the position j to which we place the modified $(n+1)$-th TA of F_q^0 (the one that accepts trees rooted at the entry). The output indices are the positions of the TAs with indices $J \setminus \{j\}$ in F_q^0 which accept trees rooted at cut-points of the border of the optimal knots.

Applying boxes. This is the last step of folding. For every F_q^0, J, and B_q which are the result of splitting F according to an index I, we construct an FA F_q that accepts graphs of F where knots enclosed in B_q are substituted by a sub-edge with the label B_q. It is created from F_q^0 by (1) leaving out the parts of root transitions of its TAs that were taken into B_q, and (2) adding the sub-term $B_q(r_1, \dots, r_m)$ to the sub-terms of root transitions of the $(n+1)$-th component of F_q^0 (these are transitions used to accept the roots of the optimal knots enclosed in B_q). The states $r_1, \dots, r_m$ are fresh states that accept root references to the appropriate elements of J (to connect the borders of knots of B_q correctly to the graphs of F_q). The FA F_q now accepts graphs where optimal knots of graphs of $L(F)$ with the signature I are hidden inside B_q. Creation of B_q and of its counterpart F_q from F_q^0 is illustrated in Figure 4.4 where $i, j, \dots \in J$.

During the analysis, the discovered boxes must be stored in a database and tested for equivalence with the newly discovered ones since the alphabets of FAs would otherwise grow with every operation of folding *ad infinitum.* That is, every discovered box is given a unique name, such as "DLL" for the box from Figure 3.2b, and whenever a semantically equivalent box is folded, the newly created sub-term is labelled by that name. This step offers an opportunity for introducing another form of acceleration of the symbolic computation. Namely, when a box B is found by the procedure described above, and another box B' with a name N s.t. $[\![B']\!] \subset [\![B]\!]$ is already in the database, we associate the name N with B instead of B' and restart the analysis (i.e. start the analysis from the scratch, remembering just the updated database of boxes). If, on the other hand, $[\![B]\!] \subseteq [\![B']\!]$, the folding is performed using the name N of B', thus over-approximating the semantics of the folded FA. As presented in Section 4.3, this variant of the procedure, called *folding by inclusion*, performs in some difficult cases significantly better than the former variant, called *folding by equivalence.*

4.2. Abstraction

The abstraction we use in our analysis is based on the general techniques described in the framework of abstract regular (tree) model checking [BHRV12]. In this setting, abstraction over-approximates the language of an automaton by collapsing some of its states (i.e. merging them together, potentially introducing new loops) according to a given equivalence relation on the states of the automaton. We, in particular, build on the *finite height abstraction* of TAs, which uses the equivalence of languages of a finite height k, denoted as $\approx_k$. The equivalence is defined as $q \approx_k q'$ iff q and q' accept trees with the same sets of prefixes of the height at most k (the prefix of height k of a tree is a subgraph of the tree which contains all paths from the root of length at most k). The equivalence $\approx_k$ is further refined to deal with various features special for FAs. Namely, it has to work over tuples of TAs and cope with the interconnection of the TAs via root references, with the hierarchical structuring, and with the fact that we use a *set* of FAs instead of a single FA to represent the abstract context at a particular program location.

Refinements of $\approx_k$. First, in order to maintain the same basic shape of the heap after abstraction (such that no cut-point would e.g. suddenly appear or disappear), we refine $\approx_k$ by requiring that equivalent states must have the same spans (as defined in Section 3.1). When applied on $\approx_1$, which corresponds to equivalence of data types, this refinement provided enough precision for most of the case studies presented later on, with the exception of the most difficult ones, namely programs with skip lists [Pug90]. To verify these programs, we needed to further refine the abstraction to distinguish automata states whenever trees from their languages encode tree components containing a different number of unique paths to some root reference, but some of these paths are hidden inside boxes. In particular, two states q, q' can be equivalent only if for every io-graph g_ϕ from the graph language of the FA, for every two nodes $u, v \in \mathit{dom}(g_\phi)$ accepted by q and q', respectively, in an accepting run of the corresponding TA, the following holds: For every $w \in \mathit{cps}(g_\phi)$, both u and v have the same number of outgoing sub-edges (selectors) in $[\![g_\phi]\!]$ which start a trace in $[\![g_\phi]\!]$ leading to w. According to our experiments, this refinement does not cost almost any performance, and hence we use it by default.

Abstraction for Sets of FAs. Our analysis works with sets of FAs. We observed that abstracting individual FAs from a set of FAs in isolation is sometimes slow since in each of the FAs, the abstraction widens some selector paths only, and it takes a while until an FA in which all possible selector paths are widened is obtained. For instance, when analysing a program that creates binary trees, the symbolic analysis generates many FAs before reaching a fixpoint, each of the FAs accepting a subset of binary trees with some of the branches restricted to a bounded length (e.g. trees with no right branches, trees with a single right branch of length one, two, etc.). In such cases, it helps when the abstraction has an opportunity to combine information from several FAs. For instance, consider an FA that encodes binary trees degenerated to an arbitrarily long left branch, and another FA that encodes trees degenerated to right branches only. Abstracting these FAs in isolation has no effect. However, if the abstraction is allowed to collapse states from both of these FAs, it can generate an FA accepting all possible branches.

Unfortunately, the natural solution to achieve the above, which is to unite FAs before abstraction, introduces a much too coarse over-approximation, even before the abstraction itself is applied. Instead, we enrich the automata structure of an FA F by TA states and transitions of another one, omitting introduction of new root states. While this does not change the language of F, it allows the abstraction to combine the information from both FAs. In particular, before abstracting an FA $F = (\mathcal{A}_1 \cdots \mathcal{A}_n, \pi)$ from a set S of FAs, we pre-process it as follows.

(1) We pick FAs $F' = (\mathcal{A}'_1 \cdots \mathcal{A}'_n, \pi) \in S$ that are compatible with F in that they have the same number of TAs, the same port references, and for each $1 \leq i \leq n$, the root states of $\mathcal{A}'_i$ have the same spans as the root states of $\mathcal{A}_i$.

(2) For all such F' and each $1 \leq i \leq n$, we add transitions and states of $\mathcal{A}'_i$ to $\mathcal{A}_i$, but we keep the original set of root states of $\mathcal{A}_i$. Since we assume that the sets of states of TAs of different FAs are disjoint, the language of $\mathcal{A}_i$ stays the same, but its structure is enriched, which helps the abstraction to perform a coarser widening.

4.3. Experimental Results

We have implemented the above proposed techniques as an extension of the Forester tool and tested their generality and efficiency on a number of case studies. In the experiments, we compare two configurations of Forester, and we also compare the results of Forester with those of Predator [DPV13], which uses a graph-based memory representation inspired by separation logic with higher-order list predicates. We do not provide a comparison with Space Invader [YLB$^+$08] and SLAyer [BCI11], based also on separation logic with higher-order list predicates, since in our experiments they were always outperformed by Predator.

In the experiments, we considered programs with various types of lists (singly and doubly linked, cyclic, nested, with skip pointers), trees, and their combinations. In the case of skip lists, we had to slightly modify the algorithms since their original versions use an ordering on the data stored in the nodes of the lists. This is done for the reason to guarantee that the search window delimited on some level of skip pointers is not left on any lower level of the skip pointers. In our modification, we avoided such a scenario by adding an additional explicit end-of-window pointer. In Chapter 5, we describe an extension of the analysis that takes into consideration also the data fields in the nodes and the additional pointer is not necessary. We checked the programs for memory safety only, i.e. we did not check other properties (such as that the result of a sorting procedure is indeed a sorted permutation of the original).

Table 4.1 gives running times in seconds (the average of 10 executions) of the tools on our case studies. “Basic” stands for Forester with the abstraction applied on individual FAs only and “SFA” stands for Forester with the abstraction for sets of FAs. The value TIMEOUT means that the running time of the tool exceeded 30 minutes, and the value ERROR means that the tool reported a spurious error. The names of the examples in the table contain the name of the data structure manipulated in the program, which is “SLL” for singly linked lists, “DLL” for doubly linked lists (the “C” prefix denotes cyclic lists), “tree” for binary trees, “tree+parents” for binary trees with parent pointers. Nested variants of SLL (DLL) are named as “SLL (DLL) of” and the type of the nested structure. In particular, “SLL of 0/1 SLLs” stands for SLL of a nested SLL of length 0 or 1, and “SLL of 2CDLLs” stands for SLL whose each node is a root of two CDLLs. The “+head” flag stands for a list where each element points to the head of the list and the subscript “Linux” denotes the implementation of lists used in the Linux kernel, which uses type casts and a restricted pointer arithmetic. The “DLL+subdata” stands for a kind of a DLL with data pointers pointing either inside the list nodes or optionally outside of them. For a “skip list”, the subscript denotes the number of skip pointers. In the example “tree+stack”, a randomly constructed tree is deleted using a stack, and “DSW” stands for the Deutsch-Schorr-Waite tree traversal (the Lindstrom variant). All experiments start with a random creation and end with a disposal of the specified structure; the indicated procedure (if any) is performed in between. The experiments were run on a machine with the Intel i7-2600 (3.40 GHz) CPU and 16 GiB of RAM.

The table further contains the column “boxes” where the value “X/Y” means that X manually created boxes were provided to the analysis that did not use learning while

Table 4.1.: Results of the experiments

Example	basic	SFA	boxes	Predator
SLL (delete)	0.03	0.04		0.04
SLL (bubblesort)	0.04	0.04		0.03
SLL (mergesort)	0.08	0.15		0.10
SLL (insertsort)	0.05	0.05		0.04
SLL (reverse)	0.03	0.03		0.03
SLL+head	0.05	0.05		0.03
SLL of 0/1 SLLs	0.03	0.03		0.11
$\text{SLL}_{\text{Linux}}$	0.03	0.03		0.03
SLL of CSLLs	2.07	0.73	3 / 4	0.12
DLL (reverse)	0.04	0.06	1 / 1	0.03
DLL (insert)	0.06	0.07	1 / 1	0.05
DLL (insertsort1)	0.35	0.40	1 / 1	0.11
DLL (insertsort2)	0.11	0.12	1 / 1	0.05
DLL of CDLLs	5.67	1.25	8 / 7	0.22
DLL+subdata	0.06	0.09	- / 2	TIMEOUT
CDLL	0.03	0.03	1 / 1	0.03
SLL of $\text{2CDLLs}_{\text{Linux}}$	0.16	0.17	13 / 5	0.25
skip list_2	0.66	0.42	- / 3	TIMEOUT
skip list_3	T	9.14	- / 7	TIMEOUT
tree	0.14	0.14		ERROR
tree+parents	0.18	0.21	2 / 2	TIMEOUT
tree+stack	0.09	0.08		ERROR
tree (DSW)	1.74	0.40		ERROR
tree of CSLLs	0.32	0.42	- / 4	ERROR

Y boxes were learnt when the box learning procedure was enabled. The value "-" of X means that we did not run the given example with manually constructed boxes since their construction was too tedious. If user-defined boxes are given to Forester in advance, the speedup is in most cases negligible, with the exception of "DLL of CDLLs" and "SLL of CSLLs", where it is up to 7 times. In a majority of cases, the learnt boxes were the same as the ones created manually. In some cases, such as "SLL of $\text{2CDLLs}_{\text{Linux}}$", the learning algorithm found a smaller set of more elaborate boxes than those provided manually.

In the experiments, we use folding by inclusion as defined in Section 4.1.2. For simpler cases, the performance matched the performance of folding by equivalence, but for the more difficult examples it was considerably faster (such as for "skip list_2" when the time decreased from 3.82 s to 0.66 s), and only when it was used the analysis of "skip list_3" succeeded. Further, the implementation folds optimal knots of the complexity ≤ 2, which is enough for the considered examples. Finally, note that the performance of Forester in the considered experiments is indeed comparable with that of Predator even though Forester can handle much more general data structures.

4.4. Conclusion

This chapter presented an extension of the shape analysis of [HHR$^+$12], presented in Chapter 3. Unlike the original analysis, the extension works fully automatically, without the need of the user to provide any information. For that purpose, we have proposed a technique of automatically learning FAs called boxes to be used as alphabet symbols in higher-level FAs when describing sets of complex heap graphs. We also proposed a way how to efficiently integrate the learning with the main analysis algorithm. Finally, we have proposed a significant improvement—both in terms of generality as well as efficiency—of the abstraction used in the original framework.

An implementation of the approach presented in this chapter inside the Forester tool allowed us to fully-automatically handle programs over quite complex heap structures, including a data-independent modification of 2-level and 3-level skip lists, which—to the best of our knowledge—we were the first to fully automatically verify (the recent work of [LGQC14], based on second-order bi-abduction in separation logic, is the only other approach we are aware of that also succeeded). At the same time, the efficiency of the analysis is comparable with other state-of-the-art analysers even though they handle less general classes of heap structures. In the next chapter, we introduce yet another extension of the forest automata-based shape analysis, an extension that takes into consideration relations among data values stored inside memory cells and, therefore, allows verification of data structures that depend on data.

5. Forest Automata-Based Shape Analysis of Programs with Data

In the previous two chapters, we focused on shape analysis that used only pointer fields of data structures and abstracted from non-pointer ones. This is in many cases sufficient, however, there are also cases where it is necessary to track the data that are stored in data structures to be able to correctly verify their higher-level and shape invariants, or even memory safety. Let us now give two examples of such data structures.

First, consider a binary search tree (BST). One of the higher-level invariant of a BST is that for every node u, the data values of all nodes in the left subtree of u are less than the data value stored in u, and the data values of all nodes in the right subtree of u are greater than the data value of u. The procedure that inserts a new data value `d` into a BST (given in Figure 5.1) uses the variable `x` to descend the BST and find the position at which the node `newNode` with the new data value `d` should be inserted. The procedure uses the relation between the new data value and the root of the tree to determine whether `d` will be stored in the left or the right subtree (or not inserted at all in the case it is equal to the data value of the root). Failure to track this relation may cause the analysis to report a spurious counterexample for some operation that relies on the higher-level invariant.

Second, the routines for manipulating a skip list [Pug90] rely on the property that the data values of lists on all levels are always sorted. Consider, for example, that we are inserting the value 7 into the 2-level skip list in Figure 5.2. The procedure for inserting starts in the node labelled as `head` and first tries to find the insertion point at level 2 (i.e. the level that uses the $\mathtt{n}_2$ pointers) by testing whether the inserted value is greater than the value in $\mathtt{head}{\rightarrow}\mathtt{n}_2$, the successor of `head`. In this case, it is not, so the procedure descends to level 1 and, because this is the ground level, traverses the list over the $\mathtt{n}_1$ pointers and finds the exact position where the new node will be inserted. Because of the sortedness property, we know that the new node cannot be inserted anywhere behind the node $\mathtt{head}{\rightarrow}\mathtt{n}_2$. Note that in this case, not only the sortedness property would be lost when not treating the data, but even the shape invariant would be corrupted!

Automated verification techniques that aim for the verification of such data structures need to handle *both* infinite sets of reachable heap configurations that have a form of complex graphs *and* the different possible relationships between data values embedded in such graphs. The few approaches that can automatically reason about data properties are often limited to specific classes of structures, mostly singly linked lists (SLLs), and/or are not fully automated (as also discussed later).

In this chapter, we propose an extension of the forest automata-based shape analysis that was described in Chapter 3 and further augmented in Chapter 4. Our extension

```
Node *insert(Node *root, Data d)
{
  Node* newNode = calloc(sizeof(Node));
  newNode→data = d;
  if (root == NULL) return newNode;
  Node *x = root;
  while (x→data != newNode→data)
  {
    if (x→data < newNode→data)
      if (x→right ≠ NULL) x = x→right;
      else {
        x→right = newNode;
        break;
      }
    else
      if (x→left ≠ NULL) x = x→left;
      else {
        x→left = newNode;
        break;
      }
  }
  if (x→data == newNode→data) free(newNode);
  x = NULL;
  return root;
}
```

Figure 5.1.: A function that inserts a new node into a BST and returns a pointer to its root node

allows us to represent relationships between data elements stored inside heap structures. As a consequence, this method makes it possible to automatically verify programs that depend on relationships between data, such as programs manipulating various search trees, lists, and skip lists, and to also verify e.g. different sorting algorithms. Technically, we express relationships between data elements associated with nodes of the heap graph by two classes of constraints. *Local data constraints* are associated with transitions of tree automata and capture relationships between data of neighbouring nodes in a tree of the forest decomposition of a heap graph; they can be used e.g. to represent ordering internal to some structure such as a binary search tree. *Global data constraints* are associated with states of TAs (even states of *different* TAs) and capture relationships between data in distant parts of the heap. In order to obtain a powerful analysis based on such extended forest automata, the entire analysis machinery must be also be extended, including a need to develop mechanisms for propagating data constraints through FAs, to adapt the abstraction mechanisms of abstract regular tree model checking (ARTMC), to develop a new inclusion check between extended FAs, and to define extended abstract transformers.

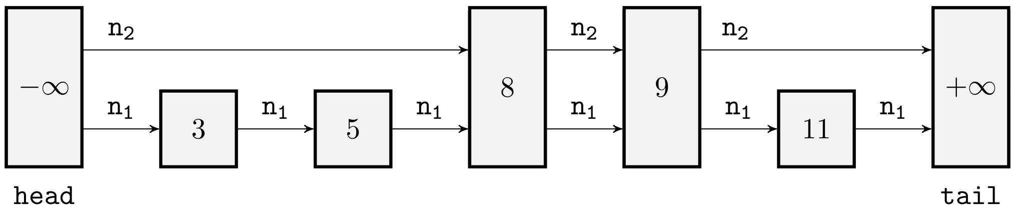


Figure 5.2.: An example of a 2-level skip list

The presented approach has been implemented as a further extension of the Forester tool. We have applied the tool to verification of data properties, notably sortedness, of sequential programs with data structures, like various forms of singly and doubly linked lists (DLLs), possibly cyclic or shared, binary search trees (BSTs), and even 2-level and 3-level skip lists. The verified programs include operations like insertion, deletion, or reversal, and also bubble-sort and insert-sort both on SLLs and DLLs. The experiments confirm that our approach is not only fully automated and rather general, but also quite efficient, outperforming many previously known approaches even though they are not of the same level of automation or generality. In the case of skip lists, our analysis is the first fully-automated shape analysis which is able to handle fully-fledged skip lists[1].

Related Work. Verification of properties depending on the ordering of data stored in SLLs was considered in [BBH+11], which translates programs with SLLs to counter automata. A subsequent analysis of these automata allows one to prove memory safety, sortedness, and termination for the original programs. The work is, however, strongly limited to SLLs. In the work presnted in this chapter, we get inspired by the way that [BBH+11] uses for dealing with ordering relations on data, but we significantly redesign it to be able to track not only ordering between simple list segments but rather general heap shapes described by FAs. In order to achieve this, in addition to proposing a suitable way of combining ordering relations with FAs, we also had to significantly modify many of the operations used over FAs.

In [AACJ09], another approach for verifying data-dependent properties of programs with lists was proposed. However, even this approach is strongly limited to SLLs, and it is also much less efficient than our current approach. In [AHH+13], concurrent programs operating on SLLs are analysed using an adaptation of the transitive closure logic (see e.g. [BR06]), which also tracks simple sortedness properties between data elements.

Verification of properties of programs depending on the data stored in dynamic linked data structures was considered in the context of the TVLA tool [LRS05] as well. Unlike our approach, [LRS05] assumes a fixed set of shape predicates and uses inductive logic programming to learn predicates needed for tracking non-pointer data. The experiments presented in [LRS05] involve verification of sorting and stability properties of several programs on SLLs (merging, reversal, bubble-sort, insert-sort) as well as insertion and deletion in BSTs. We do not handle stability, but for the other properties, our approach

[1] Note that in the experiments presented in Chapter 4, where we ignored the data stored in the nodes, we had to modify the insertion procedure for a skip list by introducing an explicit *end-of-window* pointer for every level of the skip list, so that the shape invariant did not depend on ordering relations.

is much faster. Moreover, for BSTs, we verify that a node is greater/smaller than all the nodes in its left/right subtrees (not just than the immediate successors as in [LRS05]). Another work that combines the TVLA framework with reasoning on data is [BHT06], which combines TVLA with predicate abstraction implemented in BLAST. The approach was experimentally run on several list-manipulating programs only.

An approach based on separation logic extended with constraints on the data stored inside dynamic linked data structures and capable of handling size, ordering, as well as bag properties was presented in [CDNQ12b]. Using the approach, various programs with SLLs, DLLs, and also AVL trees and red-black trees were verified. The approach, however, requires the user to manually provide inductive shape predicates as well as loop invariants. Later, the need to provide loop invariants was avoided in [QHL$^+$13], but the need to manually provide inductive shape predicates remains.

The work considered in [CR08] extends the previous work [CRN07] with data constraints. The method still needs shape invariants extended with data to be provided manually. The join and widening operations used on the shape level are extended with subsequent join and widening on the data level to cope with the data during the analysis.

Another work that targets verification of programs with dynamic linked data structures, including properties depending on the data stored in them, is [ZKR08]. It generates verification conditions in an undecidable fragment of higher-order logic and discharges them using decision procedures, first-order theorem proving, and interactive theorem proving. To generate the verification conditions, loop invariants are needed. These can either be provided manually, or sometimes synthesised semi-automatically using the approach of [WKZ$^+$07]. The latter approach was successfully applied to several programs with SLLs, DLLs, trees, trees with parent pointers, and 2-level skip lists. However, for some of them, the user still had to provide some of the needed abstraction predicates. A further extension of this approach given in [WP10] increases the degree of automation and synthesises the loop invariants automatically using counterexample guided refinement.

Several works, including [BDES12], define frameworks for reasoning about pre- and post-conditions of programs with SLLs and data. Decidable fragments that can express more complex properties on data than we consider are identified, but the approach does not perform a fully automated verification, only checking of pre-post condition pairs. Other approaches presenting various logical fragments for reasoning about heaps and the data stored in them together with decision procedures of these fragments were presented e.g. in [MN05, RBHC07, CLQR07, LQ08]. None of these approaches has been extended to a fully automatic verification method.

Outline. In Section 5.1, we present our extension to the forest automata formalism that uses constraints to specify relationships between data. values. Then, in Section 5.2, we describe the changes we made to the shape analysis algorithm that allow it to handle programs that depend on ordered data. Section 5.3 shows how our procedure handles boxes, used in order to allow processing of more complex data structures. Section 5.4 describes our implementation of the proposed ideas as well as the obtained experimental results and Section 5.5 concludes the chapter.

5.1. Forest Automata with Data Constraints

This section presents several extensions to the basic definitions presented in Chapters 2 and 3 that will be used throughout this chapter.

Graphs and Forests with Data. Let us fix a data domain $\mathbb{D}$ with a total order $\preceq$ (in the following, we also use the symbols $\prec, \succ, \succeq$, and $=$ with the obvious meaning). We extend the notion of a *graph* $g : V \to \Sigma \times V^*$ with a *data labelling* λ_g that assigns every node a value, formally, $\lambda_g : V \to (\mathbb{D} \cup \{\top\})$ where $\top \notin \mathbb{D}$ is interpreted as an undefined value. Note that data labellings extend to *trees* because a tree is only a special case of a graph, in particular a graph with a single root. For the case of *forests*, though, we need to make sure that the data labelling of the root references are consistent with the data labelling of the respective roots.

We say that a forest $t_1 \cdots t_n$ is *composable* if $\lambda_{t_k}(u) = \lambda_{t_j}(root(t_j))$ where $\ell_{t_k}(u) = \overline{j}$ for any root reference u in any tree t_k of the forest. As a consequence, the operator $\otimes$ that composes forests into graphs is defined only for composable forests. The data labelling $\lambda_{g'}$ of the resulting graph $g' = \otimes t_1 \cdots t_n$ is then obtained simply as the union of data labellings of all trees from $t_1 \cdots t_n$, restricted to the domain of g'. We will use the following notation to talk about relations of data values of nodes within a forest. Given nodes u and v of trees t and t' of a forest respectively, and a relation $\sim \in \{\prec, \preceq, =, \succ, \succeq\}$, we denote by $u \sim_{\mathtt{rr}} v$ that $\lambda_t(u) \sim \lambda_{t'}(v)$ and we denote by $u \sim_{\mathtt{ra}} v$ that $\lambda_t(u) \sim \lambda_{t'}(w)$ for all non-root-reference nodes w in the subtree of t' rooted at v, including the node v itself. We call these two types of relationships ***root-root*** and ***root-all*** relations respectively. The definition of *io-graphs* and *io-forests* with data is a straightforward extension obtained by adding data labellings to the corresponding concepts introduced in Section 3.1.

Tree Automata with Data Constraints. For the use in this chapter, we also extend the notion of tree and forest automata to consider data. Because we focus on the verification of programs that work with *ordered* data, when we represent sets of heap graphs with forest automata, we do not remember *exact* values stored in nodes of heap graphs, but only the relations among them instead. Let us first start with the modified definition of a tree automaton.

A tree automaton with data constraints (or simply a tree automaton, TA) is a tuple $\mathcal{A} = (Q, \Sigma, \Delta_c, R = \{q_0\})$ where Q is a nonempty finite set of *states*, Σ is a ranked alphabet, $R \subseteq Q$ is a *singleton set of root states* containing the root state q_0 (we use $root(\mathcal{A})$ to denote the root state of $\mathcal{A}$), and Δ_c is a set of (constrained) *transitions*. Each transition is of the form $q \to a(q_1, \ldots, q_n) : c$ where $n \geq 0$, $q, q_1, \ldots, q_n \in Q$, $a \in \Sigma$, and c is a set of *local constraints*. Every local constraint is of the form $0 \sim_{\mathtt{rr}} i$ or $0 \sim_{\mathtt{ra}} i$ where $\sim \in \{\prec, \preceq, \succ, \succeq, =\}$ (with $=$ viewed as syntactic sugar for a pair of constraints that use $\preceq$ and $\succeq$) and $1 \leq i \leq n$.

Intuitively, a local constraint of the form $0 \sim_{\mathtt{rr}} i$ associated with a transition of $\mathcal{A}$ of the form $q \to a(q_1, \ldots, q_n)$ states the following: For each tree t' accepted by $\mathcal{A}$, the data value of the *root* of the subtree t of t' that is accepted at state q is related by $\sim$ with the data value of the *root* of the i-th subtree of t accepted at state q_i. A local constraint of the form $0 \sim_{\texttt{ra}} i$ states that, in addition to the constraint imposed by $0 \sim_{\texttt{rr}} i$, the relation $\sim$ also holds between q and all nodes in the i-th subtree of t.

Moreover, we also extend the notion of a *run* of $\mathcal{A}$. In the data setting, tree automata accept trees with data. A *run* of $\mathcal{A}$ over a tree t with data labelling λ_t is a mapping $\rho : dom(t) \to Q$ such that

1. the root of t is mapped to the root state of $\mathcal{A}$, $\rho(root(t)) = q_0$ (a simplification that considers only *accepting* runs),
2. for each node $v \in dom(t)$ where $q = \rho(v)$, if $q_i = \rho(S(v)_i)$ for $1 \leq i \leq |S(v)|$, then Δ_c has a transition $q \to \ell(v)(q_1, \ldots, q_{|S(v)|}) : c$ (the definition of a run of a TA from Section 2.2), and
3. for each constraint $0 \sim_{\texttt{r}x} i$ in c where $x \in \{\texttt{r}, \texttt{a}\}$, it holds that $v \sim_{\texttt{r}x} S(v)_i$ (consistency of data constraints).

Note that for the sake of simplification, all runs start from the root state. We define the *language* of $\mathcal{A}$ as $L(\mathcal{A}) = \{t \mid \text{there is a run of } \mathcal{A} \text{ over } t\}$.

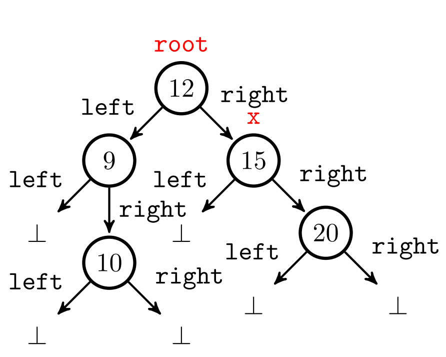
Example 5.1. *BSTs, such as the tree labelled by* `root` *but without the variable* `x` *in Figure 5.3a, are accepted by the TA* $\mathcal{A} = (\{q_1, q_\perp\}, \Sigma, \Delta_c, \{q_1\})$ *(we use* $\underline{q_1}$ *to denote that* q_1 *is a root state), where* Δ_c *contains the following transitions (we ignore the data selector in the TA symbols):*

$$
\begin{array}{llll}
\underline{q_1} \to \texttt{left}, \texttt{right}(q_1, q_1) & : 0 \succ_{\texttt{ra}} 1, 0 \prec_{\texttt{ra}} 2 & \underline{q_1} \to \texttt{left}, \texttt{right}(q_1, q_\perp) & : 0 \succ_{\texttt{ra}} 1 \\
\underline{q_1} \to \texttt{left}, \texttt{right}(q_\perp, q_1) & : 0 \prec_{\texttt{ra}} 2 & \underline{q_1} \to \texttt{left}, \texttt{right}(q_\perp, q_\perp) & \\
q_\perp \to \perp() & & &
\end{array}
$$

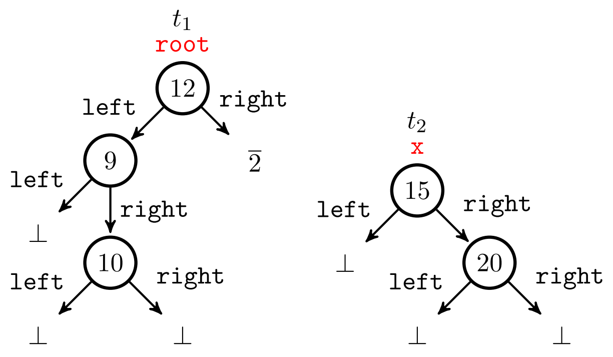
The local constraints of the transitions express that the data value in a node is always greater than the data values of all nodes in its left subtree and less than the data values of all nodes in its right subtree. □

Forest Automata with Data Constraints. We also extend FAs with data constraints. A *forest automaton with data constraints* (in this chapter, we will simply say a forest automaton, FA) over Σ is a triple of the form $F = (\mathcal{A}_1 \cdots \mathcal{A}_n, \pi, \varphi)$ where:

- $\mathcal{A}_1 \cdots \mathcal{A}_n$, with $n \geq 0$, is a sequence of TAs over the alphabet $\Sigma \cup \{1, \ldots, n\}$ whose sets of states $Q_1, \ldots, Q_n$ are pairwise disjoint,
- π is a sequence of port indices as defined in Section 3.1, and
- φ is a set of *global data constraints* between the states of $\mathcal{A}_1 \cdots \mathcal{A}_n$, each having the form $q \sim_{\texttt{rr}} q'$ or $q \sim_{\texttt{ra}} q'$ where $q, q' \in \bigcup_{i=1}^{n} Q_i$, at least one of q, q' is a root state and $\sim \in \{\prec, \preceq, \succ, \succeq, =\}$ (with $=$ again viewed as syntactic sugar). Intuitively, $q \sim_{\texttt{r}x} q'$ says that for any two nodes v and v' in a forest that are labelled in accepting runs of TAs by q and q' respectively, the data relation $v \sim_{\texttt{r}x} v'$ must hold.



(a) A graph



(b) A forest decomposition

Figure 5.3.: Decomposition of a graph into trees

An io-forest $(t_1 \cdots t_n, \pi')$ with data is *accepted* by F if there are runs $\rho_1, \ldots, \rho_n$ such that ρ_i is a run of $\mathcal{A}_i$ over t_i for every $1 \leq i \leq n$, the port indices match, $\pi' = \pi$, and for each global constraint of the form $q \sim_{\mathtt{r}x} q'$ where $x \in \{\mathtt{r}, \mathtt{a}\}$, q is a state of some $\mathcal{A}_i$ and q' is a state of some $\mathcal{A}_j$, we have $v \sim_{\mathtt{r}x} v'$ whenever $\rho_i(v) = q$ and $\rho_j(v') = q'$. The *forest language* of F, denoted as $L_f(F)$, is the set of io-forests accepted by F, and its *graph language* is the set of io-graphs $L(F)$ obtained by applying $\otimes$ on composable io-forests accepted by F[2].

Note that global constraints can imply some local ones, but they cannot in general be replaced by local constraints only. Indeed, global constraints can relate states of different automata as well as states that do not appear in a single transition and therefore relate nodes that can be arbitrarily far from each other and unrelated by any sequence of local constraints.

5.2. FA-based Shape Analysis with Data

The extension in this chapter uses the analysis described in Section 3.2 with several modifications. First, we consider a single data selector, i.e. $DSel = \{\mathtt{data}\}$. The data labelling of heaps is based on this selector in such a way that for a node v from a heap $g_{\mathtt{sf}}$, its data value $\lambda_{g_{\mathtt{sf}}}(v)$ is set to the value of the `data` selector (or $\top$ if undefined).

For the sake of brevity of the used examples, in this chapter we will represent program states using the so-called *abstract configurations*. Each abstract configuration is a pair $\langle \sigma, F \rangle$ where σ maps every variable to $\perp$, an index of a TA in F, or to an undefined value, and F is an FA representing a set of heaps (any such configuration can be easily transformed into the form used in Section 3.2 by creating a *stack frame* node in F that encodes σ). We do not write port indices in FAs from abstract configurations. Further in our examples, we will not write the often used transition $q_\perp \to \perp()$ that we consider implicitly present in all sets of transitions of TAs (in figures, we simplify the state to $\perp$).

[2] Note that from the definitions of languages of TAs and FAs, the effect of the $\sim_{\mathtt{ra}}$ data constraint (both local and global) is local to the TAs it is related to.

$$
\begin{array}{l}
F = (\mathcal{A}_1\,\mathcal{A}_2, \varphi) \\
\sigma(\texttt{root}) = 1, \sigma(\texttt{x}) = 2 \\
\mathcal{A}_1 : \left\{ \begin{array}{ll}
\underline{q_{\texttt{r}}} \rightarrow \texttt{left},\texttt{right}(q_1, \overline{2}) & : 0 \succ_{\texttt{ra}} 1, 0 \prec_{\texttt{ra}} 2 \\
q_1 \rightarrow \texttt{left},\texttt{right}(q_\perp, q_2) & : 0 \prec_{\texttt{ra}} 2 \\
q_2 \rightarrow \texttt{left},\texttt{right}(q_\perp, q_\perp) &
\end{array} \right. \\
\mathcal{A}_2 : \left\{ \begin{array}{ll}
\underline{q_{\texttt{x}}} \rightarrow \texttt{left},\texttt{right}(q_\perp, q_3) & : 0 \prec_{\texttt{ra}} 2 \\
q_3 \rightarrow \texttt{left},\texttt{right}(q_\perp, q_\perp) &
\end{array} \right. \\
\varphi = \left\{ \begin{array}{l}
q_{\texttt{x}} \succ_{\texttt{ra}} q_{\texttt{r}}, q_3 \succ_{\texttt{ra}} q_{\texttt{r}}, \\
q_{\texttt{r}} \succ_{\texttt{ra}} q_{\texttt{x}}, q_1 \prec_{\texttt{ra}} q_{\texttt{x}}, q_2 \prec_{\texttt{ra}} q_{\texttt{x}}
\end{array} \right\}
\end{array}
$$

Figure 5.4.: An example of an abstract configuration that is a possible representation of the concrete configuration shown in Figure 5.3b

Example 5.2. *Figure 5.3a shows a possible heap of the program in Figure 5.1. Nodes are shown as circles, labelled by their data values. Selectors are shown as edges. Each selector points either to a node or to $\perp$ (denoting `NULL`). Some nodes are labelled by a pointer variable that points to them. The node with data value* 15 *is a cut-point since it is referenced by variable* `x`*. Figure 5.3b shows a tree decomposition of the graph into two trees, one rooted at the node referenced by* `root`*, and the other rooted at the node pointed by* `x`*. The* `right` *selector of the root node in the first tree points to root reference* $\overline{2}$ *($\overline{i}$ denotes a reference to the i-th tree t_i) to indicate that in the graph, it points to the corresponding cut-point.* □

Example 5.3. *Figure 5.4 illustrates an abstract configuration $\langle \sigma, F \rangle$ that is a possible representation of the concrete configuration shown in Figure 5.3b.* □

The symbolic execution from Section 3.2 is modified for considering the data relations in the following way. Some of the considered operations require the so-called *constraint saturation.* The saturation procedure transforms the FAs into the saturated form, meaning that they explicitly include all (local and global) data constraints that are consequences of the existing ones.

The automata abstraction used in widening is modified by also taking into account the data relations. In particular, for a pair of states q and q' that are to be merged according to the abstraction procedure from Section 4.2, we further impose the requirement that they occur in isomorphic global data constraints. This requirement means that $q \sim_{\texttt{r}x} p$ occurs as a global constraint if and only if $q' \sim_{\texttt{r}x} p$ occurs as a global constraint, for any p and x, and it guarantees that the abstraction does indeed *over*-approximate (if we merge a pair of states with incompatible constraints, the language of the TA may become empty).

In the following subsections, we provide more detail on some of the major steps of our analysis. Section 5.2.1 describes the constraint saturation procedure, Section 5.2.2 describes the modifications made to abstract transformers, Section 5.2.3 describes the changes in the normalisation, and, finally, Section 5.2.4 describes our modified check for inclusion.

Table 5.1.: Rules for inferring global constraints from global constraints.

$$\frac{q \sim_{\mathtt{rr}} q' \qquad q' \sim'_{\mathtt{r}x} q''}{q\,(\sim \circ \sim')_{\mathtt{r}x}\, q''}\ \text{G-T}\textsc{rans}$$

$$\frac{}{q \simeq_{\mathtt{rr}} q}\ \text{G-R}\textsc{efl} \qquad\qquad \frac{q' \sim_{\mathtt{rr}} q}{q \sim^{-1}_{\mathtt{rr}} q'}\ \text{G-D}\textsc{ual}$$

$$\frac{q \sim_{\mathtt{rr}} q' \qquad \mathtt{Leaf}(q')}{q \sim_{\mathtt{ra}} q'}\ \text{G-S}\textsc{tre}$$

$$\frac{q \sim_{\mathtt{ra}} q'}{q \sim_{\mathtt{rr}} q'}\ \text{G-W}\textsc{eak}1 \qquad\qquad \frac{q \sim_{\mathtt{r}x} q'}{q \simeq_{\mathtt{r}x} q'}\ \text{G-W}\textsc{eak}2$$

$$\frac{\mathit{root}(\mathcal{A}) \sim_{\mathtt{ra}} \mathit{root}(\mathcal{A}') \qquad q' \in Q(\mathcal{A}')}{\mathit{root}(\mathcal{A}) \sim_{\mathtt{ra}} q'}\ \text{G-R}\textsc{oot}\text{A}\textsc{ll}$$

- We assume that $x \in \{r, a\}$,
- $\simeq \in \{\preceq, \succeq\}$,
- $\mathtt{Leaf}(q')$ means that q' has only nullary outgoing transitions, and
- $Q(\mathcal{A}')$ is the set of states of the TA $\mathcal{A}'$.

5.2.1. Constraint Saturation

In this section, we show the saturation rules that are used to deduce new data constraints from already existing ones. The saturation rules are used in a fixpoint computation to deduce both global and local constraints from global constraints, local constraints, or their combinations.

Before the description of the saturation rules, we first introduce some notation. For relations $\sim$ and $\sim'$ on $\mathbb{D}$, let $\sim \circ \sim'$ be the weakest relation from $\{\prec_{\mathtt{r}x}, \preceq_{\mathtt{r}x}, \succ_{\mathtt{r}x}, \succeq_{\mathtt{r}x}\}$, for $x \in \{\mathtt{r}, \mathtt{a}\}$, such that for all $d_1, d_2, d_3 \in \mathbb{D}$, it holds that $d_1 \sim d_2 \wedge d_2 \sim' d_3 \implies d_1 \,(\sim \circ \sim')\, d_3$. We write $\sim \subseteq \sim'$ iff $d \sim d'$ implies $d \sim' d'$, and we define $\sim^{-1}$ by $d \sim^{-1} d'$ iff $d' \sim d$. We say that a constraint $q \sim'_{\mathtt{r}y} q'$ is a *weakening* of a constraint $q \sim_{\mathtt{r}x} q'$ iff it holds that $\sim \subseteq \sim'$ and, in the case y is $\mathtt{a}$ (i.e. a root-all constraint), it also holds that x is $\mathtt{a}$. The saturation rules that can be used are as follows.

Inferring global constraints from global constraints

The saturation rules for inferring new global constraints from already existing ones, as shown in Table 5.1, are based on the following principles:

Table 5.2.: Rules for inferring local constraints from local constraints.

$$\frac{0 \sim_{\texttt{ra}} i \in c}{0 \sim_{\texttt{rr}} i \in c}\ \textsc{L-RootRoot}$$

$$\frac{0 \sim_{\texttt{r}x} i \in c}{0 \simeq_{\texttt{r}x} i \in c}\ \textsc{L-Weak} \qquad \frac{0 \sim_{\texttt{rr}} i \in c \qquad \texttt{Leaf}(q_i)}{0 \sim_{\texttt{ra}} i \in c}\ \textsc{L-Stre}$$

- We assume the transition $q \to a(q_1, \ldots, q_n) : c$ and $1 \leq i \leq n$,
- $x \in \{r, a\}$,
- $\simeq \in \{\preceq, \succeq\}$, and
- $\texttt{Leaf}(q)$ is true iff q has only nullary outgoing transitions.

1. properties of the ordering relations:
   - G-Trans is based on transitivity,
   - G-Refl is based on the reflexivity of $\preceq$ and $\succeq$, and
   - G-Dual is based on the duality of $\prec$ and $\succ$.
2. strengthening of existing data constraints:
   - G-Stre states that each global constraint $q \sim_{\texttt{rr}} q'$ where q' has nullary outgoing transitions only can be strengthened to $q \prec_{\texttt{ra}} q'$,
3. weakening of existing data constraints:
   - G-Weak1 states that from $q \sim_{\texttt{ra}} q'$, we can infer a weaker constraint $q \sim_{\texttt{rr}} q'$,
   - G-Weak2 gives a rule for inferring the weaker constraints $q \preceq_{\texttt{r}x} q'$ from $q \prec_{\texttt{r}x} q'$ and $q \succeq_{\texttt{r}x} q'$ from $q \succ_{\texttt{r}x} q'$ for any $x \in \{\texttt{r}, \texttt{a}\}$,
4. properties of the ra relation:
   - G-RootAll states for a pair of TAs $\mathcal{A}$ and $\mathcal{A}'$ of the given FA that if q' is a state of $\mathcal{A}'$, then a global constraint $\texttt{root}(\mathcal{A}) \sim_{\texttt{ra}} \texttt{root}(\mathcal{A}')$ implies the constraint $\texttt{root}(\mathcal{A}) \sim_{\texttt{ra}} q'$.

Inferring local constraints from local constraints

The saturation rules (shown in Table 5.2) that infer new local constraints from already existing ones in a transition $q \to a(q_1, \ldots, q_n) : c$ are, for $1 \leq i \leq n$, based on the following:

1. weakening the existing constraints: if $q \to a(q_1, \ldots, q_n) : c$ is a transition, then
   - L-RootRoot weakens a $\sim_{\texttt{ra}}$ relation to a $\sim_{\texttt{rr}}$ relation,
   - L-Weak infers the weaker constraints $0 \preceq_{\texttt{r}x} i$ from $0 \prec_{\texttt{r}x} i$ and $0 \succeq_{\texttt{r}x} i$ from $0 \succ_{\texttt{r}x} i$ for any $x \in \{\texttt{r}, \texttt{a}\}$,

Table 5.3.: Rules for inferring local constraints from global constraints.

$$\frac{q \sim_{\mathtt{r}x} q_i}{0 \sim_{\mathtt{r}x} i \in c}\ \textsc{L-G-Prop} \qquad\qquad \frac{q_i \rightarrow \overline{j}() \qquad q \sim_{\mathtt{r}x} \mathit{root}(\mathcal{A}_j)}{0 \sim_{\mathtt{r}x} i \in c}\ \textsc{L-G-Ref}$$

- We assume the transition $q \rightarrow a(q_1, \ldots, q_n) : c$ and $1 \leq i \leq n$,
- $x \in \{r, a\}$, and
- $q_i \rightarrow \overline{j}()$ is the only outgoing transition of q_i.

2. strengthening of existing data constraints:
    - L-Stre is used for q_i such that q_i has only nullary outgoing transitions to strengthen a constraint $0 \sim_{\mathtt{rr}} i$ to the constraint $0 \sim_{\mathtt{ra}} i$.

Inferring local constraints from global constraints

Inference of local constraints in a transition $q \rightarrow a(q_1, \ldots, q_n) : c$ from global constraints is done, for $1 \leq i \leq n$, using the rules shown in Table 5.3:

- L-G-Prop propagates a global constraint $q \sim_{\mathtt{r}x} q_i$ for states used in the same transition into a local constraint $0 \sim_{\mathtt{r}x} i$,
- L-G-Ref propagates a global constraint $q \sim_{\mathtt{r}x} \mathit{root}(\mathcal{A}_j)$ between a state q and the root state of a TA $\mathcal{A}_j$ into a local constraint $0 \sim_{\mathtt{ra}} i$ between q and q_i that accepts a reference to the TA $\mathcal{A}_j$.

Inferring global constraints from local constraints

Finally, new global constraints can be inferred from existing ones by propagating them over local constraints of transitions in which the states of the global constraints occur. Since a single state may be reached in several different ways, propagation of global constraints through local constraints on all transitions arriving to the given state must be considered. If some of the ways how to get to the state does not allow the propagation, it cannot be done. Moreover, since one propagation can enable another one, the propagation must be done iteratively until the fixpoint is reached. The iterative propagation must terminate since the number of constraints that can be used is finite. The propagation of constraints between states of a TA can be performed either downwards from the root towards leaves or upwards from leaves towards the root as described below. Let p be the root state of some TA $\mathcal{A}$. For each state q of $\mathcal{A}$, let $\Phi(q, p)$ be the set of global constraints between q and p. The data constraints are propagated in two directions:

Downward propagation. In the downward propagation, we simultaneously extend the sets $\Phi(q, p)$ to larger ones $\Psi(q, p)$ starting from the root state q_0 of $\mathcal{A}$ and setting

$\Psi(q_0, p) = \Phi(q_0, p)$ (i.e. no constraints are added for this case). Then, for non-root states q, we extend the set of constraints in $\Psi(q, p)$ by traversing over the transitions of $\mathcal{A}$ and adding constraints according to the following rules:

- We add the constraint $q\ ((\sim')^{-1} \circ \sim)_{\mathtt{r}x}\ p$, with $x \in \{\mathtt{a}, \mathtt{r}\}$, if, for every occurrence of q as q_i in any transition $\delta = q' \to a(q_1, \dots, q_n) : c$, there is a local constraint $0 \sim'_{\mathtt{rr}} i$ in c and a global constraint $q' \sim_{\mathtt{r}x} p$ in $\Psi(q', p)$.
- We add the constraint $p\ (\sim \circ \sim')_{\mathtt{r}x}\ q$, with $x \in \{\mathtt{a}, \mathtt{r}\}$, if, for every occurrence of q as q_i in any transition $\delta = q' \to a(q_1, \dots, q_n) : c$, there is a local constraint $0 \sim'_{\mathtt{r}x} i$ in c and a global constraint $p \sim_{\mathtt{r}y} q'$ in $\Psi(q', p)$ with $y \in \{\mathtt{a}, \mathtt{r}\}$.
- We add the constraint $p \sim_{\mathtt{ra}} q$ if, for every occurrence of q as q_i in any transition $\delta = q' \to a(q_1, \dots, q_n) : c$, it holds that $p \sim_{\mathtt{ra}} q'$ is in $\Psi(q', p)$.

Intuitively, the first two cases use transitivity to propagate a constraint involving q' to a constraint involving q_i; the last case uses the semantics of $p \sim_{\mathtt{ra}} q'$.

Upward propagation. The upward propagation can be defined analogously. Already existing sets of constraints $\Phi(q, p)$ can be extended to sets $\Psi(q, p)$ by traversing over the transitions of $\mathcal{A}$ and adding constraints according to the following rules:

- We add the constraint $p \sim_{\mathtt{ra}} q$ if there is the constraint $p \sim_{\mathtt{rr}} q$ is in $\Psi(q, p)$, and for every transition $\delta = q \to a(q_1, \dots, q_n) : c$ it holds that $p \sim_{\mathtt{ra}} q_i \in \Psi(q_i, p)$ for every $1 \leq i \leq n$.
- We add the constraint $q\ (\sim' \circ \sim)_{\mathtt{r}x}\ p$, with $x \in \{\mathtt{a}, \mathtt{r}\}$, if there is no nullary transition going from q and for every transition $\delta = q \to a(q_1, \dots, q_n) : c$, there are the constraints $0 \sim'_{\mathtt{rr}} i$ in c and $q_i \sim_{\mathtt{r}x} p$ in $\Psi(q_i, p)$ for some $1 \leq i \leq n$.
- We add the constraint $p\ (\sim \circ (\sim')^{-1})_{\mathtt{rr}}\ q$, with $x \in \{\mathtt{a}, \mathtt{r}\}$, if there is no nullary transition going from q and for every transition $\delta = q \to a(q_1, \dots, q_n) : c$, there are the constraints $0 \sim'_{\mathtt{rr}} i$ in c and $p \sim_{\mathtt{r}x} q_i$ in $\Psi(q_i, p)$ for some $1 \leq i \leq n$.

Proposition 5.1. *The constraint saturation process always terminates.*

Proof. Follows from the facts that the maximum number of constraints in an FA is finite and that adding a new constraint is a monotone operation. □

5.2.2. Abstract Transformers

In this section, we present the abstract transformers corresponding to some of the operations on abstract configurations of the form $\langle \sigma, F \rangle$ (also see Section 3.2 for the basic description of abstract transformers). For simplicity of the presentation, we assume that for all TAs $\mathcal{A}_i$ in F, (a) the root state of $\mathcal{A}_i$ does not appear on the right-hand side of any transition, and (b) it occurs on the left-hand side of exactly one transition. It is easy to see that any TA can be transformed into this form, the transformation procedure, called *unwinding*, is described in the following.

Unwinding the Root State

In order to transform a TA $\mathcal{A} = (Q, \Sigma, \Delta_c, \{q_f\})$, from an FA F into the form where q_f does not appear on the right-hand side of any transition and appears on the left-hand side of exactly one transition, we may perform the following sequence of actions:

1. create a copy q'_f of q_f, which replaces q_f on the right-hand side of all transitions,
2. duplicate all transitions from q_f to become transitions also from q'_f (while again substituting any occurrence of q_f with q'_f),
3. split $\mathcal{A}$ into several TAs, one for each transition from the accepting state q_f, creating several copies of the FA F that contains $\mathcal{A}$, and
4. adapt the local and global constraints by duplicating them whenever some state is duplicated.

An example of this transformation, which basically unfolds once all loops on q_f, will be given in Example 5.4 below.

We now introduce some common notation and operations for the below presented transformers. We use $\mathcal{A}_{\sigma(\mathtt{x})}$ and $\mathcal{A}_{\sigma(\mathtt{y})}$ to denote the TA pointed by variables `x` and `y`, respectively, and $q_{\mathtt{x}}$ and $q_{\mathtt{y}}$ to denote the root states of these TAs. Let $q_{\mathtt{y}} \to a(q_1, \ldots, q_i, \ldots, q_m) : c$ be the unique transition from $q_{\mathtt{y}}$. Before describing the actual update, let us first define how to split a TA.

Splitting a TA

The operation of *splitting* a TA $\mathcal{A}_{\sigma(\mathtt{y})}$ at the i-th position, for $1 \leq i \leq m$, is described by the following sequence of operations:

1. First, a new TA $\mathcal{A}_k$ is appended to F such that $\mathcal{A}_k$ is a copy of $\mathcal{A}_{\sigma(\mathtt{y})}$ but with q_i as the root state.
2. Second, the root transition in $\mathcal{A}_{\sigma(\mathtt{y})}$ is changed to $q_{\mathtt{y}} \to a(q_1, \ldots, q_{\overline{k}}, \ldots, q_m) : c'$ where c' is obtained from c by replacing any local constraint of the form $0 \sim_{\mathtt{r}x} i$ by the global constraint $q_{\mathtt{y}} \sim_{\mathtt{r}x} root(\mathcal{A}_k)$, and the transition $q_{\overline{k}} \to \overline{k}()$ is added to $\mathcal{A}_{\sigma(\mathtt{y})}$ (we assume $q_{\overline{k}}$ is a new state in $\mathcal{A}_{\sigma(\mathtt{y})}$).
3. Global data constraints are adapted as follows: For each constraint $q \sim_{\mathtt{r}x} p$ where q is in $\mathcal{A}_{\sigma(\mathtt{y})}$ such that $q \neq q_{\mathtt{y}}$, a new constraint $q' \sim_{\mathtt{r}x} p$ is added, where q' is the version of q in $\mathcal{A}_k$. Likewise, for each constraint $q \sim_{\mathtt{r}x} p$ where p is in $\mathcal{A}_{\sigma(\mathtt{y})}$ such that $p \neq q_{\mathtt{y}}$, a new constraint $q \sim_{\mathtt{r}x} p'$ is added (again, p' is the version of p in $\mathcal{A}_k$). Finally, for each constraint of the form $p \sim_{\mathtt{ra}} q_{\mathtt{y}}$, a new constraint $p \sim_{\mathtt{ra}} root(\mathcal{A}_k)$ is added.

An example of the splitting step is also given in Example 5.4 below.

Description of Abstract Transformers

In what follows, we assume the existence of the sub-term $q_{\mathtt{y}} \to \mathtt{sel}(q_i)$ in the (single) root transition of $\mathcal{A}_{\sigma(\mathtt{y})}$. Before performing the actual update, we check whether the operation to be performed tries to dereference a pointer to $\perp$ or to an undefined value, in which case we stop the analysis and report an error. Otherwise, we continue by performing one of the following actions, depending on the particular statement.

$\mathtt{x} := \mathtt{malloc()}$ We extend F with a new TA $\mathcal{A}_{new}$ containing one state and one transition where all selector values are undefined and assign $\sigma(\mathtt{x})$ to the index of $\mathcal{A}_{new}$ in F.

$\mathtt{x} := \mathtt{y\text{->}sel}$ If q_i is a root reference (say, j), it is sufficient to change the value of $\sigma(\mathtt{x})$ to j. Otherwise, we split $\mathcal{A}_{\sigma(\mathtt{y})}$ at the i-th position (creating $\mathcal{A}_k$) and assign k to $\sigma(\mathtt{x})$.

$\mathtt{y\text{->}sel} := \mathtt{x}$ If q_i is a state, then we split $\mathcal{A}_{\sigma(\mathtt{y})}$ at the i-th position. In both cases we insert q_{new} in the i-th position (instead of q_i) in the children states of the root transition of $\mathcal{A}_{\sigma(\mathtt{y})}$ (we assume q_{new} is a new state in $\mathcal{A}_{\sigma(\mathtt{y})}$). We follow by adding the transition $q_{new} \to \overline{\sigma(\mathtt{x})}()$ into $\mathcal{A}_{\sigma(\mathtt{y})}$. Any local constraint in c of the form $0 \sim_{\mathtt{r}x} i$ that concerns the removed root reference q_i is then removed from c.

$\mathtt{y\text{->}data} := \mathtt{x\text{->}data}$ First, we remove any local constraint that involves $q_{\mathtt{y}}$ or a root reference to $\mathcal{A}_{\sigma(\mathtt{y})}$. Then, we add a new global constraint $q_{\mathtt{y}} =_{\mathtt{rr}} q_{\mathtt{x}}$, and we also keep all global constraints of the form $q' \sim_{\mathtt{r}x} q_{\mathtt{y}}$ if $q' \sim_{\mathtt{rr}} q_{\mathtt{x}}$ is implied by the constraints obtained after the update.

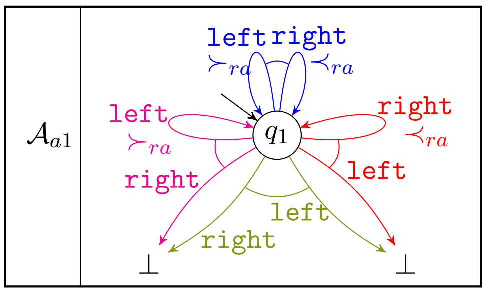
$\mathtt{y\text{->}data} \sim \mathtt{x\text{->}data}$ (where $\sim\, \in \{\prec, \preceq, \succ, \succeq\}$) First, we execute the saturation procedure in order to infer the strongest constraints between $q_{\mathtt{y}}$ and $q_{\mathtt{x}}$. Then, if there exists a global constraint $q_{\mathtt{y}} \sim' q_{\mathtt{x}}$ that implies $q_{\mathtt{y}} \sim q_{\mathtt{x}}$ (resp. its negation), we return *true* (resp. *false*). Otherwise, we copy $\langle \sigma, F \rangle$ into two abstract configurations: $\langle \sigma, F_{true} \rangle$ for the *true* branch and $\langle \sigma, F_{false} \rangle$ for the *false* branch. Moreover, we extend F_{true} with the global constraint $q_{\mathtt{y}} \sim q_{\mathtt{x}}$ and F_{false} with its negation.

$\mathtt{x} := \mathtt{y}$ **or** $\mathtt{x} := \mathtt{NULL}$ We simply update σ accordingly.

$\mathtt{free(y)}$ First, we split $\mathcal{A}_{\sigma(\mathtt{y})}$ at all j-th positions, $1 \le j \le m$, that appear in its root transition, then we remove $\mathcal{A}_{\sigma(\mathtt{y})}$ from F and set $\sigma(\mathtt{y})$ to undefined. However, to keep all possible data constraints, before removing $\mathcal{A}_{\sigma(\mathtt{y})}$, the saturation procedure is executed. After the action is done, every global constraint involving $q_{\mathtt{y}}$ is removed.

$\mathtt{x} = \mathtt{y}$ This operation is evaluated simply by checking whether $\sigma(\mathtt{x}) = \sigma(\mathtt{y})$. If $\sigma(\mathtt{x})$ or $\sigma(\mathtt{y})$ is undefined, we assume both possibilities.

After the update, we check that all TAs in F are referenced, either by a variable or from a root reference, otherwise we report an emergence of garbage.

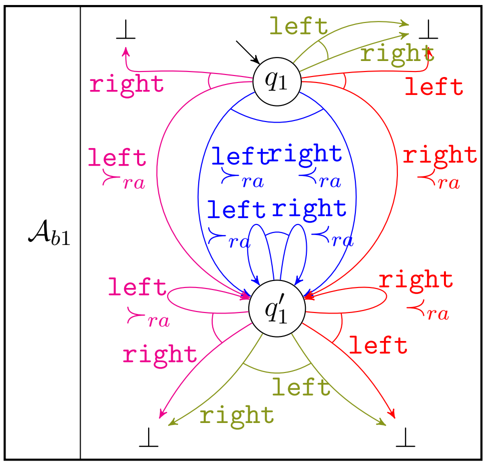


$F_a = \langle \mathcal{A}_{a1}, \emptyset \rangle$
$\sigma(\mathtt{root}) = 1, \sigma(\mathtt{x}) = 1$

$\mathcal{A}_{a1}$

$1a:$	$\underline{q_1} \to \mathtt{l}, \mathtt{r}(q_1, q_1)$	$: 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$2a:$	$\underline{q_1} \to \mathtt{l}, \mathtt{r}(q_\perp, q_1)$	$: 0 \prec_{\mathtt{ra}} 2$
$3a:$	$\underline{q_1} \to \mathtt{l}, \mathtt{r}(q_1, q_\perp)$	$: 0 \succ_{\mathtt{ra}} 1$
$4a:$	$\underline{q_1} \to \mathtt{l}, \mathtt{r}(q_\perp, q_\perp)$	

a) An example abstract configuration at line 9 of the program in Figure 5.1. The abstract configuration represents a set of BSTs ($\mathtt{l}, \mathtt{r}$ abbreviates $\mathtt{left}, \mathtt{right}$).



$F_b = \langle \mathcal{A}_{b1}, \emptyset \rangle$
$\sigma(\mathtt{root}) = 1, \sigma(\mathtt{x}) = 1$

$\mathcal{A}_{b1}$

$1b:$	$\underline{q_1} \to \mathtt{l}, \mathtt{r}(q_1', q_1')$	$: 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$2b:$	$\underline{q_1} \to \mathtt{l}, \mathtt{r}(q_\perp, q_1')$	$: 0 \prec_{\mathtt{ra}} 2$
$3b:$	$\underline{q_1} \to \mathtt{l}, \mathtt{r}(q_1', q_\perp)$	$: 0 \succ_{\mathtt{ra}} 1$
$4b:$	$\underline{q_1} \to \mathtt{l}, \mathtt{r}(q_\perp, q_\perp)$	
$5b:$	$q_1' \to \mathtt{l}, \mathtt{r}(q_1', q_1')$	$: 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$6b:$	$q_1' \to \mathtt{l}, \mathtt{r}(q_\perp, q_1')$	$: 0 \prec_{\mathtt{ra}} 2$
$7b:$	$q_1' \to \mathtt{l}, \mathtt{r}(q_1', q_\perp)$	$: 0 \succ_{\mathtt{ra}} 1$
$8b:$	$q_1' \to \mathtt{l}, \mathtt{r}(q_\perp, q_\perp)$	

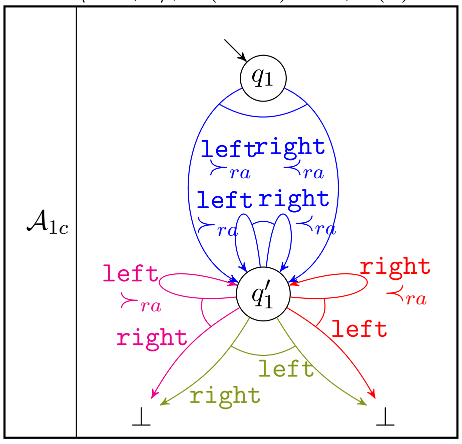
b) An intermediate state of unwinding the root state of $\mathcal{A}_{a1}$

Figure 5.5.: An example of unwinding the root state of a TA

Example 5.4. *We now present the computation of the abstract configuration that results from executing the program statements which appear at line 9 of the program in Figure 5.1 when starting from the abstract configuration described in Figure 5.5a (for the sake of brevity, we leave out the `newNode` variable and the corresponding TA from the example). In order to compute this abstract configuration, a sequence of two statements consisting of the test statement* `x->right` $\neq$ `NULL` *and the update statement* `x` $=$ `x->right` *is executed. First, the test statement* `x->right` $\neq$ `NULL` *is executed in the following two steps:*

1. *As can be seen from the FA F_a from Figure 5.5a encoding BSTs, the root state q_1 of $\mathcal{A}_{a1}$ (the only TA of F_a) occurs as a child state in three transitions of $\mathcal{A}_{a1}$, and we will therefore perform unwinding of q_1. We start by creating the state q_1', a copy of q_1, and duplicate to q_1' the four transitions leaving from q_1 (the resulting intermediate FA F_b can be seen in Figure 5.5b). Then, for each transition $t \in \{1b, 2b, 3b, 4b\}$ leaving from q_1 in $\mathcal{A}_{1b}$, we create a copy of the intermediate FA*

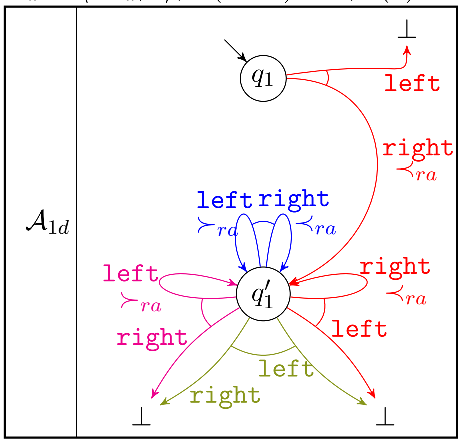
$F_c = \langle \mathcal{A}_{1c}, \emptyset \rangle$, $\sigma(\texttt{root}) = 1, \sigma(\texttt{x}) = 1$



$1c: \quad \underline{q_1} \to \texttt{1}, \texttt{r}(q_1', q_1') \quad : 0 \succ_{\texttt{ra}} 1, 0 \prec_{\texttt{ra}} 2$
$2c: \quad q_1' \to \texttt{1}, \texttt{r}(q_1', q_1') \quad : 0 \succ_{\texttt{ra}} 1, 0 \prec_{\texttt{ra}} 2$
$3c: \quad q_1' \to \texttt{1}, \texttt{r}(q_\perp, q_1') \quad : 0 \prec_{\texttt{ra}} 2$
$4c: \quad q_1' \to \texttt{1}, \texttt{r}(q_1', q_\perp) \quad : 0 \succ_{\texttt{ra}} 1$
$5c: \quad q_1' \to \texttt{1}, \texttt{r}(q_\perp, q_\perp)$

a) $\underline{q_1} \to \texttt{1}, \texttt{r}(q_1', q_1') : 0 \succ_{\texttt{ra}} 1, 0 \prec_{\texttt{ra}} 2$ $(1b)$

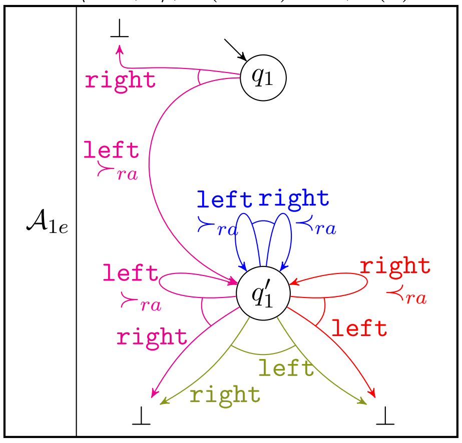
$F_d = \langle \mathcal{A}_{1d}, \emptyset \rangle$, $\sigma(\texttt{root}) = 1, \sigma(\texttt{x}) = 1$



$1d: \quad \underline{q_1} \to \texttt{1}, \texttt{r}(q_\perp, q_1') \quad : 0 \prec_{\texttt{ra}} 2$
$2d: \quad q_1' \to \texttt{1}, \texttt{r}(q_1', q_1') \quad : 0 \succ_{\texttt{ra}} 1, 0 \prec_{\texttt{ra}} 2$
$3d: \quad q_1' \to \texttt{1}, \texttt{r}(q_\perp, q_1') \quad : 0 \prec_{\texttt{ra}} 2$
$4d: \quad q_1' \to \texttt{1}, \texttt{r}(q_1', q_\perp) \quad : 0 \succ_{\texttt{ra}} 1$
$5d: \quad q_1' \to \texttt{1}, \texttt{r}(q_\perp, q_\perp)$

b) $\underline{q_1} \to \texttt{1}, \texttt{r}(q_\perp, q_1') : 0 \prec_{\texttt{ra}} 2$ $(2b)$

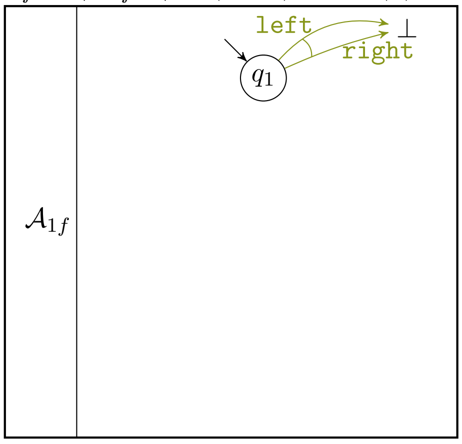
$F_e = \langle \mathcal{A}_{1e}, \emptyset \rangle$, $\sigma(\texttt{root}) = 1, \sigma(\texttt{x}) = 1$



$1e: \quad \underline{q_1} \to \texttt{1}, \texttt{r}(q_1', q_\perp) \quad : 0 \succ_{\texttt{ra}} 1$
$2e: \quad q_1' \to \texttt{1}, \texttt{r}(q_1', q_1') \quad : 0 \succ_{\texttt{ra}} 1, 0 \prec_{\texttt{ra}} 2$
$3e: \quad q_1' \to \texttt{1}, \texttt{r}(q_\perp, q_1') \quad : 0 \prec_{\texttt{ra}} 2$
$4e: \quad q_1' \to \texttt{1}, \texttt{r}(q_1', q_\perp) \quad : 0 \succ_{\texttt{ra}} 1$
$5e: \quad q_1' \to \texttt{1}, \texttt{r}(q_\perp, q_\perp)$

c) $\underline{q_1} \to \texttt{1}, \texttt{r}(q_1', q_\perp) : 0 \succ_{\texttt{ra}} 1$ $(3b)$

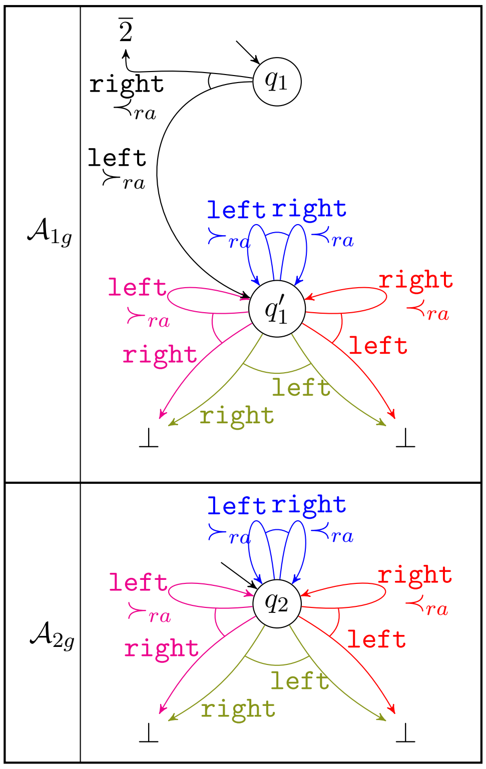
$F_f = \langle \mathcal{A}_{1f}, \emptyset \rangle$, $\sigma(\texttt{root}) = 1, \sigma(\texttt{x}) = 1$



$1f: \quad \underline{q_1} \to \texttt{1}, \texttt{r}(q_\perp, q_\perp)$

d) $\underline{q_1} \to \texttt{1}, \texttt{r}(q_\perp, q_\perp)$ $(4b)$

Figure 5.6.: The results of unwinding the root state of $\mathcal{A}_{a1}$ from Figure 5.5



$F_g = \langle \mathcal{A}_{1g}\, \mathcal{A}_{2g}, \{q_1 \prec_{\mathtt{ra}} q_2\}\rangle$
$\sigma(\mathtt{root}) = 1, \sigma(\mathtt{x}) = 2$

$\mathcal{A}_{1g}$

$1g:$	$\underline{q_1} \to \mathtt{l}, \mathtt{r}(q_1', q_{\overline{2}})$	$: 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$2g:$	$q_1' \to \mathtt{l}, \mathtt{r}(q_1', q_1')$	$: 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$3g:$	$q_1' \to \mathtt{l}, \mathtt{r}(q_\perp, q_1')$	$: 0 \prec_{\mathtt{ra}} 2$
$4g:$	$q_1' \to \mathtt{l}, \mathtt{r}(q_1', q_\perp)$	$: 0 \succ_{\mathtt{ra}} 1$
$5g:$	$q_1' \to \mathtt{l}, \mathtt{r}(q_\perp, q_\perp)$	
$6g:$	$q_{\overline{2}} \to \overline{2}()$	

$\mathcal{A}_{2g}$

$6g:$	$\underline{q_2} \to \mathtt{l}, \mathtt{r}(q_2, q_2)$	$: 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$7g:$	$\underline{q_2} \to \mathtt{l}, \mathtt{r}(q_\perp, q_2)$	$: 0 \prec_{\mathtt{ra}} 2$
$8g:$	$\underline{q_2} \to \mathtt{l}, \mathtt{r}(q_2, q_\perp)$	$: 0 \succ_{\mathtt{ra}} 1$
$9g:$	$\underline{q_2} \to \mathtt{l}, \mathtt{r}(q_\perp, q_\perp)$	

Figure 5.7.: The FA obtained from F_c (Figure 5.6a) by splitting $\mathcal{A}_{1c}$ at second position

called F_c, F_d, F_e, and F_f respectively. From the obtained TA $\mathcal{A}_{1c}$, $\mathcal{A}_{1d}$, $\mathcal{A}_{1e}$, and $\mathcal{A}_{1f}$, we subsequently remove all transitions leaving from q_1 other than t, resulting in the four FAs in Figure 5.6.

2. *The next step is to remove configurations where the root transition of the TA pointed by* `x` *has $q_\perp$ at the the second position of the tuple of children states since they do not pass the test* `x->right` $\neq$ `NULL` *(they will be processed in the* `else` *branch though). Due to this, the abstract configurations with the FAs F_e and F_f are removed.*

Second, the update statement `x` $=$ `x->right` *is executed on the abstract configurations shown in Figure 5.6a and Figure 5.6b. Here, we show the steps only for the abstract configuration from Figure 5.6a, the other one could be computed in a similar manner. The resulting abstract configuration is shown in Figure 5.7.*

1. *The first step is to compute the new FA resulting from splitting the root transition $1c$ of the TA $\mathcal{A}_{1c}$ in the FA F_c in Figure 5.6a at the second position, yielding the FA F_g. First, we create the TA $\mathcal{A}_{2g}$ from $\mathcal{A}_{1c}$ by copying it, renaming q_1' to q_2, and making the state q_2 the root state (note that q_1 becomes top-down unreachable in $\mathcal{A}_{2g}$, and so we discard it). Then, we copy $\mathcal{A}_{1c}$ to $\mathcal{A}_{1g}$ and change the root*

transition 1c of $\mathcal{A}_{1g}$ by replacing the state q'_1 at the second position of its tuple of children states (corresponding to the selector `right`*) by $q_{\overline{2}}$, and add (1) the leaf transition $q_{\overline{2}} \to \overline{2}()$ and (2) the global constraint $q_1 \prec_{\mathtt{ra}} q_2$.*

2. *The second step is to update the* valuation *σ of both abstract configurations to $\sigma := \sigma\{\mathtt{x} \mapsto 2\}$ meaning that* `x` *will point to roots of BSTs accepted by $\mathcal{A}_{2g}$ whereas $\sigma(\mathtt{root})$ is kept unchanged.* □

5.2.3. Normalisation

Normalisation transforms an FA $F = (\mathcal{A}_1 \cdots \mathcal{A}_n, \varphi)$ into a canonicity respecting FA in three major steps:

1. First, we transform F into a form in which roots of trees of accepted forests correspond to cut-points in a uniform way. In particular, for all $1 \le i \le n$ and all accepted forests $t_1 \cdots t_n$, one of the following holds: (a) If the root of t_i is the j-th cut-point in the canonical ordering of an accepted forest, then it is the j-th cut-point in the canonical ordering of all accepted forests. (b) Otherwise the root of t_i is not a cut-point of any of the accepted forests.

2. Then we merge TAs so that the roots of trees of accepted forests are cut-points only, which is described in detail below.

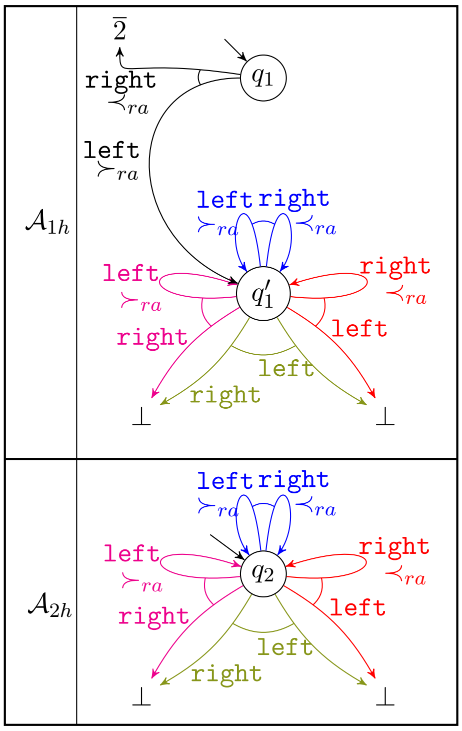
3. Finally, we reorder the TAs according to the canonical ordering of cut-points (which are roots of the accepted trees).

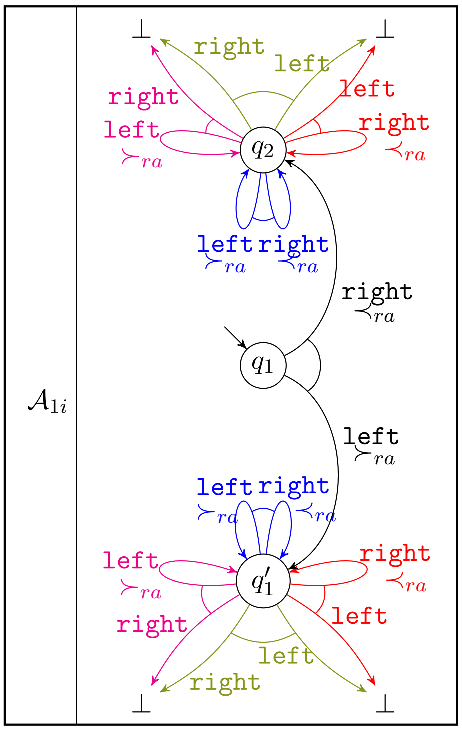
Our procedure is an augmentation of that in [HHR+12] used to normalise FAs without data constraints. The difference, which we describe below, is an update of data constraints while performing Step 2.

In order to minimise a possible loss of information encoded by data constraints, Step 2 is preceded by saturation (Section 5.2.1). Then, for all $1 \le i \le n$ such that roots of trees accepted by $\mathcal{A}_i = (Q_{\mathcal{A}}, \Sigma, \Delta_{\mathcal{A}}, \{q_{\mathcal{A}}\})$ are not cut-points of the graphs in $L(F)$ and such that there is a TA $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \Delta_{\mathcal{B}}, \{q_{\mathcal{B}}\})$ that contains a root reference to $\mathcal{A}_i$, Step 2 performs the following. The TA $\mathcal{A}_i$ is removed from F, the data constraints between q_A and non-root states of F are removed from φ, and $\mathcal{A}_i$ is connected to $\mathcal{B}$ at the places where $\mathcal{B}$ refers to it. In detail, $\mathcal{B}$ is replaced by the TA $(Q_{\mathcal{A}} \cup Q_{\mathcal{B}}, \Sigma, \Delta_{\mathcal{A}+\mathcal{B}}, \{q_{\mathcal{B}}\})$ where $\Delta_{\mathcal{A}+\mathcal{B}}$ is constructed from $\Delta_{\mathcal{A}} \cup \Delta_{\mathcal{B}}$ by modifying every transition $q \to a(q_1, \ldots, q_m) : c \in \Delta_{\mathcal{B}}$ as follows:

1. we replace by q_A all occurrences of $q_{\bar{i}}$ among $q_1, \ldots, q_m$ such that there is a transition $q_{\bar{i}} \to \bar{i}()$ in $\Delta_{\mathcal{B}}$ (note that there will be at most one such occurrence in a single transition), and

2. for all $1 \le k \le m$ such that q_k can reach the state $q_{\bar{i}}$ by following top-down a sequence of the original transitions of $\Delta_{\mathcal{B}}$, the constraint $0 \sim_{\mathtt{ra}} k$ is removed from c unless $q_k \sim_{\mathtt{ra}} q_A \in \varphi$ or $q_k = q_{\bar{i}}$ and $q \sim_{\mathtt{ra}} q_A \in \varphi$.





$F_h = \langle \mathcal{A}_{1h}\, \mathcal{A}_{2h}, \{q_1 \prec_{\mathtt{ra}} q_2\} \rangle$
$\sigma(\mathtt{root}) = 1, \sigma(\mathtt{x}) = \perp$

$F_i = \langle \mathcal{A}_{1i}, \emptyset \rangle$
$\sigma(\mathtt{root}) = 1, \sigma(\mathtt{x}) = \perp$

$\mathcal{A}_{1h}$

$1h:$	$\underline{q_1} \to \mathtt{l}, \mathtt{r}(q_1', q_{\overline{2}})$	$: 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$2h:$	$\underline{q_1'} \to \mathtt{l}, \mathtt{r}(q_1', q_1')$	$: 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$3h:$	$\underline{q_1'} \to \mathtt{l}, \mathtt{r}(q_\perp, q_1')$	$: 0 \prec_{\mathtt{ra}} 2$
$4h:$	$\underline{q_1'} \to \mathtt{l}, \mathtt{r}(q_1', q_\perp)$	$: 0 \succ_{\mathtt{ra}} 1$
$5h:$	$\underline{q_1'} \to \mathtt{l}, \mathtt{r}(q_\perp, q_\perp)$	
$6g:$	$q_{\overline{2}} \to \overline{2}()$	

$\mathcal{A}_{2h}$

$6h:$	$\underline{q_2} \to \mathtt{l}, \mathtt{r}(q_2, q_2)$	$: 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$7h:$	$\underline{q_2} \to \mathtt{l}, \mathtt{r}(q_\perp, q_2)$	$: 0 \prec_{\mathtt{ra}} 2$
$8h:$	$\underline{q_2} \to \mathtt{l}, \mathtt{r}(q_2, q_\perp)$	$: 0 \succ_{\mathtt{ra}} 1$
$9h:$	$\underline{q_2} \to \mathtt{l}, \mathtt{r}(q_\perp, q_\perp)$	

a) An abstract configuration

$\mathcal{A}_{1i}$

$1i:$	$\underline{q_1} \to \mathtt{l}, \mathtt{r}(q_1', q_2)$	$: 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$2i:$	$q_1' \to \mathtt{l}, \mathtt{r}(q_1', q_1')$	$: 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$3i:$	$q_1' \to \mathtt{l}, \mathtt{r}(q_\perp, q_1')$	$: 0 \prec_{\mathtt{ra}} 2$
$4i:$	$q_1' \to \mathtt{l}, \mathtt{r}(q_1', q_\perp)$	$: 0 \succ_{\mathtt{ra}} 1$
$5i:$	$q_1' \to \mathtt{l}, \mathtt{r}(q_\perp, q_\perp)$	
$6i:$	$q_2 \to \mathtt{l}, \mathtt{r}(q_2, q_2)$	$: 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$7i:$	$q_2 \to \mathtt{l}, \mathtt{r}(q_\perp, q_2)$	$: 0 \prec_{\mathtt{ra}} 2$
$8i:$	$q_2 \to \mathtt{l}, \mathtt{r}(q_2, q_\perp)$	$: 0 \succ_{\mathtt{ra}} 1$
$9i:$	$q_2 \to \mathtt{l}, \mathtt{r}(q_\perp, q_\perp)$	

b) The abstract configuration from (a) after normalisation

Figure 5.8.: An example of running normalisation on the abstract configuration obtained from the program in Figure 5.1 after executing line 22

Example 5.5. *In this example, we show normalisation of the FA in a possible abstract configuration after the execution of line 22 in the program in Figure 5.1. The abstract configuration can be seen in Figure 5.8a. Because the roots of the trees accepted by the TA $\mathcal{A}_{2h}$ do not correspond to the cut-points of the graphs in $L(F_h)$, we join $\mathcal{A}_{1h}$ and $\mathcal{A}_{2h}$ in the following way. First, the states and transitions of $\mathcal{A}_{2h}$ are copied to $\mathcal{A}_{1h}$ and the root state of $\mathcal{A}_{2h}$ substitutes the reference $\overline{2}$ in the transition $1h$ of $\mathcal{A}_{1h}$. Afterwards, the TA $\mathcal{A}_{2h}$ is removed together with the global data constraint $q_1 \prec_{\mathtt{ra}} q_2$ from the FA. The constraint $0 \prec_{\mathtt{ra}} 2$ is not removed from the root transition $1h$ because $q_1 \prec_{\mathtt{ra}} q_2$ was in the set of global data constraints of F_h before normalisation and, therefore, $0 \prec_{\mathtt{ra}} 2$ will still hold. The resulting FA F_i is shown in Figure 5.8b.* □

5.2.4. Checking Language Inclusion

In this section, we describe a reduction of checking language inclusion of FAs with data constraints to checking language inclusion of FAs without data constraints, which can be then done using the techniques of [HHR+12]. We note that "ordinary FAs" correspond to FAs with no global and no local data constraints, which were discussed in Chapter 3. The reduction encodes an FA with data constraints as an FA without data constraints such that its language, when decoded in a particular way, is the same as the language of the original automaton.

An *encoding* of an FA $F = (\mathcal{A}_1 \cdots \mathcal{A}_n, , \pi, \varphi)$ with data constraints is an ordinary FA $F^E = (\mathcal{A}'_1 \cdots \mathcal{A}'_n, \pi, \emptyset)$ where the data constraints are written into symbols of transitions. That is, each transition $q \to \langle a_1, \ldots, a_m\rangle(q_1, \ldots, q_m) : c$ of $\mathcal{A}_i$ is in $\mathcal{A}'_i$ replaced by the transition $q \to \langle (a_1, \ell_1, g) \cdots (a_m, \ell_m, g)\rangle(q_1, \ldots, q_m) : \emptyset$ where for $1 \le j \le m$, ℓ_j is the subset of c containing the local constraints involving j and g encodes the global constraints involving q as follows: Let r be the root state of some $\mathcal{A}_k$, for $1 \le k \le n$, that does not appear within the tuple of children states of any transition. Then for a global constraint $q \sim_{\mathtt{r}x} r$ or $r \sim_{\mathtt{r}x} q$, g contains $0 \sim_{\mathtt{r}x} k$ or $k \sim_{\mathtt{r}x} 0$ respectively. The language of $\mathcal{A}'_i$ thus consists of trees over the alphabet $\Gamma^E = \Gamma \times \mathbb{C} \times \mathbb{C}$ where $\mathbb{C}$ is the set of constraints of the form $j \sim_{\mathtt{r}x} k$ for $1 \le j, k \le n$.

To show that testing inclusion of encoded FAs is a sound approximation of language inclusion test of FAs with constraints, we need to establish a correspondence between languages of the encoded FAs and languages of the original ones. For this, we define a *decoding* of a forest $t'_1 \cdots t'_n$ from a language of an encoded FA over Γ^E as the set of forests $t_1 \cdots t_n$ over Γ such that $t_1 \cdots t_n$ arises from $t'_1 \cdots t'_n$ by (1) removing encoded constraints from the symbols, and (2) choosing data labeling that satisfies the constraints encoded within the symbols of $t'_1 \cdots t'_n$. Formally, for all $1 \le i \le n$, the set of nodes of t_i, V_{t_i}, is assigned to equal the set of nodes of t'_i, $V_{t'_i}$, and for all $a \in \Gamma$, $u, v \in V_{t_i}$, and $\ell, g \subseteq \mathbb{C}$, there is the sub-edge $u \to ((a, \ell, g), v)$ in $SE(t'_i)$ iff

(1) $u \to (a, v) \in SE(t_i)$ and

(2) for all $1 \le j \le n$: if $0 \sim_{\mathtt{r}x} j \in \ell$, then $u \sim_{\mathtt{r}x} v$ (in t_i), and if $0 \sim_{\mathtt{r}x} j \in g$, then $u \sim_{\mathtt{r}x} root(t_j)$ (symmetrically for $j \sim_{\mathtt{r}x} 0$).

Decoding of forests is naturally lifted to io-forests. The notion of decoding allows us to summarise the correspondence of languages of FAs and languages of their encodings as follows.

Lemma 5.1. *The set of io-forests accepted by an FA F is equal to the set of decodings of io-forests accepted by F^E.*

Proof. Let $F = \langle \mathcal{A}_1 \cdots \mathcal{A}_n, \pi, \varphi \rangle$ and $F^E = \langle \mathcal{A}'_1 \cdots \mathcal{A}'_n, \pi, \emptyset \rangle$. We first prove that every io-forest $(t_1 \cdots t_n, \pi)$ accepted by F is a decoding of some io-forest accepted by F^E. Let $\rho_1, \ldots, \rho_n$ be the runs of $\mathcal{A}_1, \ldots, \mathcal{A}_n$ on $t_1, \ldots, t_n$ respectively. We will construct runs $\rho'_1, \ldots, \rho'_n$ of $\mathcal{A}'_1, \ldots, \mathcal{A}'_n$ on the io-forest $(t'_1 \cdots t'_n, \pi)$ of which $(t_1 \cdots t_n, \pi)$ is a decoding of, such that for every ρ_i, we will construct the run ρ'_i. Let us first simplify the notation by denoting ρ_i, t_i, ρ'_i, t'_i, $\mathcal{A}_i$, and $\mathcal{A}'_i$ by ρ, t, ρ', t', $\mathcal{A}$, and $\mathcal{A}'$ respectively. The run ρ' is constructed as follows. The nodes of t' are set to the nodes of t, $V_{t'} = V_t$, and $\lambda_{t'}$ can be chosen arbitrarily. For every $v \in V_t$ such that $v \to (a_t^1, v_1), \ldots, v \to (a_t^m, v_m) \in SE(t, v)$, there is a transition of $\mathcal{A}$ of the form $\delta = q \to \langle a_t^1, \ldots, a_t^m \rangle (q_1, \ldots, q_m) : c$ such that the following conditions hold: $\rho(v) = q$, $\rho(v_1) = q_1, \ldots, \rho(v_m) = q_m$, the local constraints in c are satisfied by $v, v_1, \ldots, v_m$ in t, and also global constraints $q \sim_{\mathtt{rx}} r, r \sim_{\mathtt{rx}} q \in \varphi$ are satisfied by v and $\rho_k(r)$ for k such that r is a state of $\mathcal{A}_k$. The run ρ' then labels the nodes $v, v_1, \ldots, v_m$ using the transition $\delta' = q \to \vec{\alpha}(q_1, \ldots, q_m) : \emptyset$ that is the encoding of δ ($\vec{\alpha} = \langle (a_1, \ell_1, g), \ldots, (a_m, \ell_m, g) \rangle$ where g contains encoded the part of φ involving q and $c = \ell_1 \cup \cdots \cup \ell_m$). The run ρ' is obviously a run of $\mathcal{A}'$. The described construction of ρ' defines a map f that assigns to every $v, v_1, \ldots, v_m \in V_t$, where $v_1, \ldots, v_m$ are the children of v, a pair of transitions (δ, δ') of $\mathcal{A}$ and $\mathcal{A}'$ respectively, where δ and δ' are the transitions used within ρ and ρ' respectively to label the nodes $v, v_1, \ldots, v_m$.

First, let us argue that $t_1 \cdots t_n$ is indeed a decoding of $t'_1 \cdots t'_n$. It is trivially satisfied for all $1 \leq i \leq n$ that $V_{t_i} = V_{t'_i}$ and that every node has the same children in both forests. In order to argue that data values in $t_1 \cdots t_n$ satisfy the constraints encoded in $t'_1 \cdots t'_n$ as required by the definition of decoding, we let $v \in V_{t_i}$ be a node with children $v_1, \ldots, v_m$ such that $f(v, v_1, \ldots, v_m) = (\delta, \delta')$ where $\delta = q \to \langle a_1, \ldots, a_m \rangle (q_1, \ldots, q_m) : c$ and $\delta' = q \to \vec{\alpha}(q_1, \ldots, q_m) : \emptyset$ with $\vec{\alpha} = \langle (a_1, \ell_1, g) \cdots (a_m, \ell_m, g) \rangle$. Then the constraints imposed on the data value of v within $t_1 \cdots t_n$ by φ and those imposed by c due to the use of δ are the same as the constraints enforced on v due to $\vec{\alpha}$ when $t'_1 \cdots t'_n$ is decoded into $t_1 \cdots t_n$. In detail, c contains a local constraint $0 \sim k$ iff ℓ_k contains $0 \sim k$ (by the definition of encoding). This means that in the run of $\mathcal{A}$ on t, it is required that $v \sim v_k$, which is the same constraint as required by the decoding function. Further, there is a global constraint of the form $q \sim r \in \varphi$ such that r is the root state of $\mathcal{A}_k$ (not appearing within any children tuple of its transitions) iff $0 \sim k \in g$ (and analogically for the symmetrical cases). In the run of $\mathcal{A}$, $q \sim r$ enforces that $v \sim u$ where u is the root of t_k. Notice that u cannot be any other node than the root since r does not appear within the children tuple of any transition of $\mathcal{A}_k$. The constraint $v \sim u$ is precisely what is enforced due to $0 \sim k \in g$ when decoding $t'_1 \cdots t'_n$.

Second, we prove that every decoding $t_1 \cdots t_n$ of an io-forest $t'_1 \cdots t'_n \in L_f(F^E)$ is accepted by F. We will do that by showing that every n-tuple of runs $\rho'_1, \ldots, \rho'_n$ of $\mathcal{A}'_1, \ldots, \mathcal{A}'_n$ on $t_1, \ldots, t_n$ respectively also encodes runs of $\mathcal{A}_1, \ldots, \mathcal{A}_n$ on $t_1, \ldots, t_n$ respectively.

Recall first that by the definition of a decoding, for each $1 \leq i \leq n$, the trees t_i and t_i' have the same sets of nodes and every node has the same tuple of children. To simplify the notation, let $t, \rho', t', \mathcal{A}$, and $\mathcal{A}'$ be denoted as $t_i, \rho_i', t_i', \mathcal{A}_i$, and $\mathcal{A}_i'$ respectively. Let $v \in V_{t'}$ and let $v \to (\alpha_{t'}^1, v_1), \ldots, v \to (\alpha_{t'}^m, v_m) \in SE(t', v)$ where for all $1 \leq j \leq m$, $\alpha_{t'}^j = (a_j, \ell_j, g)$. By the definition of a decoding, v satisfies all constraints encoded within $\vec{\alpha} = \langle \alpha_{t'}^1, \ldots, \alpha_{t'}^m \rangle$. Since t' is accepted by $\mathcal{A}'$, there is a transition of $\mathcal{A}'$ of the form $\delta' = q \to \vec{\alpha}(q_1, \ldots, q_m) : \emptyset$ such that $\rho'(v) = q$, $\rho'(v_1) = q_1, \ldots, \rho'(v_m) = q_m$. By the definition of encoding, δ' was created from a transition $\delta = q \to \langle a_1, \ldots, a_m \rangle (q_1, \ldots, q_m) : c$ of $\mathcal{A}$ where $\ell_1 \cup \cdots \cup \ell_m = c$ and g encodes all global constraints involving q and a root state r that does not appear within a children tuple of any transition. These constraints are precisely those encoded within $\vec{\alpha}$ and hence required to hold for v in $t_1 \cdots t_n$ by decoding. The run ρ' is thus indeed a run of $\mathcal{A}$ since for every v and its children $v_1, \ldots, v_m$, there is a transition δ which can be used according to the definition of a run. $\square$

A direct consequence of Lemma 5.1 is that if $L(F_A^E) \subseteq L(F_B^E)$, then $L(F_A) \subseteq L(F_B)$. We can thus use the language inclusion checking procedure of [HHR+12] for ordinary FAs to safely approximate language inclusion of FAs with data constraints.

This language inclusion test is not complete, the above implication does not hold in the opposite direction. There are two reasons for this. First, encoding translates a constraint of F_B that is strictly weaker than a constraint of F_A into two different and unrelated labels. This may result in the situation that even though $L(F_A) \subseteq L(F_B)$, language inclusion of encodings of FAs does not hold due to the reason that the trees accepted are labelled by different symbols. For instance, let $F_A = (\mathcal{A}_1, \pi, \emptyset)$ where $\mathcal{A}_1$ contains only two transitions $\delta_A^1 = q \to a(q_{\bar{1}}) : \{0 \prec_{\texttt{rr}} 1\}$ and $\delta_A^2 = q_{\bar{1}} \to \bar{1}() : \emptyset$, and $F_B = (\mathcal{B}_1, \pi, \emptyset)$ where $\mathcal{B}_1$ also contains only two transitions $\delta_B^1 = r \to a(q_{\bar{1}}) : \emptyset$ and $\delta_B^2 = q_{\bar{1}} \to \bar{1}() : \emptyset$. It holds that $L(F_A) \subseteq L(F_B)$ (indeed, $L(F_A) = \emptyset$ due to the strict inequality on the root), but $L(F_A^E)$ is incomparable with $L(F_B^E)$. The reason is that δ_A and δ_B are encoded as transitions the symbols of which differ due to different data constraints. The fact that the constraint $\emptyset$ is weaker than the constraint of $0 \prec_{\texttt{rr}} 1$ plays no role. The second source of incompleteness of the inclusion test is that decodings of some forests accepted by F_A^E and F_B^E may be empty due to inconsistent data constraints. If the set of such inconsistent forests of F_A^E is not included in that of F_B^E, then $L(F_A^E)$ cannot be included in $L(F_B^E)$, but the inclusion $L(F_A) \subseteq L(F_B)$ can still hold since the forests with empty decodings do not contribute to $L(F_A)$ and $L(F_B)$ (in the sense of Lemma 5.1).

We do not attempt to resolve the problem of inconsistent data constraints since it does not seem to occur in practice, as witnessed by our experiments. On the other hand, the issue of incompatible encodings of related data constraints appears to be of a practical consequence. We address it with a quite simple transformation of F_B^E: We pump-up the TAs of F_B^E by variants of their transitions which encode stronger data constraints than originals and match the data constraints on transitions of F_A^E. Since we are adding transitions with stronger constraints than the existing ones, this does not change the language of F_B. For instance, in our previous example, we add the transition $r \to a(q_{\bar{1}}) : \{0 \prec_{\texttt{rr}} 1\}$ to $\mathcal{B}_1$. This transition, when encoded, can then correspond to the encoded version of the transition $q \to a(q_{\bar{1}}) : \{0 \prec_{\texttt{rr}} 1\}$ of $\mathcal{A}_1$ and the language inclusion of the encodings will hold.

Formally, we call a sequence $\vec{\alpha} = \langle (a_1, \ell_1, g), \cdots, (a_m, \ell_m, g)\rangle \in (\Gamma^E)^m$ *stronger* than a sequence $\vec{\beta} = \langle (a_1, \ell'_1, g'), \cdots, (a_m, \ell'_m, g')\rangle$ iff $\bigwedge g \implies \bigwedge g'$ and for all $1 \leq i \leq m$, $\bigwedge \ell_i \implies \bigwedge \ell'_i$. Intuitively, $\vec{\alpha}$ encodes the same sequence of symbols $\langle a_1, \dots, a_m\rangle$ as $\vec{\beta}$ and stronger local and global data constraints than $\vec{\beta}$. We modify F_B^E in such a way that for each transition $r \to \vec{\alpha}(r_1, \dots, r_m)$ of F_B^E and each transition of F_A^E of the form $q \to \vec{\beta}(q_1, \dots, q_m)$ where $\vec{\beta}$ is stronger than $\vec{\alpha}$, we add the transition $q \to \vec{\beta}(q_1, \dots, q_m)$. The modified FA, denoted by $F_B^{E^+}$, accepts the same or more forests than F_B^E (since its TAs have more transitions), but the sets of decodings of the accepted forests are the same (since the added transitions encode stronger constraints than the existing transitions). The FA $F_B^{E^+}$ can thus be used within language inclusion checking in the place of F_B^E. This technique prevents the inclusion check to fail because of incompatible encodings of data constraints. Its soundness is summarised by the following lemma.

Lemma 5.2. *Given two FAs F_A and F_B, $L(F_A^E) \subseteq L(F_B^{E^+}) \implies L(F_A) \subseteq L(F_B)$.*

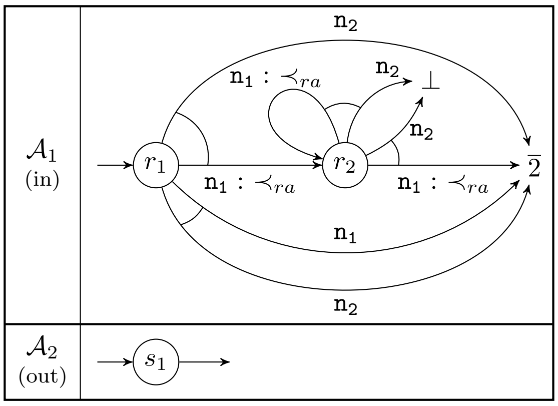
Proof sketch. Since the transformation from F_B^E to $F_B^{E^+}$ adds only versions of existing transitions encoding stronger constraints, the sets of decodings of forest of $F_B^{E^+}$ is the same as the set of decodings of forests of F_B^E. The statement then follows immediately from Lemma 5.1. □

We note that the same construction is used when checking language inclusion between sets of FAs with data constraints in a combination with the construction of [HHR+12] for checking inclusion of sets of ordinary FAs.

5.3. Boxes

In this chapter, we have so far considered only "flat" FAs, i.e. FAs without boxes. The extension of FAs by data constraints must, however, also be reflected within treatment of those. Particularly, in order not to lose information stored within data constraints, folding and unfolding require calls of the saturation procedure. When folding, saturation is used to transform global constraints into local ones. Namely, global constraints between the root state of the TA that is to become the input port of a box and the state of the TA that is to become an output port of the box is transformed into a local constraint of the newly introduced transition that uses the box as a label. When unfolding, saturation is used to transform local constraints into global ones. Namely, local constraints between the parent state of the transition with the unfolded box and a child state attached to the unfolded box is transformed to a global constraint between the root states of the TAs within the box that correspond to its input and output ports.

Example 5.6. *In this example we show how to unfold and fold boxes on a sample abstract configuration of a program manipulating a 2-level skip list. A skip list is a linked list sorted by keys. Each node is assigned a height, either 1 or 2, and one successor for every level. For example, a node of level 2 has two next pointers, here called* $\texttt{n}_1$ *and* $\texttt{n}_2$*, where* $\texttt{n}_1$ *points to the next node of level 1 and* $\texttt{n}_2$ *points to the next node of level 2. Figure 5.2*



$F_{\mathtt{skl}_2} = (\mathcal{A}_1\,\mathcal{A}_2, (1,2), \emptyset)$

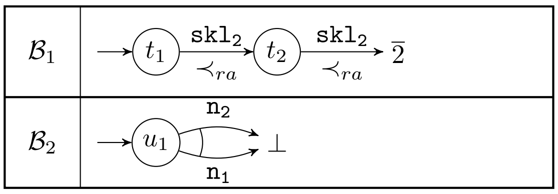
$\mathcal{A}_1$

$1a:$	$\underline{r_1} \to \mathtt{n}_1, \mathtt{n}_2(r_2, q_{\overline{2}})$	$: 0 \prec_{\mathtt{ra}} 1$
$2a:$	$\underline{r_1} \to \mathtt{n}_1, \mathtt{n}_2(q_{\overline{2}}, q_{\overline{2}})$	
$3a:$	$r_2 \to \mathtt{n}_1, \mathtt{n}_2(r_2, q_{\perp})$	$: 0 \prec_{\mathtt{ra}} 1$
$4a:$	$r_2 \to \mathtt{n}_1, \mathtt{n}_2(q_{\overline{2}}, q_{\perp})$	$: 0 \prec_{\mathtt{ra}} 1$
$5a:$	$q_{\overline{2}} \to \overline{2}()$	

$\mathcal{A}_2$

$6a:$	$\underline{s_1} \to \varepsilon()$	

a) The 2-level skip list box $\mathtt{skl}_2$



$F = (\mathcal{B}_1\,\mathcal{B}_2, (1), \varphi)$
$\varphi = \{t_1 \prec_{\mathtt{ra}} u_1, t_2 \prec_{\mathtt{ra}} u_1\}$
$\sigma(\mathtt{head}) = 1, \sigma(\mathtt{tail}) = 2$

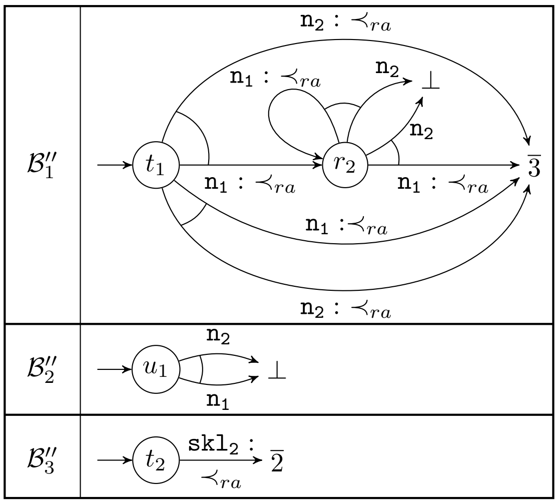
$\mathcal{B}_1$

$1b:$	$\underline{t_1} \to \mathtt{skl}_2(t_2)$	$: 0 \prec_{\mathtt{ra}} 1$
$2b:$	$t_2 \to \mathtt{skl}_2(q_{\overline{2}})$	$: 0 \prec_{\mathtt{ra}} 1$
$3b:$	$q_{\overline{2}} \to \overline{2}()$	

$\mathcal{B}_2$

$4b:$	$\underline{u_1} \to \mathtt{n}_1, \mathtt{n}_2(q_{\perp}, q_{\perp})$	

b) A heap containing a skip list with two segments



$F' = (\mathcal{B}_1''\,\mathcal{B}_2''\,\mathcal{B}_3'', (1), \varphi')$
$\varphi' = \{t_1 \prec_{\mathtt{ra}} u_1, t_2 \prec_{\mathtt{ra}} u_1, \mathbf{t_1} \prec_{\mathtt{ra}} \mathbf{t_2},$
$\mathbf{r_2} \prec_{\mathtt{ra}} \mathbf{u_1}, \mathbf{r_2} \prec_{\mathtt{ra}} \mathbf{t_2}\}$
$\sigma(\mathtt{head}) = 1, \sigma(\mathtt{tail}) = 2$

$\mathcal{B}_1''$

$1c:$	$\underline{t_1} \to \mathtt{n}_1, \mathtt{n}_2(r_2, q_{\overline{3}})$	$: 0 \prec_{\mathtt{ra}} 1, \mathbf{0} \prec_{\mathtt{ra}} \mathbf{2}$
$2c:$	$\underline{t_1} \to \mathtt{n}_1, \mathtt{n}_2(q_{\overline{3}}, q_{\overline{3}})$	$: \mathbf{0} \prec_{\mathtt{ra}} \mathbf{1}, \mathbf{0} \prec_{\mathtt{ra}} \mathbf{2}$
$3c:$	$r_2 \to \mathtt{n}_1, \mathtt{n}_2(r_2, q_{\perp})$	$: 0 \prec_{\mathtt{ra}} 1$
$4c:$	$r_2 \to \mathtt{n}_1, \mathtt{n}_2(q_{\overline{3}}, q_{\perp})$	$: 0 \prec_{\mathtt{ra}} 1$
$5c:$	$q_{\overline{3}} \to \overline{3}()$	

$\mathcal{B}_2''$

$6c:$	$\underline{u_1} \to \mathtt{n}_1, \mathtt{n}_2(q_{\perp}, q_{\perp})$	

$\mathcal{B}_3''$

$7c:$	$\underline{t_2} \to \mathtt{skl}_2(q_{\overline{2}})$	$: 0 \prec_{\mathtt{ra}} 1$
$8c:$	$q_{\overline{2}} \to \overline{2}()$	

c) Unfolding of the first occurrence of the $\mathtt{skl}_2$ box in (b)

Figure 5.9.: An example of unfolding of a box representing a 2-level skip list segment. We omitted all $\prec_{\mathtt{rr}}$ constraints which are subsumed by $\prec_{\mathtt{ra}}$ constraints.

shows an example configuration of a 2-level skip list with integer keys (the nodes `head` *and* `tail` *with the keys* $-\infty$ *and* $+\infty$ *respectively are used as sentinels).*

We can see from Figure 5.2 that each internal node of level 2 is a cut-point. In order to be able to represent a skip list of any length, it is necessary to introduce a box that effectively hides these cut-points. We use, in particular, the box $\mathtt{skl}_2$ *from Figure 5.9a, which represents all skip list segments between a pair of nodes of level 2. Figure 5.9b shows an abstract configuration of a skip list with 3 nodes of level 2: the* ***head*** *node, the* ***tail*** *node, and one regular node in between. The number of level 1 nodes (hidden inside the two* ***skl2*** *boxes) is arbitrary. Note that the single output port of* ***skl2*** *contains an automaton accepting* ε*—this is because there are no transitions leading from the output port of the box.*

Figure 5.9c shows an unfolding *of the first occurrence of the* $\mathtt{skl}_2$ *box in the FA. Intuitively, the unfolding proceeded in the following steps:*

1. *As a preparatory step for replacing the use of* $\mathtt{skl}_2$ *on the transition* $1b$ *by the contents of the box represented by* $\mathtt{skl}_2$*, the TA* $\mathcal{B}_1$ *was split at the state* t_2 *to isolate the transition* $1b$*. This produced two auxiliary TAs* $\mathcal{B}'_1$ *and* $\mathcal{B}'_3$ *consisting of the transitions* $\{[1b_1]\, t_1 \to \mathtt{skl}_2(q_{\overline{3}}) : \{0 \prec_{ra} 1\}, [1b_2]\, q_{\overline{3}} \to \overline{3}() : \emptyset\}$ *for* $\mathcal{B}'_1$ *and* $\{[2b_1]\, t_2 \to \mathtt{skl}_2(q_{\overline{2}}) : \{0 \prec_{ra} 1\}, [2b_2]\, q_{\overline{2}} \to \overline{2}() : \emptyset\}$ *for* $\mathcal{B}'_2$*, with a newly introduced cut-point 3.*

2. *Subsequently, the TA* $\mathcal{A}_1$ *corresponding to the input port of* $\mathtt{skl}_2$ *was inserted in between* t_1 *and* $q_{\overline{3}}$ *instead of the transition* $1b_1$ *over* $\mathtt{skl}_2$*, yielding the TA* $\mathcal{B}''_1$*. (Notice that if the transition* $1b_1$ *led—via other symbols than* $\mathtt{skl}_2$*—to more targets than just* $q_{\overline{3}}$*, the part of* $1b_1$ *leading from* t_1 *to such targets would be preserved and merged with the root transitions of* $\mathcal{A}_1$*.) On the other hand, the TA* $\mathcal{A}_2$ *corresponding to the single output port of* $\mathtt{skl}_2$ *was merged with the transition* $2b_1$ *leading from* t_2*. However, since* $\mathcal{A}_2$ *accepts* ε*, the resulting transition* $7c$ *of* $\mathcal{B}''_3$ *remains the same as the original transition* $2b$*. (The TA* $\mathcal{B}_2$ *was copied into the TA* $\mathcal{B}''_2$ *without any modification.)*

3. *The local data constraint from the transition* $1b : t_1 \to \mathtt{skl}_2(t_2) : 0 \prec_{ra} 1$ *was transformed into the global data constraint* $t_1 \prec_{ra} t_2$ *during the unfolding.*

The subsequent saturation then also generated the local constraints $0 \prec_{ra} 1$ *and* $0 \prec_{ra} 2$ *on the transitions* $1c$ *and* $2c$ *from* t_1 *to* $q_{\overline{3}}$*, and the global constraints* $r_2 \prec_{ra} t_2$ *and* $r_2 \prec_{ra} u_1$ *(these changes are emphasised by a bold typeface in Figure 5.9c).*

The inverse operation of folding *would transform the FA from Figure 5.9c, while using the* $\mathtt{skl}_2$ *box, into the FA in Figure 5.9b. See Chapters 3 and 4 for more details on box folding and unfolding.* □

5.4. Experimental Results

We have implemented the above presented techniques as an extension of the Forester tool and tested their generality and efficiency on a number of case studies. We considered programs dealing with SLLs, DLLs, BSTs, and skip lists. We verified the original

Table 5.4.: Results of the experiments

Example	time [s]	Example	time [s]
SLL insert	0.06	SL_2 insert	9.65
SLL delete	0.08	SL_2 delete	10.14
SLL reverse	0.07	SL_3 insert	56.99
SLL bubblesort	0.13	SL_3 delete	57.35
SLL insertsort	0.10		
DLL insert	0.14	BST insert	6.87
DLL delete	0.38	BST delete	15.00
DLL reverse	0.16	BST left rotate	7.35
DLL bubblesort	0.39	BST right rotate	6.25
DLL insertsort	0.43		

implementation of skip lists that uses the data ordering relation to detect the end of the operated window (as opposed to the implementation handled in the work presented in Chapter 4, which was modified to remove the dependency of the algorithm on sortedness). Although the examples are of a smaller size, they are very challenging as they include complex manipulation with dynamic memory that may depend on data values stored in memory cells.

Table 5.4 gives running times in seconds (the average of 10 executions) of the extension of Forester on our case studies. The names of the examples in the table contain the name of the data structure manipulated in the program, which is “SLL” for singly linked lists, “DLL” for doubly linked lists, and “BST” for binary search trees. “SL” stands for skip lists where the subscript denotes their level (the total number of `next` pointers in each cell). All experiments start with a random creation of an instance of the specified structure and end with its disposal. The indicated procedure is performed in between. The “insert” procedure inserts a node into an ordered instance of the structure, at the position given by the data value of the node, “delete” removes the first node with a particular data value, and “reverse” reverses the structure. “Bubblesort” and “insertsort” perform the given sorting algorithm on an unordered instance of the list. “Left rotate” and “right rotate” rotate the BST in the specified direction. Before the disposal of the data structure, we further check that it remained ordered after execution of the operation. The experiments were run on a machine with the Intel Core i5-480M @2.67 GHz CPU and 5 GiB of RAM.

Compared with works [LRS05, WKZ+07, BBH+11, QHL+13], which we consider the closest to our approach, the running times show that our approach is significantly faster. We, however, note that a precise comparison is not easy even with the mentioned works since as discussed in the related work paragraph, they can handle more complex properties on data, but on the other hand, they are less automated or handle less general classes of pointer structures.

5.4.1. Discussion

In the above, we described evaluation of our approach on programs manipulating skip lists of two and three levels. A natural question would be why we limit ourselves to two and three levels and not consider skip lists of even higher or, which would be the best case, of an arbitrary level.

Based on our experience, already going from 2-level to 3-level skip lists makes a huge difference in difficulty, due to the occurrence of a combinatorial explosion in the number of shapes considered by our approach. In order to make handling of a 3-level skip list feasible, we had to refine our finite height abstraction from a quite coarse one, which was sufficient for the other considered data structures, to take into account the number of unique paths from a state to a root reference (this step is described in more detail in Section 4.2 for the case without data relations). For the case of 4-level skip lists, this ad-hoc abstraction refinement was not sufficient and our experiments did not finish in reasonable time.

Moreover, in order to support skip lists with an arbitrary number of next selectors, these would need to be stored in a dynamic list, therefore making the data structure yet more complex. Even more, the support of a data structure of an arbitrary level in the current technique would need to use recursive nesting of boxes, which is not supported. Allowing this would demand to rewrite the box learning algorithm to be able to find such recursive boxes, and the operations for manipulating those, including the language inclusion algorithm. These modifications are quite challenging and an interesting future research direction.

5.5. Conclusion

In this chapter, we presented an extension of FA-based analysis of heap manipulating programs with a support for reasoning about data stored in dynamic memory. The resulting method allows verification of pointer programs where the needed inductive invariants combine complex shape properties with constraints over stored data, such as sortedness. The method is fully automatic, quite general, and its efficiency is comparable with other state-of-the-art analyses even though they handle less general classes of programs, are less automated, or both. We presented experimental results from verifying programs dealing with variants of (ordered) lists and trees. To the best of our knowledge, our method is the first one to cope fully automatically with a full C implementation of a 3-level skip list.

Part II.

Using Automata for Deciding Logics

6. Compositional Testing of Entailment for a Fragment of Separation Logic

Automatic verification of programs manipulating dynamic linked data structures is highly challenging since it requires one to reason about complex program configurations having the form of graphs of an unbounded size. For that, a highly expressive formalism is needed. Moreover, in order to scale to large programs, the use of such a formalism within program analysis should be highly efficient. In this context, *separation logic* (SL) [IO01, Rey02], a formalism complementary to forest automata presented in Chapters 3–5, has emerged as one of the most promising formalisms, offering both high expressiveness and scalability. The latter is due to its support of *compositional reasoning* based on the separating conjunction $*$ and the frame rule, which states that if a Hoare triple $\{\phi\}P\{\psi\}$ holds and P does not alter free variables in σ, then $\{\phi * \sigma\}P\{\psi * \sigma\}$ holds too. Therefore, when reasoning about P, one has to manipulate only specifications for the heap region altered by P.

Usually, SL is used together with higher-order *inductive definitions* that describe the data structures manipulated by the program. If we consider general inductive definitions, then SL is undecidable [CYO01]. Various decidable fragments of SL have been introduced in the literature [BCO05, IRŠ13, PWZ13, BFGP14] by restricting the syntax of the inductive definitions and the Boolean structure of the formulae.

In the work presented in this chapter, we focus on a fragment of SL with inductive definitions that allows one to specify program configurations (heaps) containing finite nestings of various kinds of linked lists (acyclic or cyclic, singly or doubly linked, skip lists, etc.), which are common in practice. This fragment contains formulae of the form $\exists\overrightarrow{X} : \Pi \wedge \Sigma$ where X is a set of variables, Π is a conjunction of (dis)equalities, and Σ is a set of *spatial atoms* connected by the separating conjunction. Spatial atoms can be *points-to atoms*, which describe values of pointer fields of a given heap location, or *inductively defined predicates*, which describe data structures of an unbounded size. We propose a novel (semi-)decision procedure for checking the validity of entailments of the form $\varphi \Rightarrow \psi$ where φ may contain existential quantifiers and ψ is a quantifier-free formula. Such a decision procedure can be used in Hoare-style reasoning to check inductive invariants but also in program analysis frameworks to decide termination of fixpoint computations. As usual, checking entailments of the form $\bigvee_i \varphi_i \Rightarrow \bigvee_j \psi_j$ can be soundly reduced to checking that for each i there exists j such that $\varphi_i \Rightarrow \psi_j$.

The key insight of our decision procedure is an idea to use the semantics of the separating conjunction in order to reduce the problem of checking $\varphi \Rightarrow \psi$ to the problem of checking a set of simpler entailments where the right-hand side is an inductively-defined predicate $P(\ldots)$. This reduction shows that the compositionality principle holds not only for deciding the validity of Hoare triples but also for deciding the validity of entailments between two formulae. To infer (dis)equalities implied by spatial atoms, our reduction to checking simpler entailments is based on Boolean unsatisfiability checking, which is in co-NP but can usually be checked efficiently by current SAT solvers.

Further, to check entailments $\varphi \Rightarrow P(\ldots)$ resulting from the above reduction, we define a decision procedure based on the membership problem for tree automata (TAs). In particular, we reduce the entailment to testing membership of a tree derived from φ in the language of a TA $\mathcal{A}[P]$ derived from $P(\ldots)$. The tree encoding of φ preserves some edges of the graph, called *backbone edges*, while others are re-directed to new nodes, related to the original destination by special symbols. Roughly, such a symbol may be a variable represented by the original destination, or it may show how to reach the original destination using backbone edges only.

Our procedure is complete for formulae speaking about non-nested singly as well as doubly linked lists. Moreover, it runs in polynomial time modulo an oracle for deciding validity of a Boolean formula. The procedure is incomplete for nested list structures due to not considering all possible ways in which targets of inner pointer fields of nested list predicates can be aliased. The construction can be easily extended to become complete even in such cases, but then it becomes exponential. However, even in this case, it is exponential in the size of the inductive predicates used, and not in the size of the formulae, which remains acceptable in practice.

We implemented our decision procedure and tested it successfully on verification conditions obtained from programs using singly and doubly linked nested lists as well as skip lists. The results show that our procedure does not only have a theoretically favorable complexity (for the given context), but it also behaves nicely in practice, at the same time offering the additional benefit of compositionality that can be exploited within larger verification frameworks caching the simpler entailment queries.

Related Work. Several decision procedures for fragments of SL have been introduced in the literature [BCO05, CYO01, CHO$^+$11, ESS13, IRŠ13, IRV14, PR11, PWZ13, BGP12]. Some of these works [BCO05, CYO01, CHO$^+$11, PR11] consider a fragment of SL that uses only a single predicate describing singly linked lists, which is a much more restricted setting than what is considered in this work. In particular, Cook *et al* [CHO$^+$11] prove that the satisfiability/entailment problem can be solved in polynomial time. Piskac *et al* [PWZ13] show that the Boolean closure of this fragment can be translated to a decidable fragment of first-order logic, and in this way they prove that the satisfiability/entailment problem can be decided in NP/co-NP. Furthermore, they consider the problem of combining SL formulae with constraints on data using the Nelson-Oppen theory combination framework. Adding constraints on data to SL formulae is considered also in Qiu *et al* [QGSM13].

A fragment of SL covering overlaid nested lists was considered by Enea *et al* [ESS13]. Compared with it, we currently do not consider overlaid lists, but we have enlarged the set of inductively-defined predicates to allow nesting of cyclic lists and doubly linked lists (DLLs). We also provide a novel and more efficient TA-based procedure for checking simple entailments.

Brotherston *et al* [BGP12] define a generic automated theorem prover relying on the notion of cyclic proofs and instantiate it to prove entailments in a fragment of SL with inductive definitions and disjunctions more general than what we consider here. However, they do not provide a fragment for which completeness is guaranteed. Iosif *et al* [IRŠ13] also introduce a decidable fragment of SL that can describe more complex data structures than those considered by the work presetned in this chapter, including e.g. trees with parent pointers or trees with linked leaves. However, [IRŠ13] reduces the entailment problem to MSO on graphs with a bounded tree width, resulting in a multiply-exponential complexity.

The recent work [IRV14] considers a more restricted fragment than [IRŠ13] (incomparable with ours). The work proposes a more practical, purely TA-based decision procedure, which reduces the entailment problem to *language inclusion* on TAs, establishing EXPTIME-completeness of the considered fragment. Our decision procedure deals with the Boolean structure of SL formulae using SAT solvers, thus reducing the entailment problem to the problem of entailment between a formula and an atom. Such simpler entailments are then checked using a polynomial semi-decision procedure based on the *membership problem* for TAs. The approach of [IRV14] can deal with various forms of trees and with entailment of structures with skeletons based on different selectors (e.g. DLLs viewed from the beginning and DLLs viewed from the end). On the other hand, it currently cannot deal with structures of zero length and with some forms of structure concatenation (such as concatenation of two DLL segments), which we can handle.

Contribution. Overall, the contribution of the work presented in this chapter is a novel (semi-)decision procedure for a rich class of verification conditions with singly as well as doubly linked lists, nested lists, and skip lists. As discussed in more detail in the previous paragraph, existing works that can efficiently deal with fragments of SL capable of expressing verification conditions for programs handling complex dynamic data structures are still rare. Indeed, we are not aware of any techniques that could decide the class of verification conditions considered in this work at the same level of efficiency as our procedure. In particular, compared with other approaches using TAs [IRŠ13, IRV14], our procedure is compositional as it uses TAs recognising models of predicates, not models of entire formulae. Moreover, our TAs recognise in fact formulae that entail a given predicate, reducing SL entailment to the membership problem for TAs, not the more expensive inclusion problem as in other works.

6.1. Separation Logic Fragment

Let *Vars* be a set of *program variables*, ranged over using x, y, z, and *LVars* a set of *logical variables*, disjoint from *Vars*, ranged over using X, Y, Z. We assume that *Vars* contains a variable `null`. Also, let $\mathbb{F}$ be a set of *fields*.

We consider the fragment of separation logic whose syntax is given below:

$x, y \in \mathit{Vars}$	program variables	$f \in \mathbb{F}$	fields
$X, Y \in \mathit{LVars}$	logical variables	$P \in \mathbb{P}$	predicates
$\overrightarrow{B} \in (\mathit{Vars} \cup \mathit{LVars})^*$	vectors of variables	$E, F \ ::= x \mid X$	
		$\rho ::= (f, E) \mid \rho, \rho$	

$\Pi ::= E = F \mid E \neq F \mid \Pi \wedge \Pi$	pure formulae
$\Sigma ::= \mathit{emp} \mid E \mapsto \{\rho\} \mid P(E, F, \overrightarrow{B}) \mid \Sigma * \Sigma$	spatial formulae
$\varphi \triangleq \ \exists \overrightarrow{X} : \Pi \wedge \Sigma$	formulae

W.l.o.g., we assume that existentially quantified logical variables have unique names. The set of program variables used in a formula φ is denoted by $\mathit{pv}(\varphi)$. By $\varphi(\overrightarrow{E})$ (resp. $\rho(\overrightarrow{E})$), we denote a formula (resp. a set of field-variable pairs) whose set of free variables is $\overrightarrow{E}$. Given a formula φ, $\mathit{pure}(\varphi)$ denotes its pure part Π. We allow set operations to be applied on vectors. Moreover, $E \neq \overrightarrow{B}$ is a shorthand for $\bigwedge_{B_i \in \overrightarrow{B}} E \neq B_i$.

The *points-to atom* $E \mapsto \{(f_i, F_i)\}_{i \in \mathcal{I}}$ specifies that the heap contains a location E whose f_i field points to F_i, for all i. W.l.o.g., we assume that each field f_i appears at most once in a set of pairs ρ. The fragment is parameterised by a set $\mathbb{P}$ of *inductively defined predicates*; intuitively, $P(E, F, \overrightarrow{B})$ describes a possibly empty *nested list segment* delimited by its arguments, i.e. all the locations it represents are reachable from E and allocated on the heap except the locations in $\{F\} \cup \overrightarrow{B}$.

Inductively defined predicates. We consider predicates defined as

$$\begin{aligned} P(E, F, \overrightarrow{B}) \triangleq &(E = F \wedge \mathit{emp}) \vee \\ &\big(E \neq \{F\} \cup \overrightarrow{B} \ \wedge \ \exists X_{\mathtt{tl}} : \Sigma(E, X_{\mathtt{tl}}, \overrightarrow{B}) \ * \ P(X_{\mathtt{tl}}, F, \overrightarrow{B})\big) \end{aligned} \tag{6.1}$$

where Σ is an existentially-quantified formula, called *the matrix of* P, of the form:

$$\begin{aligned} \Sigma(E, X_{\mathtt{tl}}, \overrightarrow{B}) &\triangleq \exists \overrightarrow{Z} : E \mapsto \{\rho(\{X_{\mathtt{tl}}\} \cup \overrightarrow{V})\} * \Sigma' \qquad \text{where } \overrightarrow{V} \subseteq \overrightarrow{Z} \cup \overrightarrow{B} \text{ and} \\ \Sigma' &::= Q(Z, U, \overrightarrow{Y}) \mid \circlearrowleft^{1+} Q[Z, \overrightarrow{Y}] \mid \Sigma' * \Sigma' \\ &\text{for } Z \in \overrightarrow{Z},\ U \in \overrightarrow{Z} \cup \overrightarrow{B} \cup \{E, X_{\mathtt{tl}}\},\ \overrightarrow{Y} \subseteq \overrightarrow{B} \cup \{E, X_{\mathtt{tl}}\}, \text{ and} \\ \circlearrowleft^{1+} Q[Z, \overrightarrow{Y}] &\triangleq \exists Z' : \Sigma_Q(Z, Z', \overrightarrow{Y}) * Q(Z', Z, \overrightarrow{Y}) \qquad \text{where } \Sigma_Q \text{ is the matrix of } Q. \end{aligned} \tag{6.2}$$

The formula Σ specifies the values of the fields defined in E (using the atom $E \mapsto \{\rho(\{X_{\mathtt{tl}}\} \cup \overrightarrow{V})\}$, where the fields in ρ are constants in $\mathbb{F}$) and the (possibly cyclic) nested list segments starting at the locations $\overrightarrow{Z}$ referenced by fields of E. We assume that Σ contains a single points-to atom in order to simplify the presentation. Notice that the matrix of a predicate P does not contain applications of P. The macro $\circlearrowleft^{1+} Q[Z, \overrightarrow{Y}]$ is used to represent a *non-empty* cyclic (nested) list segment on Z whose shape is described by the predicate Q.

We consider several restrictions on Σ which are defined using its *Gaifman graph* $Gf[\Sigma]$. The set of vertices of $Gf[\Sigma]$ is given by the set of free and existentially quantified variables in Σ, i.e. $\{E, X_{\mathtt{tl}}\} \cup \overrightarrow{B} \cup \overrightarrow{Z}$. The edges in $Gf[\Sigma]$ represent spatial atoms: for every (f, X) in ρ, $Gf[\Sigma]$ contains an edge from E to X labelled by f; for every predicate $Q(Z, U, \overrightarrow{Y})$, $Gf[\Sigma]$ contains an edge from Z to U labelled by Q; and for every macro $\circlearrowleft^{1+} Q[Z, \overrightarrow{Y}]$, $Gf[\Sigma]$ contains a self-loop on Z labelled by Q.

The first restriction is that $Gf[\Sigma]$ contains no cycles other than self-loops built solely of edges labelled by predicates. This ensures that the predicate is *precise*, i.e. for any heap, there exists at most one sub-heap on which the predicate holds. Precise assertions are very important for concurrent separation logic [GBC11].

The second restriction requires that all the maximal paths of $Gf[\Sigma]$ start in E and end either in a self-loop or in a node from $\overrightarrow{B} \cup \{E, X_{\mathtt{tl}}\}$. This restriction ensures that (a) all the heap locations in the interpretation of a predicate are reachable from the head of the list and that (b) only the locations represented by variables in $F \cup \overrightarrow{B}$ are dangling. Moreover, for simplicity, we require that every vertex of $Gf[\Sigma]$ has at most one outgoing edge labelled by a predicate.

For example, the predicates given in Figure 6.1 describe singly linked lists, lists of acyclic lists, lists of cyclic lists, and skip lists with three levels.

We define the relation $\prec_{\mathbb{P}}$ on $\mathbb{P}$ by $P_1 \prec_{\mathbb{P}} P_2$ iff P_2 appears in the matrix of P_1. The reflexive and transitive closure of $\prec_{\mathbb{P}}$ is denoted by $\prec_{\mathbb{P}}^*$. For example, if $\mathbb{P} = \{\mathtt{skl}_1, \mathtt{skl}_2, \mathtt{skl}_3\}$, then $\mathtt{skl}_3 \prec_{\mathbb{P}} \mathtt{skl}_2$ and $\mathtt{skl}_3 \prec_{\mathbb{P}}^* \mathtt{skl}_1$.

Given a predicate P of the matrix Σ as in Equation 6.2, let $\mathbb{F}_{\mapsto}(P)$ denote the set of fields f occurring in a pair (f, X) of ρ. For example, $\mathbb{F}_{\mapsto}(\mathtt{nll}) = \{s, h\}$ and $\mathbb{F}_{\mapsto}(\mathtt{skl}_3) = \mathbb{F}_{\mapsto}(\mathtt{skl}_1) = \{f_3, f_2, f_1\}$. Also, let $\mathbb{F}^*_{\mapsto}(P)$ denote the union of $\mathbb{F}_{\mapsto}(P')$ for all $P \prec_{\mathbb{P}}^* P'$. For example, $\mathbb{F}^*_{\mapsto}(\mathtt{nll}) = \{s, h, f\}$.

We assume that $\prec_{\mathbb{P}}^*$ is a partial order, i.e. there are no mutually recursive definitions in $\mathbb{P}$. Moreover, for simplicity, we assume that for any two predicates P_1 and P_2 which are incomparable w.r.t. $\prec_{\mathbb{P}}^*$, it holds that $\mathbb{F}_{\mapsto}(P_1) \cap \mathbb{F}_{\mapsto}(P_2) = \emptyset$. This assumption avoids predicates named differently but having exactly the same set of models.

Semantics. Let *Locs* be a set of *locations*. A *heap* is a pair (S, H) where $S : \mathit{Vars} \cup \mathit{LVars} \to \mathit{Locs}$ maps variables to locations and $H : \mathit{Locs} \times \mathbb{F} \rightharpoonup \mathit{Locs}$ is a partial function that defines values of fields for some of the locations in *Locs*. The domain of H is denoted by $\mathit{dom}(H)$ and the set of locations in the domain of H is denoted by $\mathit{ldom}(H)$. We say that a location ℓ (resp. a variable E) is *allocated* in the heap (S, H) or that (S, H) allocates ℓ (resp. E) iff ℓ (resp. $S(E)$) belongs to $\mathit{ldom}(H)$.

The set of heaps satisfying a formula φ is defined by the relation $(S, H) \models \varphi$ given in Figure 6.2. Note that a heap satisfying a predicate $P(E, F, \overrightarrow{B})$ should not allocate any variable in $F \cup \overrightarrow{B}$ since these variables are considered not to be a part of its domain. A heap satisfying this property is called *well-formed w.r.t. the atom* $P(E, F, \overrightarrow{B})$. The set of models of a formula φ is denoted by $[\![\varphi]\!]$. Given two formulae φ_1 and φ_2, we say that φ_1 entails φ_2, denoted by $\varphi_1 \Rightarrow \varphi_2$, iff $[\![\varphi_1]\!] \subseteq [\![\varphi_2]\!]$. By an abuse of notation, $\varphi_1 \Rightarrow E = F$ (resp. $\varphi_1 \Rightarrow E \neq F$) denotes the fact that E and F are interpreted to the same location (resp. different locations) in all models of φ_1.

singly linked lists:

$$\mathtt{ls}(E,F) \triangleq \mathit{lemp}(E,F) \vee \big(E \neq F \wedge \exists X_{\mathtt{tl}} : E \mapsto \{(f, X_{\mathtt{tl}})\} * \mathtt{ls}(X_{\mathtt{tl}}, F)\big)$$

lists of acyclic lists:

$$\begin{aligned}\mathtt{nll}(E,F,B) \triangleq \mathit{lemp}(E,F) \vee \big(&E \neq \{F,B\} \wedge \exists X_{\mathtt{tl}}, Z : E \mapsto \{(s, X_{\mathtt{tl}}), (h, Z)\}\\ &* \mathtt{ls}(Z,B) * \mathtt{nll}(X_{\mathtt{tl}}, F, B)\big)\end{aligned}$$

lists of cyclic lists:

$$\begin{aligned}\mathtt{nlcl}(E,F) \triangleq \mathit{lemp}(E,F) \vee \big(&E \neq F \wedge \exists X_{\mathtt{tl}}, Z : E \mapsto \{(s, X_{\mathtt{tl}}), (h, Z)\}\\ &* \circlearrowleft^{1+} \mathtt{ls}[Z] * \mathtt{nlcl}(X_{\mathtt{tl}}, F)\big)\end{aligned}$$

skip lists with three levels:

$$\begin{aligned}\mathtt{skl}_3(E,F) \triangleq \mathit{lemp}(E,F) \vee \big(&E \neq F \wedge \exists X_{\mathtt{tl}}, Z_1, Z_2 :\\ &E \mapsto \{(f_3, X_{\mathtt{tl}}), (f_2, Z_2), (f_1, Z_1)\} *\\ &\mathtt{skl}_1(Z_1, Z_2) * \mathtt{skl}_2(Z_2, X_{\mathtt{tl}}) * \mathtt{skl}_3(X_{\mathtt{tl}}, F)\big)\end{aligned}$$

$$\begin{aligned}\mathtt{skl}_2(E,F) \triangleq \mathit{lemp}(E,F) \vee \big(&E \neq F \wedge \exists X_{\mathtt{tl}}, Z_1 :\\ &E \mapsto \{(f_3, \mathtt{null}), (f_2, X_{\mathtt{tl}}), (f_1, Z_1)\} *\\ &\mathtt{skl}_1(Z_1, X_{\mathtt{tl}}) * \mathtt{skl}_2(X_{\mathtt{tl}}, F)\big)\end{aligned}$$

$$\begin{aligned}\mathtt{skl}_1(E,F) \triangleq \mathit{lemp}(E,F) \vee \big(&E \neq F \wedge \exists X_{\mathtt{tl}} :\\ &E \mapsto \{(f_3, \mathtt{null}), (f_2, \mathtt{null}), (f_1, X_{\mathtt{tl}})\} * \mathtt{skl}_1(X_{\mathtt{tl}}, F)\big)\end{aligned}$$

Figure 6.1.: Examples of inductive definitions ($\mathit{lemp}(E,F) \triangleq E = F \wedge \mathit{emp}$).

6.2. Compositional Entailment Checking

We define a procedure for reducing the problem of checking the validity of an entailment between two formulae to the problem of checking the validity of an entailment between a formula and an atom. We assume that the right-hand side of the entailment is a quantifier-free formula (which usually suffices for checking verification conditions in practice). The reduction can be extended to the general case, but it becomes incomplete.

6.2.1. Overview of the Reduction Procedure

We consider the problem of deciding validity of entailments $\varphi_1 \Rightarrow \varphi_2$ with φ_2 quantifier-free. We assume $pv(\varphi_2) \subseteq pv(\varphi_1)$; otherwise, the entailment is trivially not valid.

The main steps of the reduction are given in Algorithm 6.1. The reduction starts by a normalisation step (described in Section 6.2.2), which adds to each of the two formulae

$$
\begin{array}{lll}
(S,H) \models E = F & \text{iff} & S(E) = S(F) \\
(S,H) \models E \neq F & \text{iff} & S(E) \neq S(F) \\
(S,H) \models \varphi \wedge \psi & \text{iff} & (S,H) \models \varphi \text{ and } (S,H) \models \psi \\
(S,H) \models \mathit{emp} & \text{iff} & \mathit{dom}(H) = \emptyset \\
(S,H) \models E \mapsto \{\rho\} & \text{iff} & \mathit{dom}(H) = \{(S(E), f_i) \mid (f_i, E_i) \in \{\rho\}\} \text{ and} \\
 & & \text{for every pair } (f_i, E_i) \in \{\rho\}, \text{ it holds that} \\
 & & H(S(E), f_i) = S(E_i) \\
(S,H) \models \Sigma_1 * \Sigma_2 & \text{iff} & \text{there exist } H_1, H_2 \text{ s.t.} \\
 & & \mathit{ldom}(H) = \mathit{ldom}(H_1) \uplus \mathit{ldom}(H_2), \\
 & & (S, H_1) \models \Sigma_1, \text{ and } (S, H_2) \models \Sigma_2 \\
(S,H) \models P(E,F,\vec{B}) & \text{iff} & \text{there exists } k \in \mathbb{N} \text{ s.t.} \\
 & & (S,H) \models P^k(E,F,\vec{B}) \text{ and} \\
 & & \mathit{ldom}(H) \cap (\{S(F)\} \cup \{S(B) \mid B \in \vec{B}\}) = \emptyset \\
(S,H) \models P^0(E,F,\vec{B}) & \text{iff} & (S,H) \models E = F \wedge \mathit{emp} \\
(S,H) \models P^{k+1}(E,F,\vec{B}) & \text{iff} & (S,H) \models E \neq \{F\} \cup \vec{B} \wedge \\
 & & \exists X_{\mathtt{tl}} : \Sigma(E, X_{\mathtt{tl}}, \vec{B}) * P^k(X_{\mathtt{tl}}, F, \vec{B}) \\
(S,H) \models \exists X : \varphi & \text{iff} & \exists \ell \in \mathit{Locs} \text{ s.t. } (S[X \leftarrow \ell], H) \models \varphi
\end{array}
$$

Figure 6.2.: The $\models$ relation ($\uplus$ denotes the disjoint union of sets and $S[X \leftarrow \ell]$ denotes the function S' such that $S'(X) = \ell$ and $S'(Y) = S(Y)$ for any $Y \neq X$)

all (dis-)equalities implied by spatial sub-formulae and removes all atoms $P(E, F, \vec{B})$ representing *empty* list segments, i.e. those where $E = F$ occurs in the pure part. The normalisation of a formula outputs *false* iff the input formula is unsatisfiable.

In the second step, the procedure tests the entailment between the pure parts of the normalised formulae. This can be done using any decision procedure for quantifier-free formulae in the first-order theory with equality.

For the spatial parts, the procedure builds a mapping from spatial atoms of φ_2^n to sub-formulae of φ_1^n. Intuitively, the sub-formula $\varphi_1^n[a_2]$ associated to an atom a_2 of φ_2^n, computed by `select`, describes the region of a heap modelled by φ_1^n that should satisfy a_2. For predicate atoms $a_2 = P_2(E, F, \vec{B})$, `select` is called (in the second loop) only if there exists a model of φ_1^n where the heap region that should satisfy a_2 is non-empty, i.e. $E = F$ does not occur in φ_1^n. In this case, `select` does also check that for any model of φ_1^n, the sub-heap corresponding to the atoms in $\varphi_1^n[a_2]$ is well-formed w.r.t. a_2 (see Section 6.2.3). This is needed since all heaps described by a_2 are well-formed.

Note that in the well-formedness check above, one cannot speak about $\varphi_1^n[a_2]$ alone. This is because without the rest of φ_1^n, $\varphi_1^n[a_2]$ may have models which are not well-formed w.r.t. a_2 even if the sub-heap corresponding to $\varphi_1^n[a_2]$ is well-formed for any model of φ_1^n. For example, let $\varphi_1^n = \mathtt{ls}(x, y) * \mathtt{ls}(y, z) * z \mapsto \{(f, t)\}$, $a_2 = \mathtt{ls}(x, z)$, and $\varphi_1^n[a_2] = \mathtt{ls}(x, y) * \mathtt{ls}(y, z)$. If we consider only models of φ_1^n, the sub-heaps corresponding to $\varphi_1^n[a_2]$ are all well-formed w.r.t. a_2, i.e. the location bound to z is not allocated in these

Algorithm 6.1: Compositional entailment checking of $\varphi_1 \Rightarrow \varphi_2$ ($\prec$ is any total order compatible with $\prec^*_{\mathbb{P}}$)

```
1  $\varphi_1^n \leftarrow$ norm($\varphi_1$); $\varphi_2^n \leftarrow$ norm($\varphi_2$);           // normalisation
2  if $\varphi_1^n = \mathit{false}$ then return true;
3  if $\varphi_2^n = \mathit{false}$ then return false;
4  if $\mathit{pure}(\varphi_1^n) \not\Rightarrow \mathit{pure}(\varphi_2^n)$ then return false ;          // pure parts
   // shape parts
5  foreach points-to atom $a_2$ in $\varphi_2^n$ do                       // points-to atoms
6      $\varphi_1^n[a_2] \leftarrow$ select($\varphi_1^n, a_2$);
7      if $\varphi_1^n[a_2] \not\Rightarrow a_2$ then return false;
8  for $P_2 \leftarrow \max_{\prec}(\mathbb{P})$ downto $\min_{\prec}(\mathbb{P})$ do               // predicate atoms
9      foreach $a_2 = P_2(E, F, \overrightarrow{B})$ in $\varphi_2^n$ s.t. $\mathit{pure}(\varphi_1^n) \not\Rightarrow E = F$ do
10         $\varphi_1^n[a_2] \leftarrow$ select($\varphi_1^n, a_2$);
11         if $\varphi_1^n[a_2] \not\Rightarrow_{sh} a_2$ then return false;
12 return isMarked($\varphi_1^n$);
```

sub-heaps. However, $\varphi_1^n[a_2]$ alone has lasso-shaped models where the location bound to z is allocated on the path between x and y.

Once $\varphi_1^n[a_2]$ is obtained, one needs to check that all sub-heaps modelled by $\varphi_1^n[a_2]$ are also models of a_2. For points-to atoms a_2, this boils down to a syntactic identity (modulo some renaming given by the equalities in the pure part of φ_1^n). For predicate atoms a_2, a special entailment operator $\Rightarrow_{sh}$ (defined in Section 6.2.5) is used. We cannot use the usual entailment $\Rightarrow$ since, as we have seen in the example above, $\varphi_1^n[a_2]$ may have models which are not sub-heaps of models of φ_1^n. Thus, $\varphi_1^n[a_2] \Rightarrow_{sh} a_2$ holds iff all models of $\varphi_1^n[a_2]$, which are well-formed w.r.t. a_2, are also models of a_2.

If there exists an atom a_2 of φ_2^n that is not entailed by the associated sub-formula, then $\varphi_1 \Rightarrow \varphi_2$ is not valid. By the semantics of the separating conjunction, the sub-formulae of φ_1^n associated with two different atoms of φ_2^n must not share spatial atoms. In order to avoid such a scenario, the spatial atoms obtained from each application of `select` are marked and cannot be reused in the future. Note that the mapping is built by enumerating the atoms of φ_2^n in a particular order: first, the points-to atoms and then the inductive predicates, in a decreasing order w.r.t. $\prec_{\mathbb{P}}$. This is important for the completeness of the procedure (see Section 6.2.3).

The procedure `select` is described in Section 6.2.3. It returns *emp* if the construction of the sub-formula of φ_1^n associated with the input atom fails (this implies that also the entailment $\varphi_1 \Rightarrow \varphi_2$ is not valid). If all entailments between formulae and atoms are valid, then $\varphi_1 \Rightarrow \varphi_2$ holds provided that all spatial atoms of φ_1^n are marked (tested by `isMarked`). In Section 6.2.5, we introduce a procedure for checking entailments between a formula and a spatial atom.

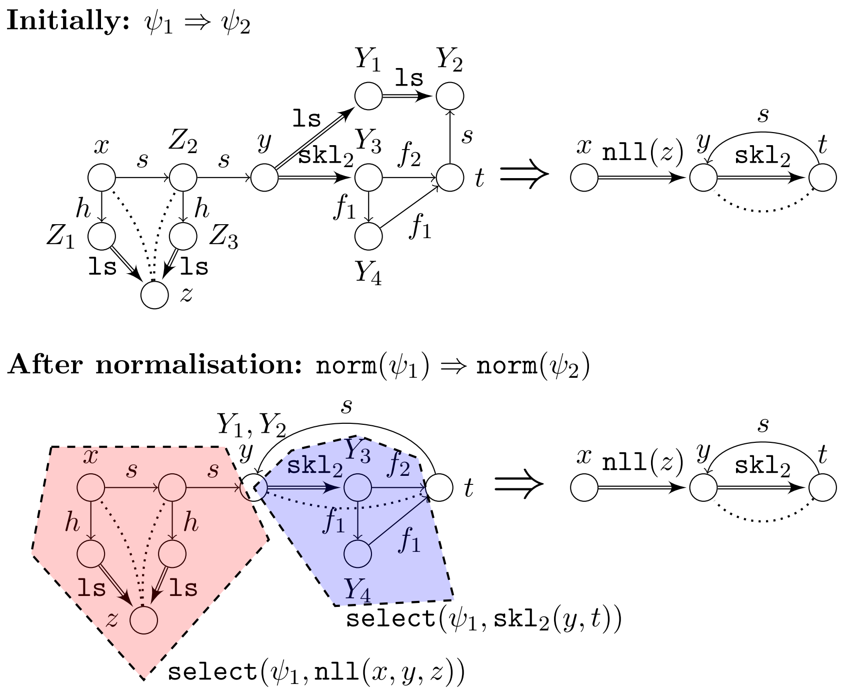

Initially: $\psi_1 \Rightarrow \psi_2$
x s Z_2 s y
h h
Z_1 Z_3
ls ls
z
Y_1 ls Y_2
ls
Y_3 skl$_2$ f_2 s t
f_1 f_1
Y_4
$\Rightarrow$
x nll(z) y s skl$_2$ t
After normalisation: $\texttt{norm}(\psi_1) \Rightarrow \texttt{norm}(\psi_2)$
Y_1, Y_2 s
x s s y Y_3 skl$_2$ f_2 t
h h f_1 f_1
ls ls z
Y_4
select$(\psi_1, \texttt{skl}_2(y, t))$
select$(\psi_1, \texttt{nll}(x, y, z))$
$\Rightarrow$
x nll(z) y s skl$_2$ t

Figure 6.3.: An example of applying compositional entailment checking. Points-to edges are represented by simple lines, predicate edges by double lines, and disequality edges by dotted lines. For readability, we omit some of the labelling with existentially-quantified variables and some of the disequality edges in the normalised graphs.

Graph representations. Some of the sub-procedures mentioned in the previous work on a graph representation of the input formulae, called *SL graphs* (which are different from the Gaifman graphs of Section 6.1). Thus, a formula φ is represented by a directed graph $G[\varphi]$ where each node represents a maximal set of variables equal w.r.t. the pure part of φ, and each edge represents a disequality $E \neq F$ or a spatial atom. Every node n is labelled by the set of variables $\texttt{Var}(n)$ it represents; for every variable E, $\texttt{Node}(E)$ denotes the node n such that $E \in \texttt{Var}(n)$. Next, (1) a disequality $E \neq F$ is represented by an undirected edge from $\texttt{Node}(E)$ to $\texttt{Node}(F)$, (2) a spatial atom $E \mapsto \{(f_1, E_1), \ldots, (f_n, E_n)\}$ is represented by n directed edges from $\texttt{Node}(E)$ to $\texttt{Node}(E_i)$ labelled by f_i for each $1 \leq i \leq n$, and (3) a spatial atom $P(E, F, \vec{B})$ is represented by a directed edge from $\texttt{Node}(E)$ to $\texttt{Node}(F)$ labelled by $P(\vec{B})$. Edges are referred to as disequality, points-to, or predicate edges, depending on the atom they represent. For simplicity, we may say that the graph representation of a formula is simply a formula.

$$F(\Pi) \triangleq \bigwedge_{E=F\in\Pi} [E=F] \;\wedge\; \bigwedge_{E\neq F\in\Pi} \neg[E=F]$$

$$F(\Sigma) \triangleq \bigwedge_{a=E\mapsto\{\rho\}\in\Sigma} [E,a] \;\wedge\; \bigwedge_{a=P(E,F,\vec{B})\in\Sigma} [E,a]\oplus[E=F]$$

$$F_= \triangleq \bigwedge_{\substack{E_1,E_2,E_3\\ \text{variables in } \varphi}} \left(\begin{array}{l} [E_1=E_1] \;\wedge\; \big([E_1=E_2] \Leftrightarrow [E_2=E_1]\big) \;\wedge \\ \Big(\big([E_1=E_2]\wedge[E_2=E_3]\big) \Rightarrow [E_1=E_3]\Big) \end{array} \right)$$

$$F_* \triangleq \bigwedge_{\substack{E,F \text{ variables in } \varphi\\ a\neq a' \text{ atoms in } \Sigma}} \big([E=F]\wedge[E,a]\big) \Rightarrow \neg[F,a']$$

Figure 6.4.: Definition of the components of BoolAbs$[\varphi]$ ($\oplus$ denotes xor)

Running example. In the following, we use as a running example the entailment $\psi_1 \Rightarrow \psi_2$ between the following formulae:

$$\begin{aligned} \psi_1 \equiv\ & \exists Y_1,Y_2,Y_3,Y_4,Z_1,Z_2,Z_3 : x\neq z \wedge Z_2 \neq z \wedge x \mapsto \{(s,Z_2),(h,Z_1)\} * \\ & Z_2 \mapsto \{(s,y),(h,Z_3)\} * \mathtt{ls}(Z_1,z) * \mathtt{ls}(Z_3,z) * \mathtt{ls}(y,Y_1) * \\ & \mathtt{skl}_2(y,Y_3) * \mathtt{ls}(Y_1,Y_2) * Y_3 \mapsto \{(f_2,t),(f_1,Y_4)\} * t \mapsto \{(s,Y_2)\} * \\ & Y_4 \mapsto \{(f_2,\mathtt{null}),(f_1,t)\} \end{aligned} \tag{6.3}$$

$$\psi_2 \equiv y \neq t \wedge \mathtt{nll}(x,y,z) * \mathtt{skl}_2(y,t) * t \mapsto \{(s,y)\} \tag{6.4}$$

The graph representations of these formulae are drawn in the top part of Figure 6.3.

6.2.2. Normalisation

To infer the implicit (dis-)equalities in a formula, we adapt the Boolean abstraction proposed in [ESS13] for our logic. Therefore, given a formula φ, we define an equisatisfiable Boolean formula BoolAbs$[\varphi]$ in CNF over a set of Boolean variables containing the Boolean variable $[E=F]$ for every two variables E and F occuring in φ and the Boolean variable $[E,a]$ for every variable E and spatial atom a of the form $E\mapsto\{\rho\}$ or $P(E,F,\vec{B})$ in φ. The variable $[E=F]$ denotes the equality between E and F, while $[E,a]$ denotes the fact that the atom a describes a heap where E is allocated.

Given $\varphi \triangleq \exists\vec{X} : \Pi\wedge\Sigma$, BoolAbs$[\varphi] \triangleq F(\Pi)\wedge F(\Sigma)\wedge F_= \wedge F_*$ where the components of BoolAbs$[\varphi]$, defined in Figure 6.4, intuitively mean the following: $F(\Pi)$ and $F(\Sigma)$ encode the atoms of φ, $F_=$ encodes reflexivity, symmetry, and transitivity of equality, and F_* encodes the semantics of the separating conjunction.

For the formula ψ_1 in our running example (Equation 6.3), $\mathsf{BoolAbs}[\psi_1]$ is a conjunction of several formulae including:

1. $[y, \mathtt{skl}_2(y, Y_3)] \oplus [y = Y_3]$, which encodes the atom $\mathtt{skl}_2(y, Y_3)$,
2. $[Y_3, Y_3 \mapsto \{(f_2, t), (f_1, Y_4)\}]$ and $[t, t \mapsto \{(s, Y_2)\}]$, encoding points-to atoms, of ψ_1,
3. $\big([t = y] \wedge [t, t \mapsto \{(s, Y_2)\}]\big) \Rightarrow \neg[y, \mathtt{skl}_2(y, Y_3)]$, which encodes the separating conjunction between $t \mapsto \{(s, Y_2)\}$ and $\mathtt{skl}_2(y, Y_3)$,
4. $\big([t = Y_3] \wedge [t, t \mapsto \{(s, Y_2)\}]\big) \Rightarrow \neg[Y_3, Y_3 \mapsto \{(f_2, t), (f_1, Y_4)\}]$, which encodes the separating conjunction between $t \mapsto \{(s, Y_2)\}$ and $Y_3 \mapsto \{(f_2, t), (f_1, Y_4)\}$.

Proposition 6.1. *Let φ be a formula. Then,* $\mathsf{BoolAbs}[\varphi]$ *is equisatisfiable with φ, and for any variables E and F of φ,* $\mathsf{BoolAbs}[\varphi] \Rightarrow [E = F]$ *(resp.* $\mathsf{BoolAbs}[\varphi] \Rightarrow \neg[E = F]$*) iff $\varphi \Rightarrow E = F$ (resp. $\varphi \Rightarrow E \neq F$).*

Example 6.1. *It holds that* $\mathsf{BoolAbs}[\psi_1] \Rightarrow \neg[y = t]$. *This is a consequence of the following propositional reasoning. From the encoding of the points-to atoms from 2, the formula from 4, and* modus tollens*, we infer $\neg[t = Y_3]$. $F_=$ contains the formula*

$$\big([t = y] \wedge [y = Y_3]\big) \Rightarrow [t = Y_3]. \tag{6.5}$$

Because $\neg[t = Y_3]$, when we apply modus tollens *on the previous formula, we obtain the formula*

$$\neg[t = y] \vee \neg[y = Y_3]. \tag{6.6}$$

The xor in 1 is equivalent to the following formula:

$$([y, \mathtt{skl}_2(y, Y_3)] \vee [y = Y_3]) \wedge (\neg[y, \mathtt{skl}_2(y, Y_3)] \vee \neg[y = Y_3]). \tag{6.7}$$

Further, from $[t, t \mapsto \{(s, Y_2)\}]$ and the formula from 3, we infer that

$$\neg[t = y] \vee \neg[y, \mathtt{skl}_2(y, Y_3)]. \tag{6.8}$$

Using resolution on the clause in Equation 6.6 and the first clause of the formula in Equation 6.7, we obtain

$$\neg[t = y] \vee [y, \mathtt{skl}_2(y, Y_3)], \tag{6.9}$$

and using resolution on the just obtained clause in Equation 6.9 and the clause in Equation 6.8, we finally infer $\neg[t = y]$. $\square$

If $\mathsf{BoolAbs}[\varphi]$ is unsatisfiable, then the output of $\mathtt{norm}(\varphi)$ is *false*. Otherwise, the output of $\mathtt{norm}(\varphi)$ is the formula φ' obtained from φ by (1) adding all (dis-)equalities $E = F$ (resp. $E \neq F$) such that $[E = F]$ (resp. $\neg[E = F]$) is implied by $\mathsf{BoolAbs}[\varphi]$ and (2) removing all predicates $P(E, F, \vec{B})$ such that $E = F$ occurs in the pure part, creating formulae φ_1^n and φ_2^n. For example, the normalisations of ψ_1 and ψ_2 are given in the bottom part of Figure 6.3. Note that the $\mathtt{ls}$ atoms reachable from y are removed because $\mathsf{BoolAbs}[\psi_1] \Rightarrow [y = Y_1]$ and $\mathsf{BoolAbs}[\psi_1] \Rightarrow [Y_1 = Y_2]$, as justified in the following example.

Example 6.2. *We show that* $\mathsf{BoolAbs}[\psi_1] \Rightarrow [y = Y_1]$*. We start with the observation that* F_* *contains the following formula:*

$$\big([Y_3 = y] \wedge [Y_3, Y_3 \mapsto \{(f_2, t), (f_1, Y_4)\}]\big) \Rightarrow \neg[y, \mathtt{ls}(y, Y_1)]. \tag{6.10}$$

Because the encoding of the points-to $[Y_3, Y_3 \mapsto \{(f_2, t), (f_1, Y_4)\}]$ *holds, this implies that*

$$[Y_3 = y] \Rightarrow \neg[y, \mathtt{ls}(y, Y_1)]. \tag{6.11}$$

From $F(\Sigma)$*, we have that* $[y, \mathtt{skl}_2(y, Y_3)] \oplus [y = Y_3]$*, from which we infer (with the help of* $F_=$ *containing* $[y = Y_3] \iff [Y_3 = y]$*) that*

$$\neg[Y_3 = y] \Rightarrow [y, \mathtt{skl}_2(y, Y_3)]. \tag{6.12}$$

Further, F_* *contains the following formula:*

$$\big([y = y] \wedge \big[y, \mathtt{skl}_2(y, Y_3)]\big) \Rightarrow \neg[y, \mathtt{ls}(y, Y_1)], \tag{6.13}$$

from which, together with $[y = y]$ *from* $F_=$ *and Equation 6.12, we infer that*

$$\neg[Y_3 = y] \Rightarrow \neg[y, \mathtt{ls}(y, Y_1)]. \tag{6.14}$$

Resolution on the clauses in Equations 6.11 and 6.14 gives us $\neg[y, \mathtt{ls}(y, Y_1)]$*, and from the formula* $[y, \mathtt{ls}(y, Y_1)] \oplus [y = Y_1]$ *contained in* $F(\Sigma)$ *we finally deduce that* $[y = Y_1]$*. Similar reasoning can be applied to deduce that* $[Y_1 = Y_2]$ *is also implied by* $\mathsf{BoolAbs}[\varphi]$*.* □

The following result is important for the completeness of the `select` procedure.

Proposition 6.2. *Let* $\mathtt{norm}(\varphi)$ *be the result of the normalisation of a formula* φ*. For any two* distinct *nodes* n *and* n' *in the SL graph of* $\mathtt{norm}(\varphi)$*, there cannot exist two* disjoint *sets of atoms* A *and* A' *in* $\mathtt{norm}(\varphi)$ *such that both* A *and* A' *represent paths between* n *and* n'*.*

If we assume, for the sake of contradiction, that $\mathtt{norm}(\varphi)$ contains two such sets of atoms, then, by the semantics of the separating conjunction, it needs to holds that one of the paths is empty, so that $\varphi \Rightarrow E = F$ where E and F label n and n' respectively. Therefore, $\mathtt{norm}(\varphi)$ does not include all equalities implied by φ, which contradicts its definition.

6.2.3. Selection of Spatial Atoms

Points-to atoms. Let $\varphi_1 \triangleq \exists \overrightarrow{X} : \Pi_1 \wedge \Sigma_1$ be a normalised formula. The procedure $\mathtt{select}(\varphi_1, E_2 \mapsto \{\rho_2\})$ outputs the sub-formula $\exists \overrightarrow{X} : \Pi_1 \wedge E_1 \mapsto \{\rho_1\}$ such that $E_1 = E_2$ occurs in Π_1 if it exists, or *emp* otherwise. The procedure `select` is called only if φ_1 is satisfiable and consequently, φ_1 cannot contain two different atoms $E_1 \mapsto \{\rho_1\}$ and $E_1' \mapsto \{\rho_1'\}$ such that $E_1 = E_1' = E_2$. Also, if there exists no such points-to atom, then $\varphi_1 \Rightarrow \varphi_2$ is not valid. Indeed, since φ_2 does not contain existentially quantified variables, a points-to atom in φ_2 could be entailed only by a points-to atom in φ_1.

In the running example, $\mathtt{select}(\psi_1, t \mapsto \{(s, y)\}) = \exists Y_2 : y = Y_2 \wedge \ldots \wedge t \mapsto \{(s, Y_2)\}$ (we have omitted some existential variables and pure atoms).

Predicate atoms. Given an atom $a_2 = P_2(E_2, F_2, \overrightarrow{B_2})$, $\mathtt{select}(\varphi_1, a_2)$ builds a sub-graph G' of $G[\varphi_1]$, and then it checks whether the sub-heaps described by G' are well-formed w.r.t. a_2. If this is not true or if G' is empty, then it outputs *emp*. Otherwise, it outputs the formula $\exists \overrightarrow{X} : \Pi_1 \wedge \Sigma'$ where Σ' consists of all atoms represented by edges of the sub-graph G'. Let $\mathtt{Dangling}[a_2] = \mathtt{Node}(F_2) \cup \{\mathtt{Node}(B) \mid B \in \overrightarrow{B_2}\}$.

The sub-graph G' is defined as the union of all paths of $G[\varphi_1]$ that (1) consist of edges labelled by fields in $\mathbb{F}^*_{\mapsto}(P_2)$ or predicates Q with $P_2 \prec^*_{\mathbb{P}} Q$, (2) start in the node labelled by E_2, and (3) end either in a node from $\mathtt{Dangling}[a_2]$ or in a cycle, in which case they must not traverse nodes in $\mathtt{Dangling}[a_2]$. The paths in G' that end in a node from $\mathtt{Dangling}[a_2]$ must not traverse other nodes from $\mathtt{Dangling}[a_2]$. Therefore, G' does not contain edges that start in a node from $\mathtt{Dangling}[a_2]$. The instances of G' for $\mathtt{select}(\psi_1, \mathtt{nll}(x, y, z))$ and $\mathtt{select}(\psi_1, \mathtt{skl}_2(y, t))$ are highlighted in the bottom part of Figure 6.3.

Next, the procedure $\mathtt{select}$ checks that in every model of φ_1, the sub-heap described by G' is well-formed w.r.t. a_2. Intuitively, this means that all cycles in the sub-heap are explicitly described in the inductive definition of P_2. For example, if $\varphi_1 = \mathtt{ls}(x, y) * \mathtt{ls}(y, z)$ and $\varphi_2 = a_2 = \mathtt{ls}(x, z)$, then the graph G' corresponds to the entire formula φ_1 and it may have lasso-shaped models (z may belong to the path between x and y) that are not well-formed w.r.t. $\mathtt{ls}(x, z)$ (whose inductive definition describes only acyclic heaps). Therefore, the procedure $\mathtt{select}$ returns *emp*, which proves that the entailment $\varphi_1 \Rightarrow \varphi_2$ does not hold. For our running example, for any model of ψ_1, in the sub-heap modelled by the graph $\mathtt{select}(\psi_1, \mathtt{skl}_2(y, t))$ in Figure 6.3, t should not be (1) interpreted as an allocated location in the list segment $\mathtt{skl}_2(y, Y_3)$ or (2) aliased to one of nodes labelled by Y_3 and Y_4.

The well-formedness test is equivalent to the fact that for every variable $V \in \{F_2\} \cup \overrightarrow{B_2}$ and every model of φ_1, the interpretation of V is different from all allocated locations in the sub-heap described by G'. This is in turn equivalent to the fact that for every variable $V \in \{F_2\} \cup \overrightarrow{B_2}$, the two following conditions hold:

1. For every predicate edge e included in G' that does not end in $\mathtt{Node}(V)$, V is allocated in all models of $E \neq F \wedge (\varphi_1 \setminus G')$ where E and F are variables labelling the source and the destination of e, respectively, and $\varphi_1 \setminus G'$ is obtained from φ_1 by deleting all *spatial* atoms represented by edges of G'.
2. For every variable V' labelling the source of a points-to edge of G', $\varphi_1 \Rightarrow V \neq V'$.

The first condition guarantees that V is not interpreted as an allocated location in a list segment described by a predicate edge of G' (this trivially holds for predicate edges ending in $\mathtt{Node}(V)$). If V was not allocated in some model (S, H_1) of $E \neq F \wedge (\varphi_1 \setminus G')$, then one could construct a model (S, H_2) of G' where e would be interpreted to a non-empty list and $S(V)$ would equal an allocated location inside this list. Therefore, there would exist a model of φ_1, defined as the union of (S, H_1) and (S, H_2), in which the heap region described by G' would not be well-formed w.r.t. a_2.

For example, in the graph $\mathtt{select}(\psi_1, \mathtt{skl}_2(y, t))$ in Figure 6.3, t is not interpreted as an allocated location in the list segment $\mathtt{skl}_2(y, Y_3)$ since t is allocated (due to the atom $t \mapsto \{(s, Y_2)\}$) in all models of $y \neq Y_3 \wedge (\psi_1 \setminus \mathtt{select}(\psi_1, \mathtt{skl}_2(y, t)))$.

To check that variables are allocated, we use the following property: given a formula $\varphi \triangleq \exists \overrightarrow{X} : \Pi \wedge \Sigma$, a variable V is allocated in every model of φ iff $\exists \overrightarrow{X} : \Pi \wedge \Sigma * V \mapsto \{(f, V_1)\}$ is unsatisfiable. Here, we assume that f and V_1 are not used in φ. Note that, by Proposition 6.1, unsatisfiability can be decided using the Boolean abstraction $\mathsf{BoolAbs}$.

The second condition guarantees that V is different from all allocated locations represented by sources of points-to edges in G'. For the subgraph $\texttt{select}(\psi_1, \texttt{nll}(x, y, z))$ in Figure 6.3, the variable z must be different from all existential variables labelling a node which is the source of a points-to edge. These disequalities appear explicitly in the formula. By Proposition 6.1, $\varphi_1 \Rightarrow V \neq V'$ can be again decided using the Boolean abstraction of φ_1.

6.2.4. Soundness and Completeness

The following theorem states that the procedure given in Algorithm 6.1 is sound and complete. The soundness is a direct consequence of the semantics. The completeness is a consequence of Propositions 6.1 and 6.2. In particular, Proposition 6.2 implies that the sub-formula returned by $\texttt{select}(\varphi_1, a_2)$ is the only one that can describe a heap region satisfying a_2.

Theorem 6.1. *Let φ_1 and φ_2 be a pair of formulae such that φ_2 is quantifier-free. Then, it holds that $\varphi_1 \Rightarrow \varphi_2$ iff the procedure in Algorithm 6.1 returns true.*

6.2.5. Checking Entailments between formulae and Predicate Atoms

Given a formula φ and an atom $P(E, F, \overrightarrow{B})$, we define a procedure for checking that $\varphi \Rightarrow_{sh} P(E, F, \overrightarrow{B})$, which works as follows: (1) $G[\varphi]$ is transformed into a tree $\mathcal{T}[\varphi]$ by splitting nodes that have multiple incoming edges, (2) the inductive definition of $P(E, F, \overrightarrow{B})$ is used to define a TA $\mathcal{A}[P]$ such that $\mathcal{T}[\varphi]$ belongs to the language of $\mathcal{A}[P]$ only if $\varphi \Rightarrow_{sh} P(E, F, \overrightarrow{B})$. Notice that we do not require the reverse implication in order to keep the size of $\mathcal{A}[P]$ polynomial in the size of the inductive definition of P. Thus, $\mathcal{A}[P]$ does not recognise the tree representations of all formulae φ such that $\varphi \Rightarrow_{sh} P(E, F, \overrightarrow{B})$. The transformation of graphs into trees is presented in Section 6.3 while the construction of the TA is introduced in Section 6.4. In Section 6.6, we also discuss how to obtain a complete method by generating a TA $\mathcal{A}[P]$ of an exponential size.

6.3. Representing SL Graphs as Trees

We define a canonical representation of SL graphs in the form of trees, which we use for checking $\Rightarrow_{sh}$. In this representation, the disequality edges are ignored because they have been dealt with previously when checking entailment of pure parts.

We start by explaining the main concepts of the tree encoding using the generic labelled graph in Figure 6.5a. We consider a graph G where all nodes are reachable from a distinguished node called *Root* (this property is satisfied by all SL graphs returned

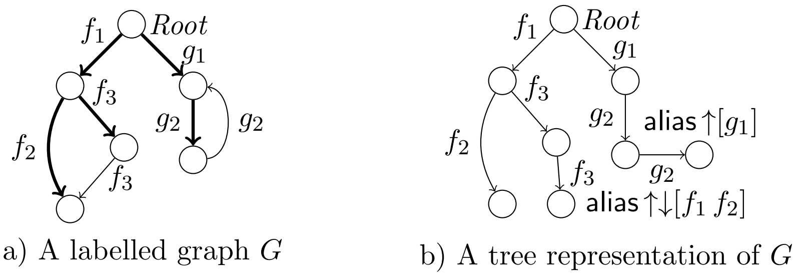


Figure 6.5.: The tree representation of a generic graph

by the `select` procedure). To construct a tree representation of G, we start with its spanning tree (highlighted using bold edges) and proceed with splitting any node with at least two incoming edges, called a *join node*, into several copies, one for each incoming edge not contained in the spanning tree. The obtained tree is given in Figure 6.5b.

In order not to loose any information, the copies of nodes should be labelled with the identity of the original node, which is kept in the spanning tree. However, since the representation does not use node identities, we label every original node with a representation of the path from *Root* to this node in the spanning tree, and we assign every copied node a "routing" label describing how it can reach the original node in the spanning tree. For example, if a node n has the label alias $\uparrow[g_1]$, this denotes the fact that n is a copy of some join node, such that this join node is the lowest ancestor of n that is reachable from *Root* by a path formed of a (non-empty) sequence of g_1 edges in the spanning tree. Further, n labelled by alias $\uparrow\downarrow[f_1\, f_2]$ denotes roughly that (1) the original node is reachable from *Root* by a path formed by a (non-empty) sequence of f_1 edges followed by a (non-empty) sequence of f_2 edges, and (2) the original node can be reached from n by going up in the tree until the first node that is labelled by a prefix of $f_1\, f_2$ and then down until the first node labelled with $f_1\, f_2$. The exact definition of these labels can be found later in this section. In general, a label of the form alias $\uparrow[\ldots]$ will be used when breaking loops while a label of the form alias $\uparrow\downarrow[\ldots]$ will be used when breaking parallel paths between nodes. Moreover, if the original node is labelled by a non-quantified variable, e.g. x, then we will use a label of the form alias $[x]$. This set of labels is enough to obtain a tree representation from SL graphs that can entail a spatial atom from the considered fragment; for arbitrary graphs, this is not the case.

When applying this construction to an SL graph, the most technical part consists of defining the spanning tree. Based on the inductive definition of predicates, we consider a total order on fields $\prec_{\mathbb{F}}$ that is extended to sequences of fields, $\prec_{\mathbb{F}^*}$ in a lexicographic way. Then, the spanning tree is defined by the set of paths labelled by sequences of fields that are minimum according to the order $\prec_{\mathbb{F}^*}$.

Intuitively, the order $\prec_{\mathbb{F}}$ reflects the order in which the unfolding of the inductive definition of P is done: (1) Fields used in the atom $E \mapsto \rho$ of the matrix of P are ordered before fields of any other predicate called by P. (2) Fields appearing in ρ and going "one-step forward" (i.e. occurring in a pair (f, X_{tl})) are ordered before fields going "down" (i.e. occurring in a pair (f, Z) with $Z \in \overrightarrow{Z}$), which are ordered before fields going to the "border" (i.e. occurring in a pair (f, B) with $B \in \overrightarrow{B}$).

Algorithm 6.2: Function `toTree()` encoding SL graphs to trees

```
Input: G : SL graph with all nodes reachable from the node of E,
       $P(E, F, \overrightarrow{B})$ : predicate atom
Output: A labelled tree that encodes G
    // compute the spanning tree
1   $\mathbb{M} :=$ nodeMarking($G, P, E, \prec_{\mathbb{F}^*}$);
    // split nodes of Vars
2   $G' :=$ splitlabelledJoin($G, \mathbb{M}, E, \{F\} \cup \overrightarrow{B}$);
    // split unlabelled join nodes
3   $T :=$ splitJoin($G', \mathbb{M}$);
    // move labels from edges to src nodes
4   $T' :=$ updateLabels($T$);
5   return $T'$;
```

Formally, given a predicate P with the matrix Σ as in Equation 6.2, we split the set $\mathbb{F}_{\mapsto}(P)$ in three disjoint sets: (a) $\mathbb{F}_{\mapsto X_{\mathtt{tl}}}(P)$ is the set of fields f occurring in a pair (f, X_{tl}) of ρ, (b) $\mathbb{F}_{\mapsto \vec{Z}}(P)$ the set of fields f occurring in a pair (f, Z) of ρ with $Z \in \overrightarrow{Z}$, and (c) $\mathbb{F}_{\mapsto \vec{B}}(P)$ the set of fields f occurring in a pair (f, B) of ρ with $B \in \overrightarrow{B}$. Then, we assume that there exists a total order $\prec_{\mathbb{F}}$ on fields such that for all P, P_1, P_2 in $\mathbb{P}$:

$$\begin{gathered}\forall f_1 \in \mathbb{F}_{\mapsto X_{\mathtt{tl}}}(P),\ \forall f_2 \in \mathbb{F}_{\mapsto \vec{Z}}(P),\ \forall f_3 \in \mathbb{F}_{\mapsto \vec{B}}(P) : f_1 \prec_{\mathbb{F}} f_2 \prec_{\mathbb{F}} f_3 \text{ and} \\ (f_1 \in \mathbb{F}_{\mapsto}(P_1) \wedge f_2 \in \mathbb{F}_{\mapsto}(P_2) \wedge f_1 \neq f_2 \wedge P_1 \prec_{\mathbb{P}} P_2) \Rightarrow f_1 \prec_{\mathbb{F}} f_2.\end{gathered} \tag{6.15}$$

For example, if $\mathbb{P} = \{\mathtt{nll}, \mathtt{ls}\}$ or $\mathbb{P} = \{\mathtt{nlcl}, \mathtt{ls}\}$, then $s \prec_{\mathbb{F}} h \prec_{\mathbb{F}} f$; and if $\mathbb{P} = \{\mathtt{skl}_2, \mathtt{skl}_1\}$, then $f_2 \prec_{\mathbb{F}} f_1$. The order $\prec_{\mathbb{F}}$ is extended to a lexicographic order $\prec_{\mathbb{F}^*}$ over sequences in $\mathbb{F}^*$.

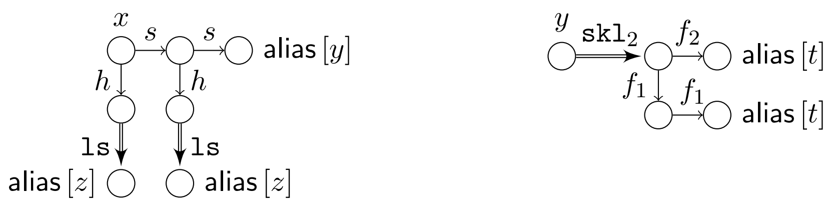
An *f-edge* of an SL graph is a points-to edge labelled by f or a predicate edge labelled by $P(\overrightarrow{N})$ such that the minimum field in $\mathbb{F}_{\mapsto}(P)$ w.r.t. $\prec_{\mathbb{F}}$ is f.

Let G be an SL graph and $P(E, F, \overrightarrow{B})$ an atom for which we want to prove that $G \Rightarrow_{sh} P(E, F, \overrightarrow{B})$. We assume that all nodes of G are reachable from the node *Root* labelled by E, which is ensured when G is constructed by `select`. The tree encoding of G is computed by the procedure `toTree`$(G, P(E, F, \overrightarrow{B}))$ (given in Algorithm 6.2) that consists of four consecutive steps that are presented below.

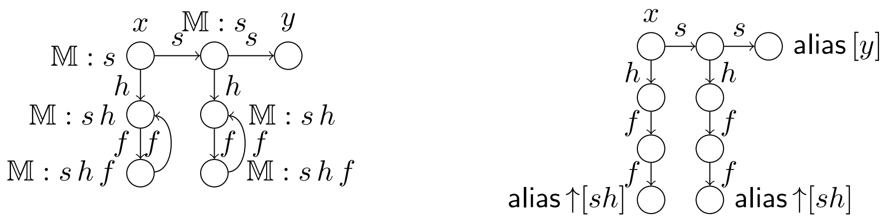
Node marking. First, `toTree` computes a mapping $\mathbb{M}$, called *node marking*, defining the spanning tree of G. Intuitively, for each node n, $\mathbb{M}(n)$ is the sequence of fields labelling a path reaching n from *Root* that is minimal w.r.t. $\prec_{\mathbb{F}^*}$. Formally, let π be a path in G starting in *Root* and consisting of the sequence of edges $e_1\, e_2 \ldots e_n$. The *labelling of* π, denoted by $\mathbb{L}(\pi)$, is the sequence of fields $f_1\, f_2 \ldots f_n$ such that for all i, e_i is an f_i-edge. The marking of a node n in G is defined by

$$\mathbb{M}(n) \triangleq \mathit{Reduce}\big(\min{}_{\prec_{\mathbb{F}}}(\mathbb{F}_{\mapsto}(P))\,.\,\mathbb{L}_{\min}(n)\big), \quad \text{where} \tag{6.16}$$

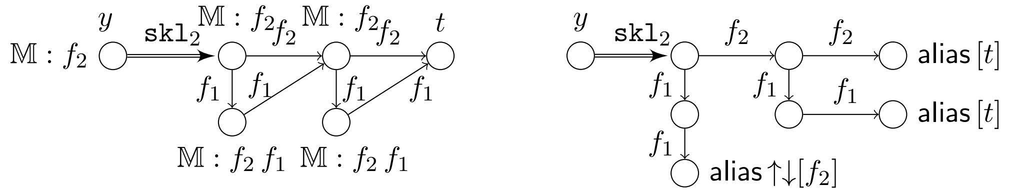
$$\mathbb{L}_{\min}(n) \triangleq \min{}_{\prec_{\mathbb{F}^*}}\big\{\mathbb{L}(\pi) \mid \mathit{Root} \xrightarrow{\pi} n\big\} \tag{6.17}$$



(a) Tree encodings for the selected subgraphs in the bottom left part of Figure 6.3



(b) Tree encodings for graphs satisfying `nlcl`



(c) Tree encodings for graphs satisfying $\texttt{skl}_2$

Figure 6.6.: Tree encodings.

where *Reduce* rewrites the sub-words of the form f^+ to f for any field f, and $Root \xrightarrow{\pi} n$ means that π is a path from the node *Root* to the node n. For technical reasons, we add the minimum field (w.r.t. $\prec_{\mathbb{F}}$) in $\mathbb{F}_{\mapsto}(P)$ at the beginning of all $\mathbb{M}(n)$.

Figures 6.6b–c depict two graphs and the markings of their nodes (for readability, we omit the markings of the nodes labelled by y and t).

Splitting join nodes. The join nodes are split in two consecutive steps, denoted as `splitlabelledJoin` and `splitJoin`, depending on whether they are labelled by variables in $\{E, F\} \cup \overrightarrow{B}$ or not. In both cases, only the edges of the spanning tree are kept in the tree, the other edges are redirected to fresh copies labelled by some alias [..].

For any join node n, the spanning tree edge is the f-edge (m, n) such that it holds that $Reduce(\mathbb{M}(m) . f) = \mathbb{M}(n)$, i.e. (m, n) is at the end of the minimum path leading to n. (For *Root*, no incoming edge is in the spanning tree.)

In `splitlabelledJoin`, a graph G' is obtained from G by replacing any edge (m, n) such that n is labelled by some $V \in \{E, F\} \cup \overrightarrow{B}$ and (m, n) is not in the spanning tree by an edge (m, n') with the same label, where n' is a fresh copy of n labelled by alias $[V]$. Moreover, for uniformity, all (even non-join) nodes labelled by a variable $V \in \{F\} \cup \overrightarrow{B}$

are labelled by $\mathsf{alias}\,[V]$ in G'. Figure 6.6a gives the output graph of `splitlabelledJoin` on the SL graphs returned in our running example by $\texttt{select}(\psi_1, \texttt{nll}(x, y, z))$ and $\texttt{select}(\psi_1, \texttt{skl}_2(y, t))$.

Subsequently, `splitJoin` builds from G' a tree by splitting unlabelled join nodes as follows. Let n be a join node and (m, n) an edge not in the spanning tree of G' (and G). The edge (m, n) is replaced in the tree by an edge (m, n') with the same edge label, where n' is a fresh copy of n labelled by:

- $\mathsf{alias}\uparrow[\mathbb{M}(n)]$ if m is reachable from n in G' and all predecessors of m in G' (by a simple path) marked by $\mathbb{M}(n)$ are also predecessors of n. Intuitively, this label is used to break loops, and it refers to the closest predecessor of n' having the given marking. The use of this labelling is illustrated in Figure 6.6b.

- $\mathsf{alias}\uparrow\downarrow[\mathbb{M}(n)]$ if there is a node p which is a predecessor of m such that all predecessors of m that have a unique successor marked by $\mathbb{M}(n)$ are also predecessors of p, and n is the unique successor of p marked by $\mathbb{M}(n)$. Intuitively, this transformation is used to break multiple paths between p and n as illustrated in Figure 6.6c.[1]

If the relation between n and n' does not satisfy the constraints mentioned above, i.e. the formula does not belong to the considered fragment, the result of `splitJoin` is an error represented by the $\perp$ tree.

At the end of these steps, we obtain a tree with labels on edges (using fields $f \in \mathbb{F}$ or predicates $Q(\overrightarrow{B})$) and labels on nodes of the form $\mathsf{alias}\,[..]$; the root of the tree is labelled by E.

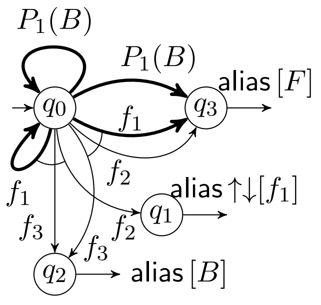
Updating the labels. In the last step, two transformations are done on the tree. First, the labels of predicate edges are changed in order to replace each argument X different from elements of the set $\{F\} \cup \overrightarrow{B}$ by the argument $\mathsf{alias}\uparrow[\mathbb{M}(n)]$ or $\mathsf{alias}\uparrow\downarrow[\mathbb{M}(n)]$, which describes the position of the node n labelled by X w.r.t. the node of G labelled by E. In the case this is not possible, the algorithm returns $\perp$.

Finally, as the generated trees will be tested for membership in the language of a TA which accepts node-labelled trees only, the labels of edges are moved to the labels of their source nodes and concatenated in the order given by $\prec_{\mathbb{F}}$ (predicates in the labels are ordered according to the minimum field in their matrix).

The following property ensures the soundness of the entailment procedure:

Proposition 6.3. *Let $P(E, F, \overrightarrow{B})$ be a predicate atom and G an SL graph. If the procedure* $\texttt{toTree}(G, P(E, F, \overrightarrow{B}))$ *returns $\perp$, then $G \not\Rightarrow P(E, F, \overrightarrow{B})$.*

[1]The combination of up and down arrows in the label corresponds to the need of going up and then down in the resulting tree—whereas in the previous case, it suffices to go up only.



$$\begin{array}{llll}
(1) & q_0 & \hookrightarrow & f_1(q_0), f_2(q_1), f_3(q_2) \\
(2) & q_1 & \hookrightarrow & \mathsf{alias}\uparrow\downarrow[f_1] \\
(3) & q_2 & \hookrightarrow & \mathsf{alias}\,[B] \\
(4) & q_0 & \hookrightarrow & f_1(q_3), f_2(q_3), f_3(q_2) \\
(5) & q_3 & \hookrightarrow & \mathsf{alias}\,[F] \\
(6) & q_0 & \hookrightarrow & P_1(B)(q_0) \\
(7) & q_0 & \hookrightarrow & P_1(B)(q_3)
\end{array}$$

Figure 6.7.: $\mathcal{A}[P_1(E, F, B)]$

6.4. Tree Automata Recognising Tree Encodings of SL Graphs

Next, we proceed to the construction of tree automata $\mathcal{A}[P(E, F, \overrightarrow{B})]$ that recognise tree encodings of SL graphs that entail atoms of the form $P(E, F, \overrightarrow{B})$. We start with an intuitive description on two typical examples and give a full description of the TA construction later. First, to simplify the exposition, we give a modified definition of tree automata for the use in the rest of this chapter (cf. Chapter 2).

Tree automata. A (nondeterministic) *tree automaton* (TA) recognising tree encodings of SL graphs is a tuple $\mathcal{A} = (Q, q_0, \Delta)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, and Δ is a finite set of transitions of the form $(q, a_1 \cdots a_n, q_1 \cdots q_n)$ or (q, a, ϵ), where $n > 0$, $q, q_1, \ldots, q_n \in Q$, a_i is an SL graph edge label (we assume them to be ordered w.r.t. the same ordering of fields $\prec_{\mathbb{F}}$ as for tree encodings), and a is $\mathsf{alias}\uparrow[m]$, $\mathsf{alias}\uparrow\downarrow[m]$, or $\mathsf{alias}\,[V]$ for a marking m and a variable V. We use $q \hookrightarrow a_1(q_1), \ldots, a_n(q_n)$ to denote $(q, a_1 \cdots a_n, q_1 \cdots q_n)$ and $q \hookrightarrow a$ to denote (q, a, ϵ). The set of trees $L(\mathcal{A})$ recognised by $\mathcal{A}$, called the *language* of $\mathcal{A}$, is defined in the same way as in Chapter 2.

Construction of $\mathcal{A}[P(E, F, \overrightarrow{B})]$. The tree automaton $\mathcal{A}[P(E, F, \overrightarrow{B})]$ is constructed by a procedure starting from the inductive definition of P. If P does not call other predicates, the TA simply recognises the tree encodings of the SL graphs that are obtained by "concatenating" a sequence of Gaifman graphs representing the matrix $\Sigma(E, X_{\mathtt{tl}}, \overrightarrow{B})$ and predicate edges $P(E, X_{\mathtt{tl}}, \overrightarrow{B})$. In these sequences, occurrences of the Gaifman graphs representing the matrix and the predicate edges can be mixed in an arbitrary order and in an arbitrary number. Intuitively, this corresponds to a partial unfolding of the predicate P in which there appear concrete segments described by points-to edges as well as (possibly multiple) segments described by predicate edges. Concatenating two Gaifman graphs means that the node labelled by $X_{\mathtt{tl}}$ in the first graph is merged with the node labelled by E in the other graph. We first illustrate this in the following examples and give the formal algorithm later. The TAs for the running examples are given in Section 6.4.3.

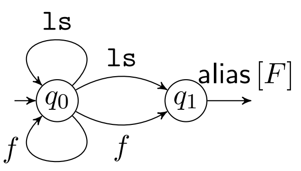


Figure 6.8.: $\mathcal{A}[\mathtt{ls}(E,F)]$

Consider a predicate $P_1(E,F,B)$ that does not invoke any other predicates and that is defined using the matrix $\Sigma_1 \triangleq E \mapsto \{(f_1, X_{\mathtt{tl}}), (f_2, X_{\mathtt{tl}}), (f_3, B)\}$. The tree automaton $\mathcal{A}[P_1(E,F,B)]$ for $P_1(E,F,B)$ has transitions given in Figure 6.7. Transitions 1–3 recognise the tree encoding of the Gaifman graph of Σ_1, assuming the following total order on the fields: $f_1 \prec_{\mathbb{F}} f_2 \prec_{\mathbb{F}} f_3$. Transition 4 is used to distinguish the "last" instance of this tree encoding, which ends in the node labelled by $\mathsf{alias}\,[F]$ accepted by Transition 5. Finally, Transitions 6 and 7 recognise predicate edges labelled by $P_1(B)$. As in the previous case, we distinguish the predicate edge that ends in the node labelled by $\mathsf{alias}\,[F]$.

Note that the TA given above exhibits the simple and generic skeleton of TAs accepting tree encodings of list segments defined in our SL fragment: The initial state q_0 is used in a loop to traverse over an arbitrary number of folded (Transition 6) and unfolded (Transition 1) occurrences of the list segments, and the state q_3 is used to recognise the end of the backbone (Transition 5). The other states (here, q_2) are used to accept alias labels only. The same skeleton can be observed in the TA recognising tree encodings of singly linked lists, which is given in Figure 6.8.

When P invokes other predicates, the automaton recognises tree encodings of concatenations of more general SL graphs, obtained from $Gf[\Sigma]$ by replacing predicate edges with unfoldings of these predicates. On the level of TAs, this operation corresponds to a substitution of transitions labelled by predicates with TAs for the nested predicates. During this substitution, $\mathsf{alias}\,[..]$ labels occurring in the TA for the nested predicate need to be modified. Labels of the form $\mathsf{alias}{\uparrow}[m]$ and $\mathsf{alias}{\uparrow\downarrow}[m]$ are adjusted by prefixing m with the marking of the source state of the transition. On the contrary, labels of the form $\mathsf{alias}\,[V]$ are substituted by the marking of $\mathtt{Node}(V)$ w.r.t. the higher-level matrix.

Let us consider a predicate $P_2(E,F)$ that calls P_1 and that has the matrix defined as $\Sigma_2 \triangleq \exists Z : E \mapsto \{(g_1, X_{\mathtt{tl}}), (g_2, Z)\} \wedge \circlearrowleft^{1+} P_1[Z,E]$. The TA $\mathcal{A}[P_2(E,F)]$ for $P_2(E,F)$ consists of the following transitions:

(1′) $qq_0 \hookrightarrow g_1(qq_0), g_2(q_0)$
transitions of $\mathcal{A}[P_1(E,F,B)]$, where
    $\mathsf{alias}\,[F]$ is substituted by $\mathsf{alias}{\uparrow}[g_1\,g_2]$,
    $\mathsf{alias}\,[B]$ by $\mathsf{alias}{\uparrow}[g_1]$, and
    $\mathsf{alias}{\uparrow\downarrow}[f_1]$ is substituted by $\mathsf{alias}{\uparrow\downarrow}[g_1\,g_2\,f_1]$

(2′) $qq_0 \hookrightarrow g_1(qq_1), g_2(q_0)$
(3′) $qq_1 \hookrightarrow \mathsf{alias}\,[F]$
(4′) $qq_0 \hookrightarrow P_2(qq_0)$
(5′) $qq_0 \hookrightarrow P_2(qq_1)$

(Transition 1′) and the transitions imported (after renaming of the respective labels) from $\mathcal{A}[P_1(E,F,B)]$ describe trees obtained from the tree encoding of $Gf[\Sigma_2]$ by replacing the edge looping in Z with a tree recognised by $\mathcal{A}[P_1(E, F, B)]$. According to $Gf[\Sigma_2]$, the node marking of Z is $g_1\, g_2$, and so the label $\mathsf{alias}\,[F]$ shall be substituted by $\mathsf{alias}{\uparrow}[g_1\, g_2]$, and the marking $\mathsf{alias}{\uparrow\downarrow}[f_1]$ shall be substituted by $\mathsf{alias}{\uparrow\downarrow}[g_1\, g_2\, f_1]$.

In the next sections, we describe our algorithm for construction of tree automata for predicates. First, we start with a description of the basic algorithm for constructing tree automata accepting unfoldings of the predicate where every singly linked list segment (both top-level and nested) is non-empty. Then, we proceed with a description of an extension of the algorithm for list segments that may be empty.

6.4.1. Basic Algorithm for Non-Empty List Segments

Consider the definition of the matrix of the predicate $P(E, F, \vec{B})$ as given in Section 6.1 repeated for the sake of convenience here:

$$\begin{aligned} P(E, F, \vec{B}) \triangleq\, & (E = F \wedge emp) \vee \\ & \big(E \neq \{F\} \cup \vec{B} \wedge \exists X_{\mathtt{tl}} : \Sigma(E, X_{\mathtt{tl}}, \vec{B}) * P(X_{\mathtt{tl}}, F, \vec{B})\big) \end{aligned}$$

where Σ is of the form:

$$\begin{aligned} \Sigma(E, X_{\mathtt{tl}}, \vec{B}) &\triangleq \exists \vec{Z} : E \mapsto \{\rho(\{X_{\mathtt{tl}}\} \cup \vec{V})\} * \Sigma' && \text{where } \vec{V} \subseteq \vec{Z} \cup \vec{B} \text{ and} \\ \Sigma' &::= Q(Z, U, \vec{Y}) \mid \circlearrowleft^{1+} Q[Z, \vec{Y}] \mid \Sigma' * \Sigma' \\ &\quad \text{for } Z \in \vec{Z},\ U \in \vec{Z} \cup \vec{B} \cup \{E, X_{\mathtt{tl}}\},\ \vec{Y} \subseteq \vec{B} \cup \{E, X_{\mathtt{tl}}\}, \text{ and} \\ \circlearrowleft^{1+} Q[Z, \vec{Y}] &\triangleq \exists Z' : \Sigma_Q(Z, Z', \vec{Y}) * Q(Z', Z, \vec{Y}) && \text{where } \Sigma_Q \text{ is the matrix of } Q. \end{aligned}$$

The construction of the automaton $\mathcal{A}[P]$ is described in the following. To ease its presentation, let us suppose that the matrix of P is of the form $\Sigma(E, X_{\mathtt{tl}}, \vec{B}) \triangleq \exists \vec{Z} : E \mapsto \{(f_1, Z_1), \ldots, (f_n, Z_n)\} * \Sigma'$. W.l.o.g. we further assume that $f_1 \prec_{\mathbb{F}} \cdots \prec_{\mathbb{F}} f_n$, i.e. f_1 is the minimum field in $\mathbb{F}_{\mapsto}(P)$.

The construction uses the SL graph of the following formula, which represents two unfoldings of the recursive definition of the predicate:

$$\exists X_{\mathtt{tl}} : \Sigma(E, X_{\mathtt{tl}}, \vec{B}) * \Sigma(X_{\mathtt{tl}}, F, \vec{B}). \tag{6.18}$$

The unfolding is done twice in order to capture all the markings (including the ones of the nodes allocated inside the list segment) that may appear in tree encodings that shall be recognised by $\mathcal{A}[P]$. The graph G is obtained from the SL graph of the formula in Equation 6.18 in such a way that the macro $\circlearrowleft^{1+} Q[Z, \vec{Y}]$ is not expanded but translated into a predicate edge from $\mathtt{Node}(Z)$ to $\mathtt{Node}(Z)$ labelled with $Q(\vec{Y})$.

Then, we get the tree encoding $\mathcal{T}[G]$ of G and check that it is not equal to $\bot$, otherwise we abort the procedure. Notice that the variable $X_{\mathtt{tl}}$ is existentially quantified in the formula, so $\mathcal{T}[G]$ does not use the aliasing relation $\mathsf{alias}\,[X_{\mathtt{tl}}]$. Instead, a node that is a copy of the node labelled with $X_{\mathtt{tl}}$ in G needs to use either the relation $\mathsf{alias}{\uparrow}[f_1]$ or the relation $\mathsf{alias}{\uparrow\downarrow}[f_1]$, because the marking of $\mathtt{Node}(X_{\mathtt{tl}})$ is f_1. Recall also that the nodes of G labelled by parameters or existentially quantified variables are pushed directly in $\mathcal{T}[G]$. So, we overload the notation $\texttt{Node}(Z)$ to denote the node of $\mathcal{T}[G]$ obtained from the node of G labelled by Z.

The construction starts with an empty automaton $\mathcal{A}[P]$. It calls the procedure `buildTACall`, which adds states and transitions to $\mathcal{A}[P]$ to recognise tree encodings of unfoldings of the atom $P(E, F, \overrightarrow{B})$. This procedure is recursive, because it is called for all atoms $Q(U, V, \overrightarrow{W})$ inside the formula in Equation 6.18. The arguments of `buildTACall` are: the predicate called, a mapping σ of the formal parameters of the predicate to an aliasing relation, the states q_0 and q_1 to be used for the source resp. the continuation of the construction, and the marking m_0 of the state q_0. The initial values of these parameters are, in order: P, $\{E \mapsto \mathsf{alias}\,[E], F \mapsto \mathsf{alias}\,[F], \overrightarrow{B \mapsto \mathsf{alias}\,[B]}\}$, fresh states q_0, q_1, and f_1. By $\overrightarrow{B \mapsto \mathsf{alias}\,[B]}$ we denote the set of mappings $\{B \mapsto \mathsf{alias}\,[B] \mid B \in \overrightarrow{B}\}$. The state q_0 is marked as the root state of $\mathcal{A}[P]$.

The procedure `buildTACall` consists of the following four steps.

I. Importing the tree encoding $\mathcal{T}[G]$. In the first step, we construct the *skeleton* of $\mathcal{A}[P]$ by taking $\mathcal{T}[G]$ and transforming it in the following way:

(a) For each node u of $\mathcal{T}[G]$, we create a unique state $q(u)$ in $\mathcal{A}[P]$, except for the nodes $\texttt{Node}(E)$ and $\texttt{Node}(F)$, for which we use the states q_0 and q_1 respectively.

(b) If the node u is labelled in $\mathcal{T}[G]$ with an aliasing relation $r \in \{\mathsf{alias}\,[B] \mid B \in \overrightarrow{B}\} \cup \{\mathsf{alias}\,\triangle[m] \mid \triangle \in \{\uparrow, \uparrow\downarrow\}\}$, where m is a marking, we add the transition

$$q(u) \hookrightarrow \ \beta(r, \sigma, m_0) \tag{6.19}$$

where $\beta(r, \sigma, m_0)$ changes r in the following way: If r is of the form $\mathsf{alias}\,[B]$ for any $B \in \overrightarrow{B}$, the result is $\sigma(B)$. On the other hand, when r is a relation $\mathsf{alias}\,\triangle[m]$ for $\triangle \in \{\uparrow, \uparrow\downarrow\}$, it is changed to $\mathsf{alias}\,\triangle[\mathit{Reduce}(m_0\,.\,m)]$.

(c) If there is a predicate edge from u to v labelled with $Q(\overrightarrow{Y})$, we add the transition

$$q(u) \hookrightarrow \ Q(\beta'(\overrightarrow{Y}, \sigma, m_0))(q(v)). \tag{6.20}$$

where $\beta'(\overrightarrow{Y}, \sigma, m_0)$ changes every Y in $\overrightarrow{Y}$ according to the following rules:

- If Y is an argument of the function call, it is changed to $\sigma(Y)$;
- if Y is an existentially quantified variable in the formula in Equation 6.18, m is the marking of $\texttt{Node}(E)$, and the relation between $\texttt{Node}(E)$ and $\texttt{Node}(Y)$ is $\mathsf{alias}\,\triangle[m]$ for $\triangle \in \{\uparrow, \uparrow\downarrow\}$, we change Y to $\mathsf{alias}\,\triangle[\mathit{Reduce}(m_0\,.\,m)]$;
- otherwise, we abort the procedure.

(d) If the node u is the source of points-to edges $e_1, \ldots, e_k$ labelled with the fields $h_1, \ldots, h_k$ respectively, assuming that $h_1 \prec_{\mathbb{F}} \cdots \prec_{\mathbb{F}} h_k$, and entering nodes $v_1, \ldots, v_k$ in this order, we add the transition

$$q(u) \hookrightarrow h_1(q(v_1)), \ldots, h_k(q(v_k)). \tag{6.21}$$

Note that this rule also creates the backbone transition

$$q_0 \hookrightarrow f_1(q(\texttt{Node}(X_{\texttt{tl}}))), f_2(q(Z_2)), \ldots, f_n(q(Z_n)). \tag{6.22}$$

(e) If the call to `buildTACall` is not nested, we add the transition

$$q_1 \hookrightarrow \ \sigma(F). \tag{6.23}$$

Observe that the created skeleton is able to accept precisely two unfoldings of the predicate P between E and F such that nested predicates are not unfolded.

II. Accepting non empty list segments. Next, we make $\mathcal{A}[P]$ accept an arbitrary number of these unfoldings along the backbone field of the predicate. To do this, we take the initial transition from Equation 6.22 and insert into $\mathcal{A}[P]$ the following transitions:

(a) a transition that accepts exactly one unfolding:

$$q_0 \hookrightarrow f_1(q_1), f_2(q(Z_2)), \ldots, f_n(q(Z_n)). \tag{6.24}$$

(b) a looping transition that allows to insert arbitrarily many unfoldings:

$$q(\texttt{Node}(X_{\texttt{tl}})) \hookrightarrow f_1(q(\texttt{Node}(X_{\texttt{tl}}))), f_2(q(Z_2)), \ldots, f_n(q(Z_n)). \tag{6.25}$$

III. Interleave with predicate edges. We add transitions allowing an arbitrary interleaving of folded and unfolded occurrences of the translated predicate P:

$$q_0 \hookrightarrow P(\overrightarrow{B}[\sigma]])(q(\texttt{Node}(X_{\texttt{tl}}))) \tag{6.26}$$

$$q(\texttt{Node}(X_{\texttt{tl}})) \hookrightarrow P(\overrightarrow{B}[\sigma])(q(\texttt{Node}(X_{\texttt{tl}}))) \tag{6.27}$$

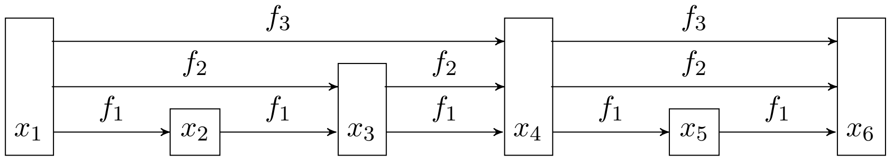
$$q(\texttt{Node}(X_{\texttt{tl}})) \hookrightarrow P(\overrightarrow{B}[\sigma])(q_1). \tag{6.28}$$

IV. Inserting tree automata of nested predicate edges. For each transition inserted in $\mathcal{A}[P]$ of the form:

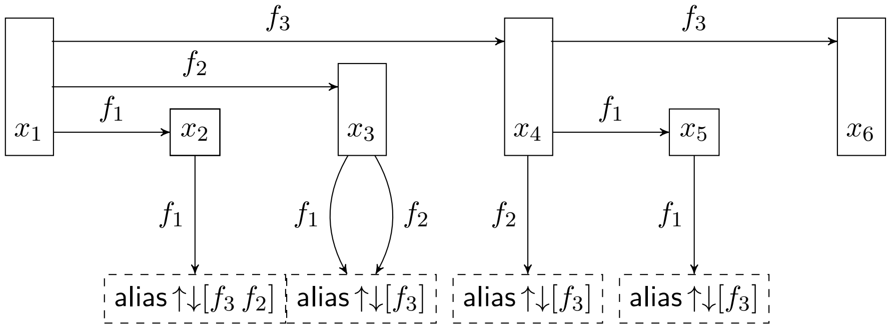
$$q(\texttt{Node}(R)) \hookrightarrow \ Q(\overrightarrow{Y})(q(\texttt{Node}(S))), \tag{6.29}$$

with $Q \neq P$, we call recursively the procedure `builTACall` to insert into $\mathcal{A}[P]$ the automaton for the call of the predicate Q with the parameters $(R, S, \overrightarrow{Y})$. The states created by each call of `builTACall` are new. The procedure `builTACall` is called with the process identifier Q,

- the mapping $\{E \mapsto r_R, F \mapsto r_S, \overrightarrow{B \mapsto r_Y}\}$, where for any $Z \in \{R, S\} \cup \overrightarrow{Y}$:
  - if $Z \in \{E, F\} \cup \overrightarrow{B}$ then r_Z is $\sigma(Z)$,
  - if $Z \in \overrightarrow{Z}$ (the set of existentially quantified variables in P) then r_Z is $\mathsf{alias}\!\uparrow\downarrow[m_Z]$ where m_Z is the marking of $\texttt{Node}(Z)$ in $\mathcal{T}[G]$,



a) The SL graph of a 3-level skip list



b) The tree encoding of the graph in (a)

Figure 6.9.: Illustration of the issue with possibly empty nested list segments. The label of the node accessible from x_5 over f_1 (labelled with alias↑↓[f_3]) reflects the fact that the second-level skip list from the node x_4 to the node x_6 is empty.

- the states $q(\texttt{Node}(R))$ and $q(\texttt{Node}(S))$, and
- the marking $\mathit{Reduce}(m_0 \,.\, m_R)$, where m_R is the marking of $\texttt{Node}(R)$ in $\mathcal{T}[G]$.

The following result states the correctness of the tree automata construction.

Theorem 6.2. *For any predicate atom $P(E, F, \overrightarrow{B})$ and any SL graph G, if the tree generated by* `toTree`$(G, P(E, F, \overrightarrow{B}))$ *is recognised by $\mathcal{A}[P(E, F, \overrightarrow{B})]$, then $G \Rightarrow P(E, F, \overrightarrow{B})$.*

6.4.2. Extending the Basic Algorithm to Possibly Empty Nested List Segments

This extension creates tree automata that can accept such unfoldings of the predicate where nested list segments may be empty. The difficulties this creates are shown in Figure 6.9. The label of the node accessible from x_5 over f_1 (labelled with alias↑↓[f_3]) reflects the fact that the second-level skip list from the node x_4 to the node x_6 is empty. Therefore, when the automaton is traversing the segment between x_4 and x_6, it needs to remember that if the second level list segment leaving x_4 is empty, the label at the end of the first level list segment leaving x_4 is not alias↑↓[$f_3 f_2$] but alias↑↓[f_3]. Note that the top-level list segment predicate is always non-empty; the case when it is empty is dealt with during the normalisation phase (see Section 6.2.2).

Suppose there are nested list segments $R_1, \ldots, R_n$ in the matrix $\Sigma(E, X_{\texttt{tl}}, \overrightarrow{B})$ of the predicate $P(E, F, \overrightarrow{B})$ (note that the predicate of some distinct R_i and R_j can be the same, e.g. $R_i = \texttt{ls}(S, T)$ and $R_j = \texttt{ls}(U, V)$). For every subset S of the set of nested list segments, $S \subseteq \{R_1, \ldots, R_n\}$, we run the procedure in Section 6.4.1 such that we first modify $\Sigma(E, X_{\texttt{tl}}, \overrightarrow{B})$ in such a way that all nested list segments not in S are substituted by their ground case and obtain the automaton $\mathcal{A}^S$. We then obtain the automaton $\mathcal{A}[P(E, F, \overrightarrow{B})]$ by uniting all the automata retrieved in the previous step together and merging their initial states into one.

Formally, given the automata $\mathcal{A}^S = (Q^S, q_0^S, \Delta^S)$ for all $S \subseteq \{R_1, \ldots, R_n\}$ (supposing their sets of states are pairwise disjoint) we create $\mathcal{A}[P(E, F, \overrightarrow{B})] = (Q, q_0, \Delta)$ in the following way.

$$Q = \{q_0\} \cup \bigcup_{S \subseteq \{R_1, \ldots, R_n\}} \left(Q^S \setminus \{q_0^S\}\right) \tag{6.30}$$

$$\Delta = \bigcup_{S \subseteq \{R_1, \ldots, R_n\}} \Delta^S \left[q_0 / q_0^S\right] \tag{6.31}$$

where $\Delta^S \left[q_0 / q_0^S\right]$ denotes the set of transitions Δ^S where every occurrence of q_0^S is substituted with q_0. It is easy to observe that the number of automata $\mathcal{A}^S$ is 2^n; the construction is therefore exponential.

6.4.3. Tree Automata for the Running Example

This section lists tree automata for the predicates from Figure 6.1. The automaton $\mathcal{A}[\texttt{ls}(E, F)]$ contains the following set of transitions (with q_0 being the initial state):

$$\begin{array}{ll} q_0 \hookrightarrow f(q_0) & q_0 \hookrightarrow \texttt{ls}(q_0) \\ q_0 \hookrightarrow f(q_1) & q_0 \hookrightarrow \texttt{ls}(q_1) \\ q_1 \hookrightarrow \mathsf{alias}\,[F] & \end{array}$$

The automaton $\mathcal{A}[\texttt{nll}(G, H, B)]$ contains the following set of transitions (with qq_0 being the initial state):

$$\begin{array}{ll} qq_0 \hookrightarrow s(qq_0), h(q_0) & qq_0 \hookrightarrow s(qq_1), h(q_0) \\ qq_1 \hookrightarrow \mathsf{alias}\,[H] & qq_0 \hookrightarrow \texttt{nll}(B)(qq_0) \\ \text{transitions of } \mathcal{A}[\texttt{ls}(E, B)] & qq_0 \hookrightarrow \texttt{nll}(B)(qq_1) \end{array}$$

The automaton $\mathcal{A}[\texttt{skl}_1(K, L)]$ contains the following set of transitions (p_0 is the initial state):

$$\begin{array}{ll} p_0 \hookrightarrow f_3(p_\perp), f_2(p_\perp), f_1(p_0) & p_0 \hookrightarrow \texttt{skl}_1(p_0) \\ p_0 \hookrightarrow f_3(p_\perp), f_2(p_\perp), f_1(p_1) & p_0 \hookrightarrow \texttt{skl}_1(p_1) \\ p_1 \hookrightarrow \mathsf{alias}\,[L] & p_\perp \hookrightarrow \mathsf{alias}\,[\texttt{NULL}] \end{array}$$

The automaton $\mathcal{A}[\mathtt{skl}_2(M,N)]$ contains the following set of transitions (pp_0 is the initial state):

$$\begin{array}{ll}
pp_0 \hookrightarrow f_3(p_\perp), f_2(pp_0), f_1(p_0) & pp_0 \hookrightarrow \mathtt{skl}_2(pp_0) \\
pp_0 \hookrightarrow f_3(p_\perp), f_2(pp_1), f_1(p_0) & pp_0 \hookrightarrow \mathtt{skl}_2(pp_1) \\
\text{transitions of } \mathcal{A}[\mathtt{skl}_1(K,L)]\text{, where} & pp_1 \hookrightarrow \mathsf{alias}\,[N] \\
\quad \mathsf{alias}\,[L] \text{ is substituted by } \mathsf{alias}\!\uparrow\downarrow\![f_2] &
\end{array}$$

The automaton $\mathcal{A}[\mathtt{skl}_3(P,R)]$ contains the following set of transitions (ppp_0 is the initial state):

$$\begin{array}{ll}
ppp_0 \hookrightarrow f_3(ppp_0), f_2(pp_0), f_1(p_0) & ppp_0 \hookrightarrow \mathtt{skl}_3(ppp_0) \\
ppp_0 \hookrightarrow f_3(ppp_1), f_2(pp_0), f_1(p_0) & ppp_0 \hookrightarrow \mathtt{skl}_3(ppp_1) \\
\text{transitions of } \mathcal{A}[\mathtt{skl}_2(M,N)]\text{, where} & ppp_1 \hookrightarrow \mathsf{alias}\,[R] \\
\quad \mathsf{alias}\,[N] \text{ is substituted by } \mathsf{alias}\!\uparrow\downarrow\![f_3] & \\
\quad \mathsf{alias}\!\uparrow\downarrow\![f_2] \text{ is substituted by } \mathsf{alias}\!\uparrow\downarrow\![f_3\, f_2] &
\end{array}$$

The automaton $\mathcal{A}[\mathtt{nlcl}(S,T)]$ contains the following set of transitions (with qq_0 being the initial state):

$$\begin{array}{ll}
qq_0 \hookrightarrow s(qq_0), h(q_0) & qq_0 \hookrightarrow s(qq_1), h(q_0) \\
qq_1 \hookrightarrow \mathsf{alias}\,[T] & qq_0 \hookrightarrow \mathtt{nlcl}(qq_0) \\
\text{transitions of } \mathcal{A}[\mathtt{ls}(E,F)]\text{, where} & qq_0 \hookrightarrow \mathtt{nlcl}(qq_1) \\
\quad \mathsf{alias}\,[F] \text{ is substituted by } \mathsf{alias}\!\uparrow\![s\, h] &
\end{array}$$

6.5. Extensions

The procedures presented above can be extended to a larger fragment of SL that uses more general inductively defined predicates. In particular, they can be extended to cover finite nestings of singly or doubly linked lists. To describe doubly linked segments, we extend the definition of a predicate from Equation 6.1 to the following:

$$\begin{aligned}
R_{dl}(E,F,P,S,\vec{B}) \triangleq (E = S \wedge F = P \wedge emp) \vee \big(&E \neq S \wedge F \neq P \wedge \\
&\exists X_{\mathtt{tl}} : \Sigma(E, X_{\mathtt{tl}}, P, \vec{B}) * R_{dl}(X_{\mathtt{tl}}, F, E, S, \vec{B})\big)
\end{aligned} \tag{6.32}$$

where Σ is an existentially-quantified matrix of the form:

$$\begin{aligned}
\Sigma(E, X_{\mathtt{tl}}, P, \vec{B}) &\triangleq \exists \vec{Z} : E \mapsto \{\rho(\{X_{\mathtt{tl}}, P\} \cup \vec{V})\} * \Sigma' \quad \text{where } \vec{V} \subseteq \vec{Z} \cup \vec{B} \text{ and} \\
\Sigma' &::= Q(Z, U, \vec{Y}) \mid Q_{dl}(Z, U, Z_p, Z_s, \vec{Y}) \mid \\
&\quad\ \circlearrowleft^{1+} Q[Z, \vec{Y}] \mid \circlearrowleft^{1+} Q_{dl}[Z, \vec{Y}] \mid \Sigma' * \Sigma' \\
&\quad\ \text{for } Z \in \vec{Z};\ U, Z_p, Z_s \in \vec{Z} \cup \vec{B} \cup \{E, X_{\mathtt{tl}}, P\};\ \vec{Y} \subseteq \vec{B} \cup \{E, X_{\mathtt{tl}}, P\}, \\
\circlearrowleft^{1+} Q[Z, \vec{Y}] &\triangleq \exists Z' : \Sigma_Q(Z, Z', \vec{Y}) * Q(Z', Z, \vec{Y}) \\
&\quad\ \text{where } \Sigma_Q \text{ is the matrix of } Q\text{, or} \\
\circlearrowleft^{1+} Q_{dl}[Z, \vec{Y}] &\triangleq \exists Z', Z_p : \Sigma_{Q_{dl}}(Z, Z', Z_p, \vec{Y}) * Q_{dl}(Z', Z_p, Z, Z, \vec{Y}) \\
&\quad\ \text{where } \Sigma_{Q_{dl}} \text{ is the matrix of } Q_{dl}.
\end{aligned}$$

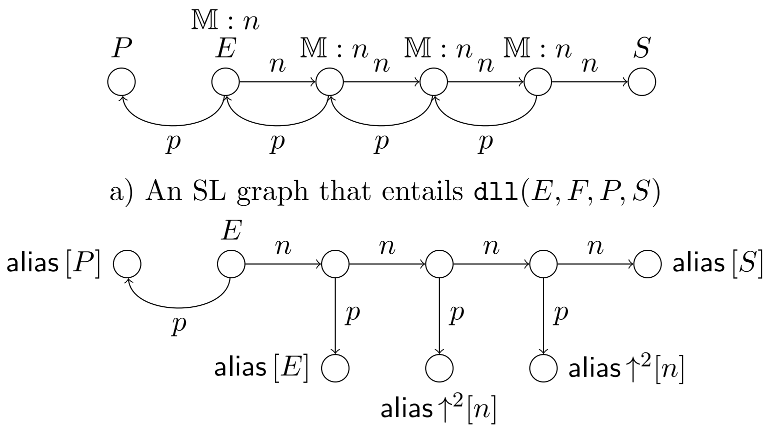


Figure 6.10.: Tree encodings for doubly linked lists

In Equation 6.32, P corresponds to the predecessor of E and S corresponds to the successor of F. For instance, to describe DLL segments between two locations E and F, one can use the predicate

$$\begin{aligned}\texttt{dll}(E,F,P,S) \triangleq (E=S \wedge F=P \wedge emp)\ \vee \big(&E\neq S \wedge F\neq P \wedge \\ &\exists X_{\texttt{tl}} : E \mapsto \{(n,X_{\texttt{tl}}),(p,P)\} * \texttt{dll}(X_{\texttt{tl}},F,E,S)\big).\end{aligned} \tag{6.33}$$

To describe a singly linked list of cyclic doubly linked lists, we may use the following predicate:

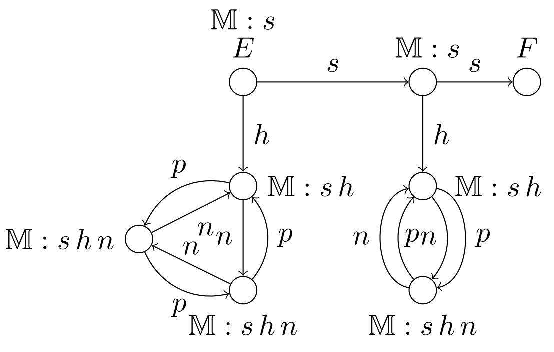
$$\begin{aligned}\texttt{nlcdl}(E,F) \triangleq (E=F\wedge emp) \vee \big(&E\neq F \wedge \\ &\exists X_{\texttt{tl}}, Z : E \mapsto \{(s,X_{\texttt{tl}}),(h,Z)\} * \circlearrowleft^{1+}\texttt{dll}(Z) * \texttt{nlcdl}(X_{\texttt{tl}},F)\big)\end{aligned} \tag{6.34}$$

where $\circlearrowleft^{1+}\texttt{dll}(Z)$ is a macro describing non-empty cyclic doubly linked lists defined by

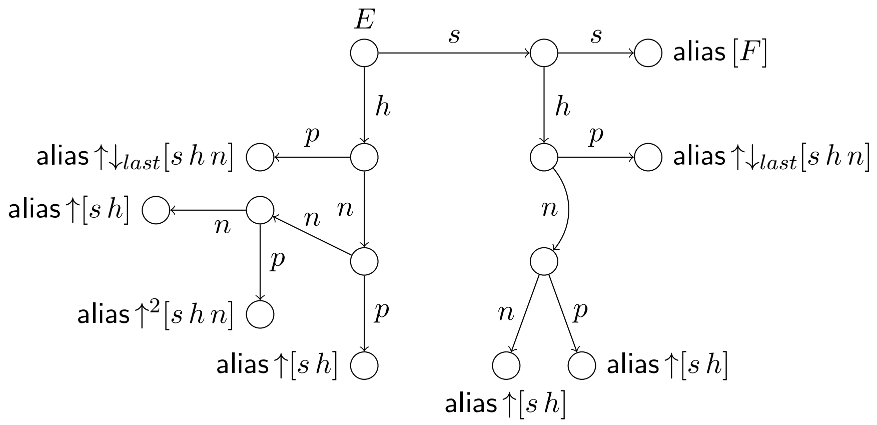
$$\circlearrowleft^{1+}\texttt{dll}[Z] \triangleq \exists Z_1, Z_2 : Z \mapsto \{(n,Z_1),(p,Z_2)\} * \texttt{dll}(Z_1,Z_2,Z,Z). \tag{6.35}$$

Representing SL Graphs as Trees. The splitJoin operation from Section 6.3 is extended with considering the following two more possible labellings: alias$\uparrow^2[\alpha]$ and alias$\uparrow\downarrow_{last}[\alpha]$. If n is a join node in a graph and (m,n) is an edge that is not in its spanning tree, then (m,n) is replaced by the edge (m,n') with the same edge label, such that n' is a fresh copy of n labelled by (in addition to the labellings from Section 6.3)

- alias$\uparrow^2[\mathbb{M}(n)]$ if m is reachable from n, m further reaches n in the spanning tree of the graph and in the spanning tree there is exactly one node marked with $\mathbb{M}(n)$ between m and n. Intuitively, this label is needed to handle inner nodes of doubly linked lists, which have two incoming edges, one from their successor and one from their predecessor (see Figure 6.10).



a) An SL graph that entails `nlcdl`(E, F)



b) The tree encoding of the graph from (a)

Figure 6.11.: Tree encodings for lists of nested cyclic doubly linked lists

- alias $\uparrow\downarrow_{last}[\mathbb{M}(n)]$ if there is a node p that is an ancestor of m (or it is m itself), such that p is also an ancestor of n, and n has no non-alias successors with the marking $\mathbb{M}(n)$. Intuitively, the label is needed for a doubly linked cyclic list to allow referring to the predecessor of the head node of the list (see Figure 6.11).

6.6. Completeness and Complexity

In general, there exist SL graphs that entail $P(E, F, \overrightarrow{B})$ whose tree encodings are *not* recognised by $\mathcal{A}[P(E, F, \overrightarrow{B})]$ created using the algorithm from Section 6.4.1. The models of these SL graphs are nested list segments where inner pointer fields specified by the matrix of P are aliased. For example, the TA for $\texttt{skl}_2$ does not recognise the tree encodings of SL graphs modelled by heaps where $X_{\texttt{tl}}$ and Z_1 are interpreted to the same location.

This issue is dealt with by the algorithm presented in Section 6.4.2. However, the size of the TA created in this way may become exponential in the size of P (defined

Table 6.1.: Average running times for Spen on the benchmarks from [PR11].

Bolognesa	Time [ms]	*Spaguetti*	Time [ms]	*Clones*	Time [ms]
bo-10	352	sp-10	146	cl-01	316
bo-11	386	sp-11	156	cl-02	314
bo-12	385	sp-12	145	cl-03	335
bo-13	394	sp-13	153	cl-04	336
bo-14	483	sp-14	189	cl-05	321
bo-15	562	sp-15	258	cl-06	334
bo-16	424	sp-16	198	cl-07	351
bo-17	510	sp-17	254	cl-08	374
bo-18	503	sp-18	249	cl-09	407
bo-19	516	sp-19	252	cl-10	436
bo-20	522	sp-20	282		

as the number of symbols in the matrices of all Q with $P \prec^*_{\mathbb{P}} Q$), as the construction considers all possible aliasing scenarios of targets of inner pointer fields permitted by the predicate.

For the verification conditions that we have encountered in our experiments, the TAs constructed using the former algorithm are precise enough in the vast majority of the cases. In particular, note that the TAs generated for non-nested predicates, such as the predicates for `ls` and `dll`, are precise. We have, however, implemented even the latter algorithm (which is complete even for nested predicates) and evaluated that it also provides good performance on practical examples (where the number of nestings is given by the use in real-world programs).

In conclusion, the overall complexity of the incomplete semi-decision procedure (where aliases between variables in the definition of a predicate are ignored) runs in polynomial time modulo an oracle for deciding validity of a Boolean formula (needed in the normalisation part of the procedure). The complete decision procedure is exponential in the size of the predicates, which remains acceptable in practice, rather than in the size of the formulae.

6.7. Implementation and Experimental Results

We implemented our decision procedure in a solver called Spen (SeParation logic ENtailment). The tool takes as the input an entailment problem $\varphi_1 \Rightarrow \varphi_2$ (including the definition of the predicates used) encoded in the SmtLib2 format. For non-valid entailments, Spen prints the atom of φ_2 which is not entailed by a sub-formula of φ_1. The tool is based on the MiniSat solver for deciding unsatisfiability of Boolean formulae and the Vata library (described in Chapter 10) as the tree automata backend.

We applied Spen to entailment problems that use various recursive predicates. First, we considered the benchmark provided in [PR11], which uses only the `ls` predicate.

Table 6.2.: Running SPEN on entailments between formulae and atoms.

φ_2	φ_1	Time [ms]	Status	States/Trans. of $\mathcal{A}[\varphi_2]$	Nodes/Edges of $T(Gf[\varphi_1])$
`nll`	`tc1`	344	valid	6/17	7/7
	`tc2`	335	valid		7/7
	`tc3`	319	invalid		6/7
`nlcl`	`tc1`	318	valid	6/15	10/9
	`tc2`	316	valid		7/7
	`tc3`	317	invalid		6/6
$\texttt{skl}_3$	`tc1`	334	valid	80/193	7/7
	`tc2`	349	valid		8/8
	`tc3`	326	invalid		6/6
`dll`	`tc1`	358	valid	9/16	7/7
	`tc2`	324	valid		7/7
	`tc3`	322	invalid		5/5

It consists of three classes of entailment problems called *Spaguetti*, *Bolognesa*, and *Clones*. The first two classes contain 110 problems each (split into 11 groups) generated randomly according to the rules specified in [PR11], whereas the last class contains 100 problems (split into 10 groups) obtained from the verification conditions generated by the tool SMALLFOOT [BCO06]. In all experiments[2], SPEN finished in less than 1 second with the deviation of running times ± 100 ms w.r.t. the ones reported for SELOGER [HIOP13][3], the most efficient tool for deciding entailments of SL formulae with singly linked lists we are aware of (average times for each group are given in Table 6.1).

The TA for the predicate `ls` is quite small, and so the above experiments did not evaluate much the performance of our procedure for checking entailments between formulae and atoms. For a more thorough evaluation, we further considered the experiments listed in Table 6.2 (among which, $\texttt{skl}_3$ required the extension of our approach to a full decision procedure as discussed at the end of Section 6.4). The full benchmark is available with our tool [ELSV14b]. The entailment problems are extracted from verification conditions of operations like adding or deleting an element at the beginning, in the middle, or at the end of various kinds of list segments (see Figure 6.12). Table 6.2 gives for each example the running time, whether the entailment is valid or invalid, and the size of the tree encoding and TA for φ_1 and φ_2, respectively. We find the resulting times quite encouraging.

Moreover, SPEN participated in three divisions of the first competition of separation logic solvers SL-COMP'14 [sl-14]: division FDB_entl containing problems with extended acyclic lists, such as doubly linked lists, nested lists, or skip lists (the results for this

[2]Our experiments were performed on a PC with an Intel Core 2 Duo @2.53 GHz processor and 4 GiB DDR3 @1067 MHz running a virtual machine with Fedora 20 (64-bit).

[3]The times reported for SELOGER in [HIOP13] were obtained on a PC with an Intel Core i5-2467M @1.60 GHz processor and 4 GiB DDR3 @1066 MHz under Windows 7 (64-bit).

$\varphi_2 = \mathtt{nll}(x, y, z)$

$\mathtt{tc1} \triangleq x \mapsto \{(s, u), (h, a)\} * u \mapsto \{(s, y), (h, b)\} * \mathtt{ls}(a, z) * \mathtt{ls}(b, z)$

$\mathtt{tc2} \triangleq \mathtt{nll}(x, u, z) * u \mapsto \{(s, w), (h, a)\} * a \mapsto \{(f, b)\} * \mathtt{ls}(b, z) * \mathtt{nll}(w, y, z)$

$\mathtt{tc3} \triangleq \mathtt{nll}(x, u, z) * u \mapsto \{(s, w), (h, a)\} * a \mapsto \{(f, b)\} * b \mapsto \{(f, a)\} * \mathtt{nll}(w, y, z)$

$\varphi_2 = \mathtt{nlcl}(x, y)$

$\mathtt{tc1} \triangleq x \mapsto \{(s, u), (h, a)\} * a \mapsto \{(f, b)\} * b \mapsto \{(f, a)\} * u \mapsto \{(s, y), (h, c)\} * c \mapsto \{(f, d)\} * \mathtt{ls}(d, c)$

$\mathtt{tc2} \triangleq \mathtt{nlcl}(x, u) * u \mapsto \{(s, v), (h, a)\} * a \mapsto \{(f, b)\} * \mathtt{ls}(b, a) * \mathtt{nlcl}(v, y)$

$\mathtt{tc3} \triangleq \mathtt{nlcl}(x, u) * u \mapsto \{(s, v), (h, a)\} * a \mapsto \{(f, y)\} * \mathtt{nlcl}(v, y)$

$\varphi_2 = \mathtt{skl}_3(x, y)$

$\mathtt{tc1} \triangleq x \mapsto \{(f_1, z), (f_2, z), (f_3, z)\} * z \mapsto \{(f_1, y), (f_2, y), (f_3, y)\}$

$\mathtt{tc2} \triangleq \mathtt{skl}_3(x, z) * z \mapsto \{(f_3, w), (f_2, z_2)(f_1, z_1)\} * \mathtt{skl}_1(z_1, z_2) * \mathtt{skl}_2(z_2, w) * \mathtt{skl}_3(w, y)$

$\mathtt{tc3} \triangleq x \mapsto \{(f_1, w), (f_2, w), (f_3, w)\} * w \mapsto \{(f_1, z), (f_2, w_2), (f_3, z)\} * \mathtt{skl}_2(w_2, z) * \mathtt{skl}_3(z, y)$

$\varphi_2 = \mathtt{dll}(x, y, z, v)$

$\mathtt{tc1} \triangleq x \mapsto \{(n, u), (p, z)\} * u \mapsto \{(n, y), (p, x)\} * y \mapsto \{(n, v), (p, u)\}$

$\mathtt{tc2} \triangleq x \mapsto \{(n, u), (p, z)\} * \mathtt{dll}(u, w, x, y) * y \mapsto \{(n, v), (p, w)$

$\mathtt{tc3} \triangleq x \mapsto \{(n, u), (p, z)\} * \mathtt{dll}(u, w, x, y) * y \mapsto \{(n, v)\}$

Figure 6.12.: Definition of formulae for φ_1 in the experiments

division are in Table 6.3a), and divisions sll0a_entl and sll0a_sat containing problems with singly linked lists (the results for these divisions are in Table 6.3b). The tables contain for each solver the number of problems for which the solver responded incorrectly (column Errors), the number of problems for which it responded correctly (column Solved), the number of problems for which it did not give an answer (column ¬Solved), and the total time of the solver in seconds (column Time). Spen won division FDB_entl with a huge difference, solving all problems in less than a minute; further, notice that Spen is the only tool that correctly answered all problems in this division. In addition to this, Spen was also placed second in both divisions with singly linked lists, where the first placed was won by Asterix, a solver specialised for this particular data structure.

Table 6.3.: Results of SL-COMP'14

a) Results for extended acyclic lists (FDB_entl)

Solver	Errors	Solved	¬Solved	Time
Spen	0	43	0	0.61
Cyclist-SL	0	19	24	141.78
Slide	0	0	43	0.00
Sleek-06	1	31	11	43.65

b) Results for singly linked lists

Solver	sll0a_entl				sll0a_sat			
	Errors	Solved	¬Solved	Time	Errors	Solved	¬Solved	Time
Asterix	0	292	0	2.98	0	110	0	1.06
Spen	0	292	0	7.58	0	110	0	3.27
Sleek-06	0	292	0	14.13	0	110	0	4.99
Cyclist-SL	0	55	237	11.78	55	55	0	0.55

6.8. Conclusion

This chapter presented a novel (semi-)decision procedure for a fragment of SL with inductive predicates describing various forms of lists (singly or doubly linked, nested, circular, with skip links, etc.). The procedure is compositional in that it reduces the given entailment query to a set of simpler queries between a formula and an atom. For solving them, we proposed a novel reduction to testing membership of a tree derived from the formula in the language of a TA derived from a predicate. We implemented the procedure, and our experiments show that it has not only a favourable theoretical complexity, but also efficiently handles practical verification conditions. Moreover, when compared with other tools for deciding separation logic formulae in the first competition of separation logic solvers SL-COMP'14 [sl-14], Spen won the first place in one division (being by several orders of magnitude faster and even more successful in correctly deciding the decision problems), and the second place in two divisions.

In the future, we plan to investigate extensions of our approach to formulae with a more general Boolean structure or using more general inductive definitions. Concerning the latter, we plan to investigate whether some ideas from [IRV14] could be used to extend our decision procedure for entailments between formulae and atoms. From a practical point of view, apart from improving the implementation of our procedure, we plan to integrate it into a complete program analysis framework.

7. Deciding WS1S Formulae Using Nested Antichains

Weak monadic second-order logic of one successor (WS1S) is a powerful, concise, and decidable logic for describing regular properties of finite words. Despite its nonelementary worst case complexity [Mey72], it has been shown useful in numerous applications. Most of the successful applications were due to the tool MONA [EKM98], which implements a finite automata-based decision procedure for WS1S and WS2S (a generalisation of WS1S to finite binary trees). The authors of MONA list a multitude of its diverse applications [KM01], ranging from software and hardware verification through controller synthesis to computational linguistics, and further on. Among more recent applications, verification of pointer programs and deciding related logics [MPQ11, MQ11, IRŠ13, CDNQ12a, ZKR08] can be mentioned, as well as synthesis from regular specifications [HJK10]. MONA is still the standard tool and the most common choice when it comes to deciding WS1S/WS2S. There are other related automata-based tools that are more recent, such as jMosel [TWMS06] for the M2L(Str) logic, and other than automata-based approaches, such as [GK10]. They implement optimisations that allow them to outperform MONA on some benchmarks, however, none provides an evidence of being consistently more efficient. Despite many optimisations implemented in MONA and the other tools, the worst case complexity of the problem sometimes strikes back. Authors of methods using the translation of their problem to WS1S/WS2S are then forced to either find workarounds to circumvent the complexity blowup, such as in [MQ11], or, often restricting the input of their approach, give up translating to WS1S/WS2S altogether [WMK11].

The decision procedure of MONA works with deterministic automata; it uses determinisation extensively and relies on minimisation of deterministic automata to suppress the complexity blow-up. However, the worst case exponential complexity of determinisation often significantly harms the performance of the tool. Recent works on efficient methods for handling nondeterministic automata suggest a way of alleviating this problem, in particular works on efficient testing of language inclusion and universality of finite automata [DR10, WDHR06, ACH$^+$10] and size reduction [BG00, ABH$^+$08] based on a simulation relation. Handling nondeterministic automata using these methods, while avoiding determinisation, has been shown to provide great efficiency improvements in [BHH$^+$08] (abstract regular model checking) and also [HHR$^+$12] (shape analysis). In this chapter, we present a work that makes a major step towards building the entire decision procedure of WS1S on nondeterministic automata using similar techniques. We propose a generalisation of the antichain algorithms of [DR10] that addresses the main bottleneck of the automata-based decision procedure for WS1S, which is also the source of its nonelementary complexity: elimination of alternating quantifiers on the automata level.

More concretely, the automata-based decision procedure translates the input WS1S formula into a finite word automaton such that its language represents exactly all models of the formula. The automaton is built in a bottom-up manner according to the structure of the formula, starting with predefined atomic automata for literals and applying a corresponding automata operation for every logical connective and quantifier $(\wedge, \vee, \neg, \exists)$. The cause of the nonelementary complexity of the procedure can be explained on an example formula of the form $\varphi' = \exists X_m \forall X_{m-1} \ldots \forall X_2 \exists X_1 : \varphi_0$. The universal quantifiers are first replaced by negation and existential quantification, which results in $\varphi = \exists X_m \neg \exists X_{m-1} \ldots \neg \exists X_2 \neg \exists X_1 : \varphi_0$. The algorithm then builds a sequence of automata for the sub-formulae $\varphi_0, \varphi_0^\sharp, \ldots, \varphi_{m-1}, \varphi_{m-1}^\sharp$ of φ where for $0 \leq i < m$, $\varphi_i^\sharp = \exists X_{i+1} : \varphi_i$, and $\varphi_{i+1} = \neg \varphi_i^\sharp$. Every automaton in the sequence is created from the previous one by applying the automata operations corresponding to negation or elimination of the existential quantifier, the latter of which may introduce nondeterminism. Negation applied on a nondeterministic automaton may then yield an exponential blowup: given an automaton for ψ, the automaton for $\neg\psi$ is constructed by the classical automata-theoretic construction consisting of determinisation by the subset construction followed by swapping of the sets of final and non-final states. The subset construction is exponential in the worst case. The worst case complexity of the procedure run on φ is then a tower of exponentials with one level for every quantifier alternation in φ; note that, in general, we cannot do much better—this nonelementary complexity is an inherent property of the problem.

Main ideas of our approach. Our new algorithm for processing alternating quantifiers in the prefix of a formula avoids the explicit determinisation of automata in the classical procedure and significantly reduces the state space explosion associated with it. It is based on a generalisation of the antichain principle used for deciding universality and language inclusion of finite automata [WDHR06, ACH$^+$10]. It generalises the antichain algorithms so that instead of being used to process only one level of the chain of automata, it processes the whole chain of quantifications with i alternations on the fly. This leads to 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). The algorithm uses nested symbolic terms to represent sets of such automata states and a generalised version of antichain subsumption pruning which descends recursively down the structure of the terms while pruning on all its levels.

Our nested antichain algorithm can be—in its current form—used only to 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 optimised algorithm for processing inner negations and alternating quantifiers which are not a part of the quantifier prefix. However, despite this and the fact that our implementation is far less mature than that of MONA, our experimental results still show significant improvements over its performance, especially in terms of generated state space. We consider this a strong indication that using techniques for nondeterministic automata to decide WS1S (and WSkS) is highly promising. There are many more opportunities of improving the decision procedure based on nondeterministic automata, by using techniques such as simulation relations or bisimulation up-to congruence [BP13], and applying them to process not only the quantifier prefix, but all logical connectives of a formula. We consider the work presented in this chapter to be the first step towards a decision procedure for WS1S/WSkS with an entirely different scalability than the current state of the art.

Outline. The structure of this chapter is as folows: We define the logic WS1S in Section 7.1. In Sections 7.2 and 7.3, we introduce finite word automata and describe the classical decision procedure for WS1S based on finite word automata. In Section 7.4, we introduce our method for dealing with alternating quantifiers. Finally, we give an experimental evaluation and conclude the chapter in Sections 7.5 and 7.6.

7.1. WS1S

In this section we give an introduction into the *weak monadic second-order logic of one successor* (WS1S). We introduce only its minimal syntax here, for the full standard syntax and a more thorough introduction, see Section 3.3 in [CDG$^+$07].

WS1S is a monadic second-order logic over the universe of discourse $\mathbb{N}_0$. This means that the logic allows second-order *variables*, usually denoted using upper-case letters $X, Y, \ldots$, that range over finite subsets of $\mathbb{N}_0$, e.g. $X = \{0, 3, 42\}$. Atomic formulae are of the form (i) $X \subseteq Y$, (ii) $\mathrm{Sing}(X)$, (iii) $X = \{0\}$, and (iv) $X = Y + 1$, where X and Y are variables. The atomic formulae are interpreted in turn as (i) standard set inclusion, (ii) the singleton predicate, (iii) X is a singleton containing 0, and (iv) $X = \{x\}$ and $Y = \{y\}$ are singletons and x is the successor of y, i.e. $x = y + 1$. Formulae are built from the atomic formulae using the logical connectives $\wedge, \vee, \neg$, and the quantifier $\exists X$ (for a second-order variable X).

Given a WS1S formula $\varphi(X_1, \ldots, X_n)$ with free variables $X_1, \ldots, X_n$, the assignment $\rho = \{X_1 \mapsto S_1, \ldots, X_n \mapsto S_n\}$, where $S_1, \ldots, S_n$ are finite subsets of $\mathbb{N}_0$, *satisfies* φ, written as $\rho \models \varphi$, if the formula holds when every variable X_i is replaced with its corresponding value $S_i = \rho(X_i)$. We say that φ is *valid*, denoted as $\models \varphi$, if it is satisfied by all assignments of its free variables to finite subsets of $\mathbb{N}_0$. 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).

7.2. Preliminaries and Finite Automata

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. 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 $\mathbb{D} = \{D_1, \dots, D_n\}$, returns the set of all sets $\{d_1, \dots, d_n\}$ obtained by taking one element d_i from every set D_i. Formally,

$$\coprod\mathbb{D} = \Big\{\{d_1, \dots, d_n\} \mid (d_1, \dots, d_n) \in \prod_{i=1}^{n} D_i\Big\} \tag{7.1}$$

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 $\coprod\mathbb{D} = \emptyset$.

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 $\overline{0}$ to denote the symbol which maps all variables from $\mathbb{X}$ to 0, $\overline{0} \in \Sigma_{\mathbb{X}}$.

A (nondeterministic) *finite* (word) *automaton* (abbreviated as NFA in the following) over a set of variables $\mathbb{X}$ 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 Δ 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 NFA $\mathcal{A}$ over $\mathbb{X} = \emptyset$, $\mathcal{A}$ is a unary NFA with the alphabet $\Sigma_{\mathbb{X}} = \{\overline{0}\}$.

A *run* r of $\mathcal{A}$ over a word $w = \tau_1\tau_2 \dots \tau_n \in \Sigma_{\mathbb{X}}^*$ from the state $p \in Q$ to the state $s \in Q$ is a sequence of states $r = q_0q_1 \dots q_n \in Q^+$ 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 $L_{\mathcal{A}}(q) = \{w \mid q \overset{w}{\Longrightarrow} q_f, q_f \in F\}$, while the language of a set of states $S \subseteq Q$ is defined as $L_{\mathcal{A}}(S) = \bigcup_{q \in S} L_{\mathcal{A}}(q)$. When it is clear which NFA $\mathcal{A}$ we refer to, we only write $L(q)$ or $L(S)$. The language of $\mathcal{A}$ is defined as $L(\mathcal{A}) = L_{\mathcal{A}}(I)$. We say that the state q accepts w and that the automaton $\mathcal{A}$ accepts w to express that $w \in L_{\mathcal{A}}(q)$ and $w \in L(\mathcal{A})$ respectively. We call a language $L \subseteq \Sigma_{\mathbb{X}}^*$ *universal* iff $L = \Sigma_{\mathbb{X}}^*$.

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

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

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

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 $\pi_{[X]}(\mathbb{X})$. For a transition relation Δ, the projection of X from Δ, denoted as $\pi_{[X]}(\Delta)$, is the transition relation $\left\{p \xrightarrow{\pi_{[X]}(\tau)} q \mid p \xrightarrow{\tau} q \in \Delta\right\}$.

7.3. Deciding WS1S with Finite Automata

The classical decision procedure for WS1S by Büchi [Büc59] (as described in Section 3.3 of [CDG$^+$07]) is based on a logic-automata connection and decides validity (satisfiability) of a WS1S formula $\varphi(X_1, \ldots, X_n)$ by constructing the NFA $\mathcal{A}_\varphi$ over $\{X_1, \ldots, X_n\}$ which recognises encodings of exactly the models of φ. The automaton is built in a bottom-up manner, according to the 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 the language of $\mathcal{A}_\psi$, $L(\mathcal{A}_\psi)$, represents exactly all models of ψ, terminating with the result $\mathcal{A}_\varphi$.

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 $\substack{X_1:a \\ X_2:b}$ 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. $\substack{X_1:\epsilon \\ X_2:\epsilon}$, $\substack{X_1:011 \\ X_2:101}$, or $\substack{X_1:01100 \\ X_2:10100}$. We denote the i-th symbol of w as $w[i]$, for $i \in \mathbb{N}_0$. An assignment $\rho : \mathbb{X} \to 2^{\mathbb{N}_0}$ 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. 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 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, $\substack{X_1:0 \\ X_2:1}$, $\substack{X_1:00 \\ X_2:10}$, $\substack{X_1:000 \\ X_2:100}$, $\substack{X_1:000\ldots0 \\ X_2:100\ldots0}$ are all 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.

The automata $\mathcal{A}_{\varphi\wedge\psi}$ and $\mathcal{A}_{\varphi\vee\psi}$ are constructed from the automata $\mathcal{A}_\varphi$ and $\mathcal{A}_\psi$ by standard automata-theoretic union and intersection operations, preceded by the so-called cylindrification which unifies the alphabets of $\mathcal{A}_\varphi$ and $\mathcal{A}_\psi$. Since these operations, as well as the automata for the atomic formulae, are not the subject of the contribution proposed in the presented work, we refer the interested reader to [CDG$^+$07] for details.

The part of the procedure which is central for the work presented in this chapter is processing negation and existential quantification; we will therefore describe it in detail. The NFA $\mathcal{A}_{\neg\varphi}$ is constructed as the complement of $\mathcal{A}_\varphi$. Then, all encodings of the assignments that were accepted by $\mathcal{A}_\varphi$ are rejected by $\mathcal{A}_{\neg\varphi}$ and vice versa. The NFA $\mathcal{A}_{\exists X:\varphi}$ is obtained from the NFA $\mathcal{A}_\varphi = (Q, \Delta, I, F)$ by first projecting X from the transition relation Δ, yielding the NFA $\mathcal{A}'_\varphi = (Q, \pi_{[X]}(\Delta), I, F)$. However, $\mathcal{A}'_\varphi$ cannot be directly used as $\mathcal{A}_{\exists X:\varphi}$. The reason is that $\mathcal{A}'_\varphi$ may now be inconsistent in accepting some encodings of an assignment ρ while rejecting other encodings of ρ. For example, suppose that $\mathcal{A}_\varphi$ accepts the words $\substack{X_1:010\\X_2:001}$, $\substack{X_1:0100\\X_2:0010}$, $\substack{X_1:0100\ldots0\\X_2:0010\ldots0}$ and we are computing the NFA for $\exists X_2 : \varphi$. When we remove the X_2 row from all symbols, we obtain the NFA $\mathcal{A}'_\varphi$ that accepts the words $X_1:010$, $X_1:0100$, $X_1:0100\ldots0$, but does not accept the word $X_1:01$ that encodes the same assignment (because $\substack{X_1:01\\X_2:??} \notin L(A_\varphi)$ for any values in the places of "?"s). As a remedy for this situation, we need to modify $\mathcal{A}'_\varphi$ to also accept the rest of the encodings of ρ. This is done by enlarging the set of final states of $\mathcal{A}'_\varphi$ to also contain all states that can reach a final state of $\mathcal{A}'_\varphi$ by a sequence of $\overline{0}$ symbols. Formally, $\mathcal{A}_{\exists X:\varphi} = (Q, \pi_{[X]}(\Delta), I, F^\sharp)$ is obtained from $\mathcal{A}'_\varphi = (Q, \pi_{[X]}(\Delta), I, F)$ by computing $F^\sharp$ from F using the fixpoint computation $F^\sharp = \mu Z \,.\, F \cup \mathit{pre}_{[\pi_{[X]}(\Delta),\overline{0}]}(Z)$. Intuitively, the least fixpoint denotes the set of states backward-reachable from F following transitions of $\pi_{[X]}(\Delta)$ labelled by the symbol $\overline{0}$.

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 $L(\mathcal{A}_\varphi) = \{\overline{0}\}^*$ in the case φ is valid, or $L(\mathcal{A}_\varphi) = \emptyset$ in the case φ is invalid.

7.4. Nested Antichain-based Approach for Alternating Quantifiers

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

$$\varphi = \underbrace{\neg \exists \mathcal{X}_m \neg \ldots \neg \exists \mathcal{X}_2 \underbrace{\neg \exists \mathcal{X}_1 : \varphi_0(\mathbb{X})}_{\varphi_1}}_{\varphi_m} \tag{7.2}$$

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.

The classical procedure presented in Section 7.3 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 the subformulae of φ shown in Equation 7.2. Since eliminating existential quantification on the automata level introduces nondeterminism (due to the projection on the transition relation), every $\mathcal{A}_{\varphi_i^\sharp}$ may be nondeterministic. The computation of $\mathcal{A}_{\varphi_{i+1}}$ then involves subset construction and becomes exponential. The worst case complexity of eliminating the prefix is therefore the tower of exponentials of the height m. Even though the construction may be optimised, e.g. by minimising every $\mathcal{A}_{\varphi_i}$ (which is implemented by MONA), the size of the generated automata can quickly become intractable.

The main idea of our algorithm is inspired by the so-called antichain algorithms [DR10] (a general description of the principles of antichain algorithms can be found in Chapter 8) 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 determinisation 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 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. 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 vastly reduces the search space.

We notice that individual steps of the algorithm for constructing $\mathcal{A}_\varphi$ are very similar to testing universality. Automaton $\mathcal{A}_{\varphi_i}$ arises by subset construction from $\mathcal{A}_{\varphi_{i-1}^\sharp}$, and to compute $\mathcal{A}_{\varphi_i^\sharp}$, it is necessary to compute the set of final states $F_i^\sharp$. 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 $\overline{0}$ by π_{i+1}). To compute $F_i^\sharp$, the antichain algorithms could be actually taken off-the-shelf and run with $\mathcal{A}_{\varphi_{i-1}^\sharp}$ in the role of the input $\mathcal{A}$ and $\mathcal{A}_{\varphi_i^\sharp}$ in the role of $\overline{\mathcal{A}}$. However, this approach has the following two problems. First, antichain algorithms do not produce the automaton $\overline{\mathcal{A}}$ (here $\mathcal{A}_{\varphi_i^\sharp}$), but only a symbolic representation of a set of (backward) reachable states (here of $F_i^\sharp$). Since $\mathcal{A}_{\varphi_i^\sharp}$ 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 $\mathcal{A}_{\varphi_{i-1}^\sharp}$ 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.

The main contribution of the work presented in this chapter is an algorithm that alleviates the two problems mentioned above. It is based on a novel way of performing not only one, but all the $2m$ steps of the construction of $\mathcal{A}_\varphi$ on the fly. It uses 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.

7.4.1. Structure of the Algorithm

Let us now start explaining our on-the-fly algorithm for handling quantifier alternation. Following the construction of automata described in Section 7.3, the structure of the automata from the previous section, $\mathcal{A}_{\varphi_0}, \mathcal{A}_{\varphi_0^\sharp}, \ldots, \mathcal{A}_{\varphi_{m-1}^\sharp}, \mathcal{A}_{\varphi_m}$, 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 NFA over $\mathbb{X}$. Then, for each $0 \leq i < m$, $\mathcal{A}_{\varphi_i^\sharp}$ and $\mathcal{A}_{\varphi_{i+1}}$ are NFAs over $\pi_{i+1}(\mathbb{X})$ that have from the construction the following structure:

$$
\begin{array}{rl|rl}
\mathcal{A}_{\varphi_i^\sharp} & = (Q_i, \Delta_i^\sharp, I_i, F_i^\sharp) \text{ where} & \mathcal{A}_{\varphi_{i+1}} & = (Q_{i+1}, \Delta_{i+1}, I_{i+1}, F_{i+1}) \text{ where} \\
\Delta_i^\sharp & = \pi_{i+1}(\Delta_i) \quad \text{and} & Q_{i+1} & = 2^{Q_i}, \\
F_i^\sharp & = \mu Z \,.\, F_i \cup \mathit{pre}[\Delta_i^\sharp, \bar{0}](Z). & \Delta_{i+1} & = \left\{ R \xrightarrow{\tau} \mathit{post}[\Delta_i^\sharp, \tau](R) \;\middle|\; R \in Q_{i+1} \right\}, \\
 & & I_{i+1} & = \{I_i\}, \quad \text{and} \\
 & & F_{i+1} & = \downarrow\{Q_i \setminus F_i^\sharp\}.
\end{array}
$$

We recall that $\mathcal{A}_{\varphi_i^\sharp}$ directly corresponds to existential quantification of the variable X_i (cf. Section 7.3), and $\mathcal{A}_{\varphi_{i+1}}$ directly corresponds to the complement of $\mathcal{A}_{\varphi_i^\sharp}$ (cf. Section 7.2).

A crucial observation behind our approach is that, because φ is ground, $\mathcal{A}_\varphi$ is an NFA over an empty set of variables, and, therefore, $L(\mathcal{A}_\varphi)$ is either the empty set $\emptyset$ or the set $\{\bar{0}\}^*$ (as described in Section 7.3). 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

$$\models \varphi \quad \text{iff} \quad I_m \cap F_m \neq \emptyset. \tag{7.3}$$

To compute I_m from I_0 is straightforward (it equals $\{\{\ldots\{\{I_0\}\}\ldots\}\}$ nested m-times). In the rest of the section, we will describe how to compute F_m (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 $\mathcal{A}_{\varphi_0}$, $\mathcal{A}_{\varphi_0^\sharp}$, …, $\mathcal{A}_{\varphi_{m-1}^\sharp}$, $\mathcal{A}_{\varphi_m}$ 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 suffices for testing the validity of φ. The algorithm starts with F_0 and uses the following recursive equations:

$$
\begin{array}{llll}
\text{(i)} \quad F_{i+1} & = \downarrow\{N_i^\sharp\}, & \text{(ii)} \quad F_i^\sharp & = \mu Z \,.\, F_i \cup \mathit{pre}[\Delta_i^\sharp, \bar{0}](Z), \\
\text{(iii)} \quad N_{i+1} & = \uparrow\coprod\{F_i^\sharp\}, & \text{(iv)} \quad N_i^\sharp & = \nu Z \,.\, N_i \cap \mathit{cpre}[\Delta_i^\sharp, \bar{0}](Z).
\end{array}
\tag{7.4}
$$

Intuitively, Equations (i) and (ii) are directly from the definition of $\mathcal{A}_i$ and $\mathcal{A}_i^\sharp$. Equation (iii) is a dual of Equation (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, Equation (iv) is a dual of Equation (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 $\overline{0}$ within k steps. In the next iteration, only those states of Z are kept such that all their $\overline{0}$-successors are in Z. Hence, the new value of Z is the set of states that cannot reach F_i over $\overline{0}$ in $k+1$ steps, and the computation stabilises with the set of states that cannot reach F_i over $\overline{0}$ 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/downward closed sets. Particularly, in Sections 7.4.2 and 7.4.3, we show how the fixpoints from Equations (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 7.4.4 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 7.4.5 shows how to test emptiness of $I_m \cap F_m$ on the symbolic terms, and Section 7.4.6 describes the subsumption relation used to minimise the symbolic term representation used within computations of Equations (ii) and (iv). Proofs of the lemmas can be found at the ends of the respective sections.

7.4.2. 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 7.4(iv):

$$f_{N_i^\sharp}(Z) = N_i \cap cpre_{[\Delta_i^\sharp,\overline{0}]}(Z). \tag{7.5}$$

We will show that the whole fixpoint computation from Equation 7.4(iv) can be carried out symbolically on the representatives of Z. We will explain that: (a) All intermediate values of Z have the form $\uparrow\coprod\mathcal{R}$, $\mathcal{R} \subseteq Q_i$, so the sets $\mathcal{R}$ can be used as their symbolic representatives. (b) *cpre* and $\cap$ can be computed on such symbolic representation efficiently.

Let us start with the computation of $cpre_{[\Delta_i^\sharp,\tau]}(Z)$ where $\tau \in \pi_{i+1}(\mathbb{X})$, assuming that Z is of the form $\uparrow\coprod\mathcal{R}$, 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, formalised in the following lemma.

Lemma 7.1. *Let $\mathcal{R}$ be a finite set of sets. Then, it holds that*

$$\uparrow\coprod\mathcal{R} = \bigcap_{R_j\in\mathcal{R}} \uparrow\coprod\{R_j\}. \tag{7.6}$$

Z can thus be written as the *cpre*-image $cpre_{[\Delta_i^\sharp,\tau]}(\bigcap\mathcal{S})$ of the intersection of the elements of a set $\mathcal{S}$ having the form $\uparrow\coprod\{R_j\}, R_j \in \mathcal{R}$. Further, because *cpre* distributes over $\cap$, we can compute the *cpre*-image of an intersection by computing intersection of the *cpre*-images, i.e.

$$cpre_{[\Delta_i^\sharp,\tau]}(\bigcap\mathcal{S}) = \bigcap_{S\in\mathcal{S}} cpre_{[\Delta_i^\sharp,\tau]}(S). \tag{7.7}$$

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

$$\mathit{cpre}[\Delta_i^\sharp,\tau](S) = \bigcap_{\omega \in \pi_{i+1}^{-1}(\tau)} \mathit{cpre}[\Delta_i,\omega](S). \tag{7.8}$$

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 Lemma 7.1 and Equations 7.7 and 7.8, we can write $\mathit{cpre}[\Delta_i^\sharp,\tau](Z)$ as

$$\bigcap_{\substack{S \in \mathcal{S} \\ \omega \in \pi_{i+1}^{-1}(\tau)}} \mathit{cpre}[\Delta_i,\omega](S). \tag{7.9}$$

To compute the individual conjuncts $\mathit{cpre}[\Delta_i,\omega](S)$, we take advantage of the fact that every S is in the special form $\uparrow\coprod\{R_j\}$, and that Δ_i is, by its definition (obtained from determinisation 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 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 of singletons $S = \uparrow\coprod\{R_j\}$, we get the following lemma:

Lemma 7.2. *Let $R_j \subseteq Q_{i-1}$ and ω be a symbol over $\pi_i(\mathbb{X})$ for $i > 0$. Then*

$$\mathit{cpre}[\Delta_i,\omega](\uparrow\coprod\{R_j\}) = \uparrow\coprod\left\{\mathit{pre}[\Delta_{i-1}^\sharp,\omega](R_j)\right\}. \tag{7.10}$$

Intuitively, the 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. $q \xrightarrow{\omega} p \in \Delta_{i-1}^\sharp$. Using Lemma 7.2, $\mathit{cpre}[\Delta_i^\sharp,\tau](Z)$ can be rewritten as

$$\bigcap_{\substack{R \in \mathcal{R} \\ \omega \in \pi_{i+1}^{-1}(\tau)}} \uparrow\coprod\left\{\mathit{pre}[\Delta_{i-1}^\sharp,\omega](R_j)\right\}. \tag{7.11}$$

By applying Lemma 7.1, we get the final formula for $\mathit{cpre}[\Delta_i^\sharp,\tau]$ shown in the lemma below.

Lemma 7.3. *Let $\mathcal{R} \subseteq Q_i$ and τ be a symbol over $\pi_{i+1}(\mathbb{X})$. Then*

$$\mathit{cpre}[\Delta_i^\sharp,\tau](\uparrow\coprod\mathcal{R}) = \uparrow\coprod\left\{\mathit{pre}[\Delta_{i-1}^\sharp,\omega](R_j) \mid \omega \in \pi_{i+1}^{-1}(\tau), R_j \in \mathcal{R}\right\}. \tag{7.12}$$

In order to compute $f_{N_i^\sharp}(Z)$, it remains to intersect $\mathit{cpre}[\Delta_i^\sharp,\bar{0}](Z)$, computed using Lemma 7.3, with N_i. By Equation 7.4(iii), N_i equals $\uparrow\coprod\{F_{i-1}^\sharp\}$, and, by Lemma 7.1, the intersection can be done symbolically as

$$f_{N_i^\sharp}(Z) = \uparrow\coprod\Big(\{F_{i-1}^\sharp\} \cup \left\{\mathit{pre}[\Delta_{i-1}^\sharp,\omega](R_j) \mid \omega \in \pi_{i+1}^{-1}(\bar{0}), R_j \in \mathcal{R}\right\}\Big). \tag{7.13}$$

Finally, note that a symbolic application of $f_{N_i^\sharp}$ 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 $F_{i-1}^\sharp$. The computation starts from $N_i = \uparrow\coprod\{F_{i-1}^\sharp\}$, represented by $\{F_{i-1}^\sharp\}$, and each of its steps, implemented by Equation 7.13, preserves the form of sets $\uparrow\coprod\mathcal{R}$, represented by $\mathcal{R}$.

Proofs of the Used Lemmas

Lemma 7.4. *Let $\mathcal{X}$ and $\mathcal{Y}$ be sets of sets. Then it holds that*

$$\uparrow\coprod\mathbb{X} \cap \uparrow\coprod\mathbb{Y} = \uparrow\coprod(\mathbb{X} \cup \mathbb{Y}). \tag{7.14}$$

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

$$\begin{aligned}\uparrow\coprod\mathbb{X} &= \uparrow\big\{\{x_1,\dots,x_n\} \,\big|\, (x_1,\dots,x_n) \in \prod\mathbb{X}\big\} \quad \text{and}\\ \uparrow\coprod\mathbb{Y} &= \uparrow\big\{\{y_1,\dots,y_m\} \,\big|\, (y_1,\dots,y_m) \in \prod\mathbb{Y}\big\}.\end{aligned} \tag{7.15}$$

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

$$\uparrow\big\{\{x_1,\dots,x_n\} \cup \{y_1,\dots,y_m\} \,\big|\, (x_1,\dots,x_n) \in \prod X \wedge (y_1,\dots,y_m) \in \prod Y\big\}. \tag{7.16}$$

We observe that this set can be also expressed as

$$\uparrow\big\{\{x_1,\dots,x_n,y_1,\dots,y_m\} \,\big|\, (x_1,\dots,x_n,y_1,\dots y_m) \in \prod(X \cup Y)\big\} \tag{7.17}$$

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

Lemma 7.1. *Let $\mathcal{R}$ be a finite set of sets. Then, it holds that*

$$\uparrow\coprod\mathcal{R} = \bigcap_{R_j \in \mathcal{R}} \uparrow\coprod\{R_j\}. \tag{7.6}$$

Proof. Because intersection and union are both associative operations and $\mathcal{R}$ is a finite set $\mathcal{R} = \{R_1,\dots,R_n\}$, this lemma is a simple consequence of Lemma 7.4. □

Lemma 7.2. *Let $R_j \subseteq Q_{i-1}$ and ω be a symbol over $\pi_i(\mathbb{X})$ for $i > 0$. Then*

$$cpre_{[\Delta_i,\omega]}(\uparrow\coprod\{R_j\}) = \uparrow\coprod\big\{pre_{[\Delta_{i-1}^\sharp,\omega]}(R_j)\big\}. \tag{7.10}$$

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 $\coprod\big\{pre_{[\Delta_{i-1}^\sharp,\omega]}(R_j)\big\}$ 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 $\coprod\big\{pre_{[\Delta_{i-1}^\sharp,\omega]}(R_j)\big\}$.

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

$$\mathit{post}_{[\Delta_i,\omega]}(\{S\}) \subseteq \uparrow\coprod\{R_j\}, \tag{7.18}$$

and from the definition of Δ_i, it holds that

$$\mathit{post}_{[\Delta_i,\omega]}(\{S\}) = \{\mathit{post}_{[\Delta^\sharp_{i-1},\omega]}(S)\}. \tag{7.19}$$

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

$$\mathit{post}_{[\Delta^\sharp_{i-1},\omega]}(T) \supseteq \mathit{post}_{[\Delta^\sharp_{i-1},\omega]}(S) \tag{7.20}$$

and, therefore, it also holds that

$$\mathit{post}_{[\Delta_i,\omega]}(\{T\}) = \{\mathit{post}_{[\Delta^\sharp_{i-1},\omega]}(T)\} \subseteq \uparrow\coprod\{R_j\}. \tag{7.21}$$

Therefore, $T \in \mathit{cpre}_{[\Delta_i,\omega]}(\uparrow\coprod\{R_j\})$ and the set $\mathit{cpre}_{[\Delta_i,\omega]}(\uparrow\coprod\{R_j\})$ is upward closed.

2. Proving that for every element S from $\coprod\left\{\mathit{pre}_{[\Delta^\sharp_{i-1},\omega]}(R_j)\right\}$ it holds that S is in $\mathit{cpre}_{[\Delta_i,\omega]}(\uparrow\coprod\{R_j\})$: From the properties of $\coprod$, it holds that $S = \{s\}$ is a singleton. Because $s \in \mathit{pre}_{[\Delta^\sharp_{i-1},\omega]}(R_j)$, there is a transition $s \xrightarrow{\omega} r \in \Delta^\sharp_{i-1}$ for some $r \in R_j$. Since $\mathit{post}_{[\Delta^\sharp_{i-1},\omega]}(S) \supseteq \{r\}$, it follows from the definition of Δ_i that $\mathit{post}_{[\Delta_i,\omega]}(\{S\}) = \{T\}$ where $T \supseteq \{r\}$, and so $T \in \uparrow\coprod\{R_j\}$ and $\mathit{post}_{[\Delta_i,\omega]}(\{S\}) \subseteq \uparrow\coprod\{R_j\}$. We use the definition of *cpre* to conclude that $S \in \mathit{cpre}_{[\Delta_i,\omega]}(\uparrow\coprod\{R_j\})$.

3. Proving that for every $T \in \mathit{cpre}_{[\Delta_i,\omega]}(\uparrow\coprod\{R_j\})$ there exists some element $S \in \coprod\left\{\mathit{pre}_{[\Delta^\sharp_{i-1},\omega]}(R_j)\right\}$ such that $S \subseteq T$: From $T \in \mathit{cpre}_{[\Delta_i,\omega]}(\uparrow\coprod\{R_j\})$ and the definition of Δ_i, we have that

$$\mathit{post}_{[\Delta_i,\omega]}(\{T\}) = \{P\} \subseteq \uparrow\coprod\{R_j\} \tag{7.22}$$

for P s.t. $\mathit{post}_{[\Delta^\sharp_{i-1},\omega]}(T) = P$. 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 $t \in \mathit{pre}_{[\Delta^\sharp_{i-1},\omega]}(\{r\})$, we choose $S = \{t\}$ and we are done. $\square$

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

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

$$f_{F^\sharp_i}(Z) = F_i \cup \mathit{pre}_{[\Delta^\sharp_i,\overline{0}]}(Z). \tag{7.23}$$

We will show that the whole fixpoint computation from Equation 7.4(ii) can be again carried out symbolically. We will explain the following: (a) All intermediate values of

Z are of the form $\downarrow\mathcal{R}$, $\mathcal{R} \subseteq Q_i$, so the sets $\mathcal{R}$ can be used as their symbolic representatives. (b) *pre* and $\cup$ can be computed efficiently on such a symbolic representation. The computation is a simpler analogy of the one in Section 7.4.2.

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 $\mathcal{R} = \{R_1, \ldots, R_n\}$. A simple analogy to Lemma 7.1 and Equation 7.7 of Section 7.4.2 is that the union of downward closed sets is a downward closed set generated by the union of their generators, i.e. $\downarrow\mathcal{R} = \bigcup_{R_j \in \mathcal{R}} \downarrow\{R_j\}$ and that *pre* distributes over union, i.e.

$$pre[\Delta_i^\sharp,\tau](\bigcup \mathcal{R}) = \bigcup_{R_j \in \mathcal{R}} pre[\Delta_i^\sharp,\tau](\downarrow\{R_j\}). \tag{7.24}$$

An analogy of Equation 7.8 holds too:

$$pre[\Delta_i^\sharp,\tau](S) = \bigcup_{\omega \in \pi_{i+1}^{-1}(\tau)} pre[\Delta_i,\omega](S). \tag{7.25}$$

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} leaves with the target in S. Using Equation 7.25, we can write $pre[\Delta_i^\sharp,\tau](Z)$ as

$$\bigcup_{\substack{R_j \in \mathcal{R} \\ \omega \in \pi_{i+1}^{-1}(\tau)}} pre[\Delta_i,\omega](\downarrow\{R_j\}). \tag{7.26}$$

To compute the 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 (obtained from determinisation 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 monotonicity, the $pre[\Delta_i,\omega]$-image of a downward closed set is downward closed. Moreover, we observe that it can be computed symbolically using *cpre* on elements of its generators. In particular, for a set $\downarrow\{R_j\}$, we get the following lemma, which is a dual of Lemma 7.2:

Lemma 7.5. *Let $R_j \subseteq Q_{i-1}$ and ω be a symbol over $\pi_i(\mathbb{X})$ for $i > 0$. Then*

$$pre[\Delta_i,\omega](\downarrow\{R_j\}) = \downarrow\{cpre[\Delta_{i-1}^\sharp,\omega](R_j)\}. \tag{7.27}$$

Intuitively, the sets with the *post*-images below the set R_j are those which do not have an outgoing transition leading outside R_j. The largest such set is $cpre[\Delta_{i-1}^\sharp,\omega](R_j)$. Using Lemma 7.5, $pre[\Delta_i^\sharp,\tau](Z)$ can be rewritten as

$$\bigcup_{\substack{R_j \in \mathcal{R} \\ \omega \in \pi_{i+1}^{-1}(\tau)}} \downarrow\{cpre[\Delta_{i-1}^\sharp,\omega](R_j)\} \tag{7.28}$$

which gives us the final formula for $pre[\Delta_i^\sharp,\tau]$ described in Lemma 7.6.

Lemma 7.6. *Let $\mathcal{R} \subseteq Q_i$ and τ be a symbol over $\pi_{i+1}(\mathbb{X})$. Then*

$$pre_{[\Delta_i^\sharp,\tau]}(\downarrow\mathcal{R}) = \downarrow\{cpre_{[\Delta_{i-1}^\sharp,\omega]}(R_j) \mid \omega \in \pi_{i+1}^{-1}(\tau), R_j \in \mathcal{R}\}. \tag{7.29}$$

In order to compute $f_{F_i^\sharp}(Z)$, it remains to unite $pre_{[\Delta_i^\sharp,\bar{0}]}(Z)$, which is computed using Lemma 7.6, with F_i. From Equation 7.4(i), F_i equals $\downarrow\{N_{i-1}^\sharp\}$, so the union can be done symbolically as

$$f_{F_i^\sharp}(Z) = \downarrow\Big(\{N_{i-1}^\sharp\} \cup \big\{cpre_{[\Delta_{i-1}^\sharp,\omega]}(R_j) \mid \omega \in \pi_{i+1}^{-1}(\bar{0}), R_j \in \mathcal{R}\big\}\Big). \tag{7.30}$$

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 Equation 7.30, preserves the form of sets $\downarrow\mathcal{R}$, represented by $\mathcal{R}$.

Proofs of the Used Lemmas

Lemma 7.5. *Let $R_j \subseteq Q_{i-1}$ and ω be a symbol over $\pi_i(\mathbb{X})$ for $i > 0$. Then*

$$pre_{[\Delta_i,\omega]}(\downarrow\{R_j\}) = \downarrow\{cpre_{[\Delta_{i-1}^\sharp,\omega]}(R_j)\}. \tag{7.27}$$

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 $pre_{[\Delta_i,\omega]}(\downarrow\{R_j\})$ 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

$$post_{[\Delta_i,\omega]}(\{S'\}) = \{post_{[\Delta_{i-1}^\sharp,\omega]}(S')\} \subseteq \downarrow\{R_j\}, \tag{7.31}$$

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

$$post_{[\Delta_{i-1}^\sharp,\omega]}(T) \subseteq post_{[\Delta_{i-1}^\sharp,\omega]}(S') \tag{7.32}$$

   and so it also holds that

$$post_{[\Delta_i,\omega]}(\{T\}) = \{post_{[\Delta_{i-1}^\sharp,\omega]}(T)\} \subseteq \downarrow\{R_j\}. \tag{7.33}$$

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

2. Proving that $S = cpre_{[\Delta_{i-1}^\sharp,\omega]}(R_j) \in pre_{[\Delta_i,\omega]}(\downarrow\{R_j\})$: From the definition of *cpre*, it holds that

$$post_{[\Delta_{i-1}^\sharp,\omega]}(S) = S' \subseteq R_j. \tag{7.34}$$

   Further, from the definition of Δ_i, it holds that $S \xrightarrow{\omega} S' \in \Delta_i$ and, therefore, $S \in pre_{[\Delta_i,\omega]}(\downarrow\{R_j\})$.

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
$$P \subseteq R_j \quad \Longrightarrow \quad \mathit{cpre}[\Delta^\sharp_{i-1},\omega](P) \subseteq \mathit{cpre}[\Delta^\sharp_{i-1},\omega](R_j). \tag{7.35}$$
Therefore, we can conclude that $T \subseteq \mathit{cpre}[\Delta^\sharp_{i-1},\omega](R_j) = S$. □

7.4.4. Computation of $F^\sharp_i$ and $N^\sharp_i$ on Symbolic Terms

Sections 7.4.2 and 7.4.3 show how sets of states arising within the fixpoint computations from Equations 7.4(ii) and 7.4(iv) can be represented symbolically using representatives which are sets of states of the lower level. The sets of states of the lower level will be again represented symbolically. When computing the fixpoint of level i, we will work with nested symbolic representation of states of depth i. Particularly, sets of states of Q_k, $0 \leq k \leq i$, are represented by *terms of level k* where a term of level 0 is a subset of Q_0, a term of level $2j+1$, $j \geq 0$, 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 $2j$, $j > 0$, 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 $f_{N^\sharp_{2j+1}}$ on a term of level $2j+1$ and computation of *pre* and $f_{F^\sharp_{2j}}$ 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 $f_{N^\sharp_{2j+1}}$ called on a term of level $2j+1$, Lemma 7.3 reduces the computation to a computation of *pre* on its sub-terms of level $2j$, which is again reduced by Lemma 7.6 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 $f_{F^\sharp_{2j}}$ called on a term of level $2j$ is symmetrical.

Example. We will demonstrate the run of our algorithm on the following abstract example. Consider a ground WS1S formula $\varphi = \neg\exists\mathcal{X}_3\neg\exists\mathcal{X}_2\neg\exists\mathcal{X}_1 : \varphi_0$ and an NFA $\mathcal{A}_0 = (Q_0, \Delta_0, I_0 = \{a\}, F_0 = \{a, b\})$ that represents φ_0. Recall that our method decides validity of φ by computing symbolically the sequence of sets $F^\sharp_0, N_1, N^\sharp_1, F_2, F^\sharp_2, N_3$, each of them represented using a symbolic term, and then checks if $I_3 \cap N_3 \neq \emptyset$. In the following paragraph, we will show how such a sequence is computed and interleave the description with examples of possible intermediate results.

The fixpoint computation from Equation 7.4(ii) of the first set in the sequence, $F^\sharp_0$, is an explicit computation of the set of states backward-reachable from F_0 via $\overline{0}$ transitions of $\Delta^\sharp_0$. It is done using Equation 7.23, yielding e.g. the term
$$t[F^\sharp_0] = F^\sharp_0 = \{a, b, c\}.$$
The fixpoint computation of $N^\sharp_1$ from Equation 7.4(iv) is done symbolically. It starts from the set N_1 represented using Equation 7.4(iii) as the term $t[N_1] = \uparrow\coprod\big\{\{a, b, c\}\big\}$,

and each of its iterations is carried out using Equation 7.13. Equation 7.13 transforms the problem of computing $cpre[\Delta_1,\omega']$-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 Equation 7.23 in the same way as when computing $t[F_0^\sharp]$, ending with e.g. the term

$$t[N_1^\sharp] = \uparrow\coprod\big\{\{a,b,c\},\{b,c\},\{c,d\}\big\}.$$

The term representing F_2 is then $t[F_2] = \downarrow\{t[N_1^\sharp]\}$, due to Equation 7.4(i). The symbolic fixpoint computation of $F_2^\sharp$ from Equation 7.4(ii) then starts from $t[F_2]$, in our example

$$t[F_2] = \downarrow\Big\{\uparrow\coprod\big\{\{a,b,c\},\{b,c\},\{c,d\}\big\}\Big\}.$$

Its steps are computed using Equation 7.30, which transforms the computation of the image of $pre[\Delta_2^\sharp,\omega'']$ into computations of a series of $cpre[\Delta_1^\sharp,\omega']$-images of sub-terms. These are in turn transformed by Lemma 7.3 into computations of $pre[\Delta_0^\sharp,\omega]$-images of sub-sub-terms, subsets of Q_0, in our example yielding e.g. the term

$$t[F_2^\sharp] = \downarrow\Big\{\uparrow\coprod\big\{\{a,b,c\},\{b,c\},\{c,d\}\big\},\uparrow\coprod\big\{\{b\},\{d\}\big\},\uparrow\coprod\big\{\{a\},\{c,d\}\big\}\Big\}.$$

Using Equation 7.4(iv), the final term representing N_3 is then

$$t[N_3] = \uparrow\coprod\bigg\{\downarrow\Big\{\uparrow\coprod\big\{\{a,b,c\},\{b,c\},\{c,d\}\big\},\uparrow\coprod\big\{\{b\},\{d\}\big\},\uparrow\coprod\big\{\{a\},\{c,d\}\big\}\Big\}\bigg\}.$$

In the next section, we will describe how we check whether $I_3 \cap F_3 \neq \emptyset$ using the computed term $t[N_3]$.

7.4.5. Testing $I_m \cap F_m \overset{?}{\neq} \emptyset$ on Symbolic Terms

Due to the special form of the set I_m (every $I_i, 1 \leq i \leq m$, is the singleton set $\{I_{i-1}\}$, cf. Section 7.4.1), the test $I_m \cap F_m \neq \emptyset$ can be done efficiently over the symbolic terms representing F_m. Because $I_m = \{I_{m-1}\}$ is a singleton set, testing $I_m \cap F_m \neq \emptyset$ is equivalent to testing $I_{m-1} \in F_m$. If m is odd, our approach computes the symbolic representation of N_m instead of F_m. Obviously, since N_m is the complement of F_m, it holds that $I_{m-1} \in F_m \iff I_{m-1} \notin N_m$. Our way of testing $I_{m-1} \in Y_m$ on a symbolic representation of the set Y_m of level m is based on the following equations:

$$\{x\} \in \downarrow\mathbb{Y} \iff \exists Y \in \mathbb{Y} : x \in Y \tag{7.36}$$
$$\{x\} \in \uparrow\coprod\mathbb{Y} \iff \forall Y \in \mathbb{Y} : x \in Y \tag{7.37}$$
$$\text{and for } i = 0, \qquad I_0 \in \uparrow\coprod\mathbb{Y} \iff \forall Y \in \mathbb{Y} : I_0 \cap Y \neq \emptyset. \tag{7.38}$$

Given a symbolic term $t[X]$ of level m representing a set $X \subseteq Q_m$, testing emptiness of $I_m \cap F_m$ or $I_m \cap N_m$ can be done over $t[X]$ by a recursive procedure that descends along the structure of $t[X]$ using Equations 7.36 and 7.37, essentially generating an *And-Or* tree, terminating the descent by the use of Equation 7.38.

Example. In the example of Section 7.4.4, we would test whether $\{\{\{\{a\}\}\}\} \cap N_3 = \emptyset$ over $t_{[N_3]}$. This is equivalent to testing whether $I_2 = \{\{\{a\}\}\} \in N_3$. From Equation 7.37 we get that

$$I_2 \in N_3 \iff I_1 = \{\{a\}\} \in F_2^\sharp \tag{7.39}$$

because $F_2^\sharp$ is the denotation of the only sub-term $t_{[F_2^\sharp]}$ of $t_{[N_3]}$. Equation 7.36 implies that

$$I_1 = \{\{a\}\} \in F_2^\sharp \iff \{a\} \in N_1^\sharp \vee \{a\} \in \uparrow\coprod\big\{\{b\},\{d\}\big\} \vee \{a\} \in \uparrow\coprod\big\{\{a\},\{c,d\}\big\}. \tag{7.40}$$

Each of the disjuncts could then be further reduced by Equation 7.37 into a conjunction of membership queries on the base level which would be solved by Equation 7.38. Since none of the disjuncts is satisfied, we conclude that $I_1 \notin F_2^\sharp$, so $I_2 \notin N_3$, implying that $I_2 \in F_3$ and thus obtain the result $\models \varphi$.

7.4.6. 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 $\mathbb{X}$ containing a pair of distinct elements $Y, Z \in \mathbb{X}$ s.t. $Y \subseteq Z$, it holds that

$$\downarrow\mathbb{X} = \downarrow(\mathbb{X} \setminus Y) \quad \text{and} \quad \uparrow\coprod\mathbb{X} = \uparrow\coprod(\mathbb{X} \setminus Z). \tag{7.41}$$

Therefore, if $\mathbb{X}$ is used to represent the set $\downarrow\mathbb{X}$, the element Y is *subsumed* by Z and can be removed from $\mathbb{X}$ without changing its denotation. Likewise, if $\mathbb{X}$ is used to represent $\uparrow\coprod\mathbb{X}$, the element Z is *subsumed* by Y and can be removed from $\mathbb{X}$ 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:

$$\downarrow\mathbb{X} \subseteq \downarrow\mathbb{Y} \iff \forall X \in \mathbb{X} \exists Y \in \mathbb{Y} : X \subseteq Y \tag{7.42}$$

$$\uparrow\coprod\mathbb{X} \subseteq \uparrow\coprod\mathbb{Y} \iff \forall Y \in \mathbb{Y} \exists X \in \mathbb{X} : X \subseteq Y. \tag{7.43}$$

Using Equations 7.42 and 7.43, 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 Equations 7.42 and 7.43 on levels greater than 0, and on level 0, where terms are subsets of Q_0, it tests subsumption by set inclusion.

Example. In the example from Section 7.4.4, we can use the inclusion $\{b,c\} \subseteq \{a,b,c\}$ and Equation 7.41 to reduce $t_{[N_1^\sharp]} = \uparrow\coprod\big\{\{a,b,c\},\{b,c\},\{c,d\}\big\}$ to the term

$$t_{[N_1]}' = \uparrow\coprod\big\{\{b,c\},\{c,d\}\big\}.$$

Table 7.1.: Results for practical examples

Benchmark	Time [s]		Space [states]	
	MONA	dWiNA	MONA	dWiNA
`reverse-before-loop`	0.01	0.01	179	47
`insert-in-loop`	0.01	0.01	463	110
`bubblesort-else`	0.01	0.01	1 285	271
`reverse-in-loop`	0.02	0.02	1 311	274
`bubblesort-if-else`	0.02	0.23	4 260	1 040
`bubblesort-if-if`	0.12	1.14	8 390	2 065

Moreover, Equation 7.43 implies that the term $\uparrow\coprod\big\{\{b,c\},\{c,d\}\big\}$ is subsumed by the term $\uparrow\coprod\big\{\{b\},\{d\}\big\}$, and, therefore, we can reduce the term $t_{[F_2^\sharp]}$ to the term

$$t_{[F_2^\sharp]}' = \downarrow\Big\{\uparrow\coprod\big\{\{b\},\{d\}\big\},\uparrow\coprod\big\{\{a\},\{c,d\}\big\}\Big\}.$$

7.5. Experimental Evaluation

We implemented a prototype of the approach presented in this chapter in the tool `dWiNA` [FHLV14] and evaluated it in a benchmark of both practical and generated examples. The tool uses the frontend of MONA to parse input formulae and also for the construction of the base automaton $\mathcal{A}_{\varphi_0}$, and further uses the semi-symbolic encoding of NFAs (represented as unary TAs) from the VATA library, which is described in Chapters 9 and 10. The tool supports the following two modes of operation.

In mode I, we use MONA to generate the deterministic automaton $\mathcal{A}_{\varphi_0}$ corresponding to the matrix of the formula φ, translate it to VATA and run our algorithm for handling the prefix of φ using VATA. In mode II, we first translate the formula φ into the formula φ' in prenex normal form (i.e. it consists of a quantifier prefix and a quantifier-free matrix) where the occurence of negation in the matrix is limited to literals, and then construct the nondeterministic automaton $\mathcal{A}_{\varphi_0}$ directly using VATA.

Our experiments were performed on an Intel Core i7-4770@3.4 GHz processor with 32 GiB RAM. The practical formulae for our experiments that we report on here were obtained from the shape analysis of [MQ11] and evaluated using mode I of our tool; the results are shown in Table 7.1 (see [FHLV14] for additional experimental results). We measure the time of runs of the tools for processing only the prefix of the formulae. We can observe that w.r.t. the speed, we get comparable results; in some cases `dWiNA` is slower than MONA, which we attribute to the fact that our prototype implementation is, when compared with MONA, quite immature. Regarding space, we compare the sum of the number of states of all automata generated by MONA when processing the prefix of φ with the number of symbolic terms generated by `dWiNA` for processing the same. We can observe a significant reduction in the generated state space. We also tried to

Table 7.2.: Results for generated formulae

	Time [s]		**Space [states]**	
k	`MONA`	`dWiNA`	`MONA`	`dWiNA`
1	0.11	0.01	10 718	39
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

run `dWiNA` on the modified formulae in mode II but ran into the problem that we were not able to construct the nondeterministic automaton for the quantifier-free matrix φ_0 in reasonable time. This was because after transformation of φ into prenex normal form, if φ_0 contains many conjunctions, the sizes of the automata generated using intersection grow too large (one of the reasons for this is that Vata in its current version does not support efficient reduction of automata).

To better evaluate the scalability of our approach, we created several parameterised families of WS1S formulae. We start with basic formulae encoding interesting relations among subsets of $\mathbb{N}_0$, such as existence of certain transitive relations, singleton sets, or intervals (their full definition can be found in [FHLV14]). From these we algorithmically create families of formulae with larger quantifier depth, regardless of the meaning of the created formulae (though their semantics is still nontrivial). In Table 7.2, we give the results for one of the families where the basic formula expresses existence of an ascending chain of n sets ordered w.r.t. $\subset$ (the value ∞ denotes a timeout). The parameter k stands for the number of alternations in the prefix of the formulae:

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

We ran the experiments in mode II of `dWiNA` (the experiment in mode I was not successful due to a too costly conversion of a large base automaton from MONA to Vata).

7.6. Conclusion and Future Work

We presented a new approach for dealing with alternating quantifications within the automata-based decision procedure for WS1S. Our approach is based on a generalisation 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 (our tool outperforms MONA in many cases) and show that the direction started in this work—using modern techniques for nondeterministic automata in the context of deciding WS1S formulae—is promising.

An interesting direction of further development seems to be lifting the symbolic *pre*/*cpre* operators to a more general notion of terms that would allow one to work with general sub-formulae (which may include logical connectives and nested quantifiers). The algorithm could then be run over arbitrary formulae, without the need of the transformation into the prenex form. This would open a way of adopting optimisations used in other tools as well as syntactical optimisations of the input formula such as anti-prenexing. Another way of improvement is using simulation-based techniques to reduce the generated automata as well as to weaken the term-subsumption relation (an efficient algorithm for computing simulation over BDD-represented automata is needed). We also plan to extend the algorithms to WSkS and tree-automata, and perhaps even further to more general inductive structures.

Part III.

Efficient Techniques for Manipulation of Nondeterministic Tree Automata

8. Downward Inclusion Checking for Tree Automata

The previous chapters of this thesis introduced several formal verification techniques that rely on finite tree automata. Even before, there have been numerous other applications of TAs, such as (abstract) regular tree model checking [AJMd02, BHRV12], verification of programs with complex dynamic data structures [BHRV06], analysis of network firewalls [Bou11], and implementation of decision procedures of logics such as WS2S or MSO [KMS02], which themselves have numerous applications (among the most recent and promising ones, let us mention at least verification of programs manipulating heap structures with data [MPQ11]).

Recently, there has been notable progress in the development of algorithms for efficient manipulation of nondeterministic finite tree automata (TAs), more specifically, in solving the crucial problems of automata reduction [ABH$^+$08] and of checking language inclusion [TH03, BHH$^+$08, ACH$^+$10]. As shown e.g. in [BHH$^+$08], replacing deterministic automata by nondeterministic ones can—in combination with the new methods for handling TAs—lead to great efficiency gains. In the work presented in this chapter, we further advance the research on efficient algorithms for handling TAs by proposing a new algorithm for inclusion checking that turns out to significantly outperform the existing algorithms in most of our experiments.

Upward inclusion checking. The classic textbook algorithm for checking inclusion $L(\mathcal{A}_S) \subseteq L(\mathcal{A}_B)$ between two TAs $\mathcal{A}_S$ (Small) and $\mathcal{A}_B$ (Big) first bottom-up determinises $\mathcal{A}_B$, computes the complement automaton $\overline{\mathcal{A}_B}$ of $\mathcal{A}_B$ (the states, called *macrostates*, of which are sets of states of $\mathcal{A}_B$), and then checks language emptiness of the product automaton accepting $L(\mathcal{A}_S) \cap L(\overline{\mathcal{A}_B})$. This approach has been optimised in [TH03, BHH$^+$08, ACH$^+$10] by avoiding the construction of the whole product automaton (which can be exponentially larger than $\mathcal{A}_B$ and which is indeed extremely large in many practical cases) by constructing its states and checking language emptiness on the fly. The optimised algorithm is based on starting from the leaf states of both automata and maintaining a set of reachable pairs (q_S, P_B) where q_S is a state of $\mathcal{A}_S$ and P_B is a set of states of $\mathcal{A}_B$. New pairs (q_S, P_B) are generated by taking a tuple of states $q_1, \ldots, q_n$ such that every q_i appears in some reachable pair (q_i, P_i) and q_S is a bottom-up *post* of the tuple in $\mathcal{A}_S$ over some symbol a. The set P_B is then obtained as the bottom-up a-post in $\mathcal{A}_B$ of all tuples in $P_1 \times \cdots \times P_n$. In case q_S is a root state and P_B, on the other hand, contains no root state, the algorithm terminates with the answer $L(\mathcal{A}_S) \not\subseteq L(\mathcal{A}_B)$ (this corresponds to finding a witness from the set $L(\mathcal{A}_S) \cap L(\overline{\mathcal{A}_B})$). If no new pair can be generated, the algorithm concludes that $L(\mathcal{A}_S) \subseteq L(\mathcal{A}_B)$.

The particular optimisation used in [TH03, BHH$^+$08, ACH$^+$10], called the *antichain* principle, is based on removing from the set of reachable pairs those pairs (q_S, P_B) for which there is already a reachable pair (q_S, P'_B) in the set, with $P'_B \subseteq P_B$. The argument why this pruning is correct is that P'_B has a higher chance to generate a set of states that contains no root state. On the other hand, for every set of states reachable from P'_B, there will be a corresponding larger (w.r.t. inclusion) set of states reachable from P_B, so if the set reachable from P'_B contains a root state r, the set reachable from P_B will also contain r. This can be even more optimized by the approach of [ACH$^+$10], which uses the upward simulation relation to weaken the conditions for removing a pair from the set of reachable states. The mentioned optimisations in practice often prove or refute inclusion by constructing a small part of the product automaton only[1]. We denote these algorithms as *upward* algorithms to reflect the direction in which they traverse automata $\mathcal{A}_S$ and $\mathcal{A}_B$.

The upward algorithms are sufficiently efficient in many practical cases. However, they have two drawbacks: (i) When generating the bottom-up post-image of a set $\mathcal{S}$ of macrostates (which are sets of states of $\mathcal{A}_B$), all possible n-tuples of states from all possible products $S_1 \times \ldots \times S_n$ where $S_i \in \mathcal{S}$ need to be enumerated[2]. (ii) Moreover, these algorithms are known to be compatible with only upward simulations as a means of their possible optimisation, which is a disadvantage since downward simulations are often much richer and also cheaper to compute.

Downward inclusion checking. The alternative *downward* approach to checking TA language inclusion was first proposed in [HVP05] in the context of subtyping of XML types. With hindsight, we can consider it as an on-the-fly version of the algorithm for constructing the *difference automaton* for a pair of TAs, proposed by Hosoya [Hos11]. The inclusion algorithm is not derivable from the textbook approach and has a more complex structure with its own weak points; nevertheless, it does not suffer from the two issues of the upward algorithm mentioned above. We generalise the algorithm of [HVP05] for automata over alphabets with an arbitrary rank ([HVP05] considers rank at most two), and, most importantly, we improve it significantly by using the antichain principle, empowered by a use of the cheap and usually large downward simulation. In this way, we obtain an algorithm which is complementary to and highly competitive with the upward algorithm as shown by our experimental results (in which the newly proposed algorithm significantly dominates in most of the considered cases).

[1] The work of [TH03] does, in fact, not use the terminology of antichains despite implementing them in a symbolic, BDD-based way. It specialises to binary tree automata only. A more general introduction of antichains within a lattice-theoretic framework appeared in the context of finite word automata in [WDHR06]. Subsequently, [BHH$^+$08] generalised [WDHR06] for explicit upward inclusion checking on TAs and experimentally advocated its use within the abstract regular tree model checking framework [BHH$^+$08]. See also [DR10] for other combinations of antichains and simulations for finite word automata.

[2] Note that this can be slightly optimised by a technique presented in Chapter 10.

Outline. The rest of this chapter is organised as follows. Section 8.1 describes our basic downward inclusion checking algorithm, followed by Section 8.2 that contains a description of its further optimisations. Section 8.3 presents experimental comparison of the downward algorithms with the upward algorithms, and Section 8.4 concludes the chapter.

8.1. Downward Inclusion Checking

Let us fix two tree automata $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, R_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, R_B)$ for which we want to check whether the language inclusion $L(\mathcal{A}_S) \subseteq L(\mathcal{A}_B)$ holds. If we try to answer this query top-down and we proceed in a naïve way, we immediately realise that the fact that the top-down successors of particular states are *tuples* of states leads us to checking inclusion of the languages of tuples of states. Subsequently, the need to compare the languages of each corresponding pair of states in these tuples will again lead to comparing the languages of tuples of states, and hence, we end up comparing the languages of *tuples of tuples* of states, and the need to deal with more and more nested tuples of states never stops.

For instance, given a transition $q \to a(p_1, p_2)$ in $\mathcal{A}_S$, transitions $r \to a(s_1, s_2)$ and $r \to a(t_1, t_2)$ in $\mathcal{A}_B$, and assuming that there are no further top-down transitions from q and r, it holds that $L(q) \subseteq L(r)$ if and only if $L((p_1, p_2)) \subseteq L((s_1, s_2)) \cup L((t_1, t_2))$. Note that the union $L((s_1, s_2)) \cup L((t_1, t_2))$ cannot be computed component-wise, this is, $L((s_1, s_2)) \cup L((t_1, t_2)) \neq (L(s_1) \cup L(t_1)) \times (L(s_2) \cup L(t_2))$. For instance, provided $L(s_1) = L(s_2) = \{b\}$ and $L(t_1) = L(t_2) = \{c\}$, it holds that $L((s_1, s_2)) \cup L((t_1, t_2)) = \{(b, b), (c, c)\}$, but the component-wise union is a larger set $(L(s_1) \cup L(t_1)) \times (L(s_2) \cup L(t_2)) = \{(b, b), (b, c), (c, b), (c, c)\}$. Hence, we cannot simply check whether $L(p_1) \subseteq L(s_1) \cup L(t_1)$ and $L(p_2) \subseteq L(s_2) \cup L(t_2)$ to answer the original query, and we have to proceed by checking inclusion on the obtained tuples of states. However, exploring the top-down transitions that lead from the states that appear in these tuples will lead us to dealing with tuples of tuples of states, etc.

Fortunately, there is a way out of the above trap. In particular, as first observed in [HVP05] in the context of XML type checking, we can exploit the following property of the Cartesian product of sets $G, H \subseteq \mathcal{U}$ for a universe $\mathcal{U}$:

$$G \times H = (G \times \mathcal{U}) \cap (\mathcal{U} \times H). \tag{8.1}$$

Continuing in our example, this means that we can rewrite the expression

$$L(p_1) \times L(p_2) \subseteq L((s_1, s_2)) \cup L((t_1, t_2)), \tag{8.2}$$

which is equivalent to

$$L(p_1) \times L(p_2) \subseteq (L(s_1) \times L(s_2)) \cup (L(t_1) \times L(t_2)), \tag{8.3}$$

as the expression

$$\begin{aligned} L(p_1) \times L(p_2) \quad \subseteq \quad & \big((L(s_1) \times T_\Sigma) \cap (T_\Sigma \times L(s_2))\big) \quad \cup \\ & \big((L(t_1) \times T_\Sigma) \cap (T_\Sigma \times L(t_2))\big). \end{aligned} \tag{8.4}$$

This can further be rewritten, using the distributive laws in the $(2^{T_\Sigma \times T_\Sigma}, \subseteq)$ lattice, as

$$\begin{aligned} L(p_1) \times L(p_2) \quad \subseteq \quad & \big((L(s_1) \times T_\Sigma) \cup (L(t_1) \times T_\Sigma)\big) \quad \cap \\ & \big((L(s_1) \times T_\Sigma) \cup (T_\Sigma \times L(t_2))\big) \quad \cap \\ & \big((T_\Sigma \times L(s_2)) \cup (L(t_1) \times T_\Sigma)\big) \quad \cap \\ & \big((T_\Sigma \times L(s_2)) \cup (T_\Sigma \times L(t_2))\big). \end{aligned} \tag{8.5}$$

It is easy to see that inclusion between a set and an intersection of several sets holds exactly if it holds for all components of the intersection. In our example, this means that the inclusion from Equation 8.5 holds if and only if the following formula is true:

$$\begin{aligned} L(p_1) \times L(p_2) \quad &\subseteq \quad \big((L(s_1) \times T_\Sigma) \cup (L(t_1) \times T_\Sigma)\big) \quad \wedge \\ L(p_1) \times L(p_2) \quad &\subseteq \quad \big((L(s_1) \times T_\Sigma) \cup (T_\Sigma \times L(t_2))\big) \quad \wedge \\ L(p_1) \times L(p_2) \quad &\subseteq \quad \big((T_\Sigma \times L(s_2)) \cup (L(t_1) \times T_\Sigma)\big) \quad \wedge \\ L(p_1) \times L(p_2) \quad &\subseteq \quad \big((T_\Sigma \times L(s_2)) \cup (T_\Sigma \times L(t_2))\big). \end{aligned} \tag{8.6}$$

Two things should be noted in the previous formula.

1. If we are computing the union of languages of a pair of tuples such that they have T_Σ at all indices other than some index i, we can compute it component-wise, i.e. the inclusion test

   $$L(p_1) \times L(p_2) \subseteq ((L(s_1) \times T_\Sigma) \cup (L(t_1) \times T_\Sigma)) \tag{8.7}$$

   can be simplified to the test

   $$L(p_1) \times L(p_2) \subseteq (L(s_1) \cup L(t_1)) \times T_\Sigma. \tag{8.8}$$

   Because $L(p_2)$ is always a subset of T_Σ, the above clearly holds iff $L(p_1) \subseteq L(s_1) \cup L(t_1)$.

2. If T_Σ does not appear at the same positions as in the inclusion

   $$L(p_1) \times L(p_2) \subseteq ((L(s_1) \times T_\Sigma) \cup (T_\Sigma \times L(t_2)))\,, \tag{8.9}$$

   it must hold that either

   $$L(p_1) \subseteq L(s_1) \qquad \text{or} \qquad L(p_2) \subseteq L(t_2). \tag{8.10}$$

Using the above observation and Equation 8.6, we can finally rewrite the equation

$$L(p_1) \times L(p_2) \subseteq L((s_1, s_2)) \cup L((t_1, t_2)) \tag{8.11}$$

into the following formula, which does not contain languages of tuples but of single states only:

$$\begin{array}{llll} L(p_1) \subseteq L(s_1) \cup L(t_1) & & & \wedge \\ \big(L(p_1) \subseteq L(s_1) & \vee & L(p_2) \subseteq L(t_2)\big) & \wedge \\ \big(L(p_1) \subseteq L(t_1) & \vee & L(p_2) \subseteq L(s_2)\big) & \wedge \\ & & L(p_2) \subseteq L(s_2) \cup L(t_2). & \end{array} \tag{8.12}$$

The above reasoning can be generalised to dealing with transitions of any arity as shown in Theorem 8.1. In the theorem, we conveniently exploit the notion of *choice functions*. Given $P_B \subseteq Q_B$ and $a \in \Sigma$, $\#a = n \geq 1$, we denote by $cf(P_B, a)$ the set of all choice functions f that assign an index i, $1 \leq i \leq n$, to all n-tuples $(q_1, \ldots, q_n) \in Q_B^n$ such that there exists a state in P_B that can make a top-down transition over a to $(q_1, \ldots, q_n)$; formally, $cf(P_B, a) = \{f \mid f : down_a(P_B) \to \{1, \ldots, \#a\}\}$.

Theorem 8.1. *Let $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, R_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, R_B)$ be tree automata. For sets $P_S \subseteq Q_S$ and $P_B \subseteq Q_B$ it holds that $L(P_S) \subseteq L(P_B)$ if and only if $\forall p_S \in P_S, \forall a \in \Sigma$: if $p_S \to a(r_1, \ldots, r_{\#a})$,*

$$\textit{then} \quad \begin{cases} down_a(P_B) = \{()\} & \textit{if } \#a = 0, \\ \forall f \in cf(P_B, a), \exists 1 \leq i \leq \#a : L(r_i) \subseteq \bigcup\limits_{\substack{\overline{u} \in down_a(P_B) \\ f(\overline{u}) = i}} L(u_i) & \textit{if } \#a > 0. \end{cases}$$

Proof. For two sets $P_S \subseteq Q_S, P_B \subseteq Q_B$, it clearly holds that $L(P_S) \subseteq L(P_B)$ if and only if $\forall p_S \in P_S, \forall a \in \Sigma$:

$$p_S \to a(r_1, \ldots, r_n) \implies L((r_1, \ldots, r_n)) \subseteq \bigcup_{(u_1, \ldots, u_n) \in down_a(P_B)} L((u_1, \ldots, u_n)). \tag{8.13}$$

For the case when $\#a = 0$, the above formula collapses to

$$p_S \to a() \implies L(()) \subseteq \bigcup_{() \in down_a(P_B)} L(()). \tag{8.14}$$

Since $down_a(P_B) \subseteq \{()\}$ for $\#a = 0$, the first part of the theorem is proven. We prove the second part (when $\#a > 0$) in the following steps. Let us fix $n = \#a$, $\overline{u} = (u_1, \ldots, u_n)$, $\overline{r} = (r_1, \ldots, r_n)$. Then we can observe that the inclusion

$$L((r_1, \ldots, r_n)) \subseteq \bigcup_{\overline{u} \in down_a(P_B)} L((u_1, \ldots, u_n)) \tag{8.15}$$

is equivalent to the inclusion

$$\prod_{i=1}^{n} L(r_i) \subseteq \bigcup_{\overline{u} \in down_a(P_B)} \prod_{i=1}^{n} L(u_i), \tag{8.16}$$

where $\prod_{i=1}^{n} S_i$ denotes the Cartesian product of a family of sets $\{S_1, \ldots, S_n\}$. We can further observe that for a universe $\mathcal{U}$ and a family of sets $\{S_1, \ldots, S_n\}$ such that $S_i \subseteq \mathcal{U}$ for all $1 \leq i \leq n$, it holds that

$$\prod_{i=1}^{n} S_i = \bigcap_{i=1}^{n} \left[\mathcal{U}^{i-1} \times S_i \times \mathcal{U}^{n-i}\right]. \tag{8.17}$$

Given the family of sets $\{L(u_1), \ldots, L(u_n)\}$ and the decomposition from Equation 8.17, we can rewrite the formula from Equation 8.16 as

$$\prod_{i=1}^{n} L(r_i) \subseteq \bigcup_{\overline{u} \in down_a(P_B)} \left[\bigcap_{i=1}^{n} \left[T_\Sigma^{i-1} \times L(u_i) \times T_\Sigma^{n-i} \right] \right]. \tag{8.18}$$

Since the power set lattice $(2^{Q_B}, \subseteq)$ is completely distributive, we can exploit the fact that for any doubly indexed set $\{x_{j,k} \in 2^{Q_B} \mid j \in J, k \in K_j\}$ it holds

$$\bigcup_{j \in J} \bigcap_{k \in K_j} x_{j,k} = \bigcap_{f \in F} \bigcup_{j \in J} x_{j,f(j)} \tag{8.19}$$

where F is the set of all choice functions f choosing for each index $j \in J$ some index $f(j) \in K_j$. For our purpose, we introduce the set of choice functions:

$$cf(P_B, a) = \{f \mid f : down_a(P_B) \to \{1, \ldots, n\}\} \tag{8.20}$$

where every f assigns to every tuple from $down_a(P_B)$ an index. Therefore, after applying the distributive law on Equation 8.18, we obtain

$$\prod_{i=1}^{n} L(r_i) \subseteq \bigcap_{f \in cf(P_B,a)} \left[\bigcup_{\overline{u} \in down_a(P_B)} \left[T_\Sigma^{f(\overline{u})-1} \times L(u_{f(\overline{u})}) \times T_\Sigma^{n-f(\overline{u})} \right] \right]. \tag{8.21}$$

Due to the fact that for a universe $\mathcal{U}$, a set $T \subseteq \mathcal{U}$ in this universe, and an intersection of a family of sets $R \subseteq 2^{\mathcal{U}}$, it holds that

$$T \subseteq \bigcap_{S_i \in R} S_i \iff \forall S_i \in R : T \subseteq S_i, \tag{8.22}$$

we can simplify our case to

$$\forall f \in cf(P_B, a) : \prod_{i=1}^{n} L(r_i) \subseteq \bigcup_{\overline{u} \in down_a(P_B)} \left[T_\Sigma^{f(\overline{u})-1} \times L(u_{f(\overline{u})}) \times T_\Sigma^{n-f(\overline{u})} \right]. \tag{8.23}$$

Further, observe that for a fixed choice function f, we can use f to split the tuples from $down_a(P_B)$ into n sets, each of them containing tuples $\overline{u}$ that are assigned by f the same index $i = f(\overline{u})$. We can then rewrite the right-hand side of the previous inclusion query to the following:

$$\bigcup_{\overline{u} \in down_a(P_B)} \left[T_\Sigma^{f(\overline{u})-1} \times L(u_{f(\overline{u})}) \times T_\Sigma^{n-f(\overline{u})} \right] = \tag{8.24}$$

$$\bigcup_{i=1}^{n} \left[\bigcup_{\substack{\overline{u} \in down_a(P_B) \\ f(\overline{u})=i}} \left[T_\Sigma^{i-1} \times L(u_i) \times T_\Sigma^{n-i} \right] \right] = \tag{8.25}$$

$$\bigcup_{i=1}^{n} \left[T_\Sigma^{i-1} \times \left[\bigcup_{\substack{\overline{u} \in down_a(P_B) \\ f(\overline{u})=i}} L(u_i) \right] \times T_\Sigma^{n-i} \right] \tag{8.26}$$

It can be observed that for a universe $\mathcal{U}$ and two families of sets $\{S_1, \ldots, S_n\}$ and $\{S'_1, \ldots, S'_n\}$ such that $S_i, S'_i \subseteq \mathcal{U}$ for all $1 \leq i \leq n$, it holds that

$$\prod_{i=1}^{n} S_i \subseteq \bigcup_{i=1}^{n} [\mathcal{U}^{i-1} \times S'_i \times \mathcal{U}^{n-i}] \qquad \text{iff} \qquad \exists 1 \leq i \leq n : S_i \subseteq S'_i. \tag{8.27}$$

We can now finally deduce that the formula

$$\forall f \in F : \prod_{i=1}^{n} L(r_i) \subseteq \bigcup_{i=1}^{n} \left[T_\Sigma^{i-1} \times \left[\bigcup_{\substack{\overline{u} \in down_a(P_B) \\ f(\overline{u})=i}} L(u_i) \right] \times T_\Sigma^{n-i} \right] \tag{8.28}$$

is equivalent to the formula

$$\forall f \in F, \exists 1 \leq i \leq n : L(r_i) \subseteq \bigcup_{\substack{\overline{u} \in down_a(P_B) \\ f(\overline{u})=i}} L(u_i), \tag{8.29}$$

which concludes the proof. □

8.1.1. Basic Algorithm for Downward Inclusion Checking

Next, we construct a basic algorithm for downward inclusion checking on tree automata $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, R_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, R_B)$. The algorithm is shown as Algorithm 8.1. Its main idea relies on a recursive application of Theorem 8.1 in function `expand1`. The function is given a pair $(p_S, P_B) \in Q_S \times 2^{Q_B}$ for which we want to prove that $L(p_S) \subseteq L(P_B)$—initially, the function is called for every pair (q_S, F_B) where $q_S \in F_S$. The function enumerates all possible top-down transitions that $\mathcal{A}_S$ can do from p_S (lines 3–8). For each such transition, the function either checks whether there is some transition $p_B \to a()$ for $p_B \in P_B$ if $\#a = 0$ (line 5), or it starts enumerating and recursively checking queries $L(p'_S) \subseteq L(P'_B)$ on which the result of $L(p_S) \subseteq L(P_B)$ depends according to Theorem 8.1 (lines 9–16).

The `expand1` function keeps track of which inclusion queries are currently being evaluated in the set *workset* (line 2). Encountering a query $L(p'_S) \subseteq L(P'_B)$ with $(p'_S, P'_B) \in$ *workset* means that the result of $L(p'_S) \subseteq L(P'_B)$ depends on the result of $L(p'_S) \subseteq L(P'_B)$ itself. In this case, the function immediately successfully returns because the result of the query then depends only on the other branches of the call tree.

Algorithm 8.1: Downward inclusion

```
Input: TAs $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, R_S), \mathcal{A}_B = (Q_B, \Sigma, \Delta_B, R_B)$
Output: $true$ if $L(\mathcal{A}_S) \subseteq L(\mathcal{A}_B)$, $false$ otherwise
1  foreach $q_S \in R_S$ do
2      if $\neg$expand1($q_S, R_B, \emptyset$) then return $false$;
3  return $true$;
```

Function expand1(p_S, P_B, *workset*)

```
   // $p_S \in Q_S$, $P_B \subseteq Q_B$, and $workset \subseteq Q_S \times 2^{Q_B}$
1  if $(p_S, P_B) \in workset$ then return $true$;
2  $workset := workset \cup \{(p_S, P_B)\}$;
3  foreach $a \in \Sigma$ do
4      if $\#a = 0$ then
5          if $down_a(p_S) \neq \emptyset \wedge down_a(P_B) = \emptyset$ then return $false$ ;
6      else
7          $W := down_a(P_B)$;
8          foreach $(r_1, \ldots, r_{\#a}) \in down_a(p_S)$ do          // $p_S \to a(r_1, \ldots, r_{\#a})$
9              foreach $f \in \{W \to \{1, \ldots, \#a\}\}$ do          // $\forall f \in cf(P_B, a)$
10                 $found := false$;
11                 foreach $1 \leq i \leq \#a$ do                    // $\exists 1 \leq i \leq \#a$
12                     $S := \{q_i \mid (q_1, \ldots, q_{\#a}) \in W, f((q_1, \ldots, q_{\#a})) = i\}$;
13                     if expand1($r_i, S, workset$) then            // if $L(r_i) \subseteq L(S)$
14                         $found := true$;
15                         break;
16                 if $\neg found$ then return $false$;
17 return $true$;
```

Using Theorem 8.1 and noting that Algorithm 8.1 necessarily terminates because all its loops are bounded, and the recursion in function `expand1` is also bounded due to the use of *workset*, it is not difficult to see that the following theorem holds.

Theorem 8.2. *When applied on a pair of TAs $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, R_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, R_B)$ s.t. $Q_S \cap Q_B = \emptyset$, Algorithm 8.1 terminates and returns true if and only if $L(\mathcal{A}_S) \subseteq L(\mathcal{A}_B)$.*

8.2. Optimisations of Downward Inclusion Checking

In this section, we propose several optimisations of the basic algorithm presented above that, according to our experiments, often have a huge impact on the efficiency of the algorithm—making it in many cases the most efficient algorithm for checking inclusion

Algorithm 8.2: Downward inclusion (antichains + preorder)

```
  Input: TAs $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, R_S), \mathcal{A}_B = (Q_B, \Sigma, \Delta_B, R_B)$,
         preorder $\preceq\ \subseteq (Q_S \cup Q_B)^2$
  Output: $true$ if $L(\mathcal{A}_S) \subseteq L(\mathcal{A}_B)$, $false$ otherwise
  Data: $NN := \emptyset$;
1 foreach $q_S \in R_S$ do
2     if $\neg$expand2$(q_S, R_B, \emptyset)$ then return $false$;
3 return $true$;
```

on tree automata that we are currently aware of. In general, the optimisations are based on an original use of simulations and antichains in a way suitable for the context of downward inclusion checking.

In what follows, we assume that there is available a preorder $\preceq \subseteq (Q_S \cup Q_B)^2$ compatible with language inclusion, i.e. such that $p \preceq q \Longrightarrow L(p) \subseteq L(q)$, and we use $P \preceq^{\forall\exists} R$ where $P, R \subseteq (Q_S \cup Q_B)^2$ to denote that $\forall p \in P\, \exists r \in R : p \preceq r$. An example of such a preorder, which can be efficiently computed, is the maximal downward simulation $\preceq_D$ (see [HŠ09]).

8.2.1. Optimisation with Antichains and Simulation-based Pruning

First, we propose the following concrete optimisations of the downward checking of $L(p_S) \subseteq L(P_B)$:

a) If there exists a state $p_B \in P_B$ such that $p_S \preceq p_B$, then the inclusion clearly holds (from the assumption made about $\preceq$), and no further checking is needed.

b) Next, it can be seen without any further computation that the inclusion does *not* hold if there exists some (p'_S, P'_B) such that $p'_S \preceq p_S$ and $P_B \preceq^{\forall\exists} P'_B$, and we have already established that $L(p'_S) \not\subseteq L(P'_B)$. Indeed, we have $L(P_B) \subseteq L(P'_B) \not\supseteq L(p'_S) \subseteq L(p_S)$, and therefore $L(p_S) \not\subseteq L(P_B)$.

c) Finally, we can stop evaluating the given inclusion query if there is some $(p'_S, P'_B) \in$ *workset* such that $p_S \preceq p'_S$ and $P'_B \preceq^{\forall\exists} P_B$. Indeed, this means that the result of $L(p'_S) \subseteq L(P'_B)$ depends on the result of $L(p_S) \subseteq L(P_B)$. However, if $L(p'_S) \subseteq L(P'_B)$ holds, then also $L(p_S) \subseteq L(P_B)$ holds because we have $L(p_S) \subseteq L(p'_S) \subseteq L(P'_B) \subseteq L(P_B)$. On the other hand, if $L(p'_S) \subseteq L(P'_B)$ does *not* hold, the path between (p'_S, P'_B) and (p_S, P_B) cannot be the only reason for that since a counterexample has not been found on that path yet, and the chance of finding a counterexample is only smaller from (p_S, P_B).

The version of Algorithm 8.1 including all the above proposed optimisations is shown as Algorithm 8.2 (the changes are highlighted in the pseudocode). The optimisations can be found in the function `expand2` that replaces the function `expand1`. In particular, line 2 implements optimisation (a), line 1 optimisation (b), and line 3 optimisation (c). In order

Function expand2(p_S, P_B, *workset*)

```
// $p_S \in Q_S$, $P_B \subseteq Q_B$, and $workset \subseteq Q_S \times 2^{Q_B}$
if $\exists (p'_S, P'_B) \in NN : p'_S \preceq p_S \wedge P_B \preceq^{\forall\exists} P'_B$ then return $false$ ;
if $\exists p \in P_B : p_S \preceq p$ then return $true$ ;
if $\exists (p'_S, P'_B) \in workset : p_S \preceq p'_S \wedge P'_B \preceq^{\forall\exists} P_B$ then return $true$ ;
$workset := workset \cup \{(p_S, P_B)\}$;
foreach $a \in \Sigma$ do
    if $\#a = 0$ then
        if $down_a(p_S) \neq \emptyset \wedge down_a(P_B) = \emptyset$ then return $false$ ;
    else
        $W := down_a(P_B)$;
        foreach $(r_1, \ldots, r_{\#a}) \in down_a(p_S)$ do          // $p_S \to a(r_1, \ldots, r_{\#a})$
            foreach $f \in \{W \to \{1, \ldots, \#a\}\}$ do          // $\forall f \in cf(P_B, a)$
                $found := false$;
                foreach $1 \leq i \leq \#a$ do                       // $\exists 1 \leq i \leq \#a$
                    $S := \{q_i \mid (q_1, \ldots, q_{\#a}) \in W, f((q_1, \ldots, q_{\#a})) = i\}$;
                    if expand2($r_i, S, workset$) then               // if $L(r_i) \subseteq L(S)$
                        $found := true$;
                        break ;
                    if $\nexists (r', H) \in NN : r' \preceq r_i \wedge S \preceq^{\forall\exists} H$ then
                        $NN := (NN \setminus \{(r', H) \mid H \preceq^{\forall\exists} S, r_i \preceq r'\}) \cup \{(r_i, S)\}$;
                if $\neg found$ then return $false$;
return $true$;
```

to implement optimisation (b), the algorithm maintains a new set NN. This set stores pairs (p_S, P_B) for which it has already been shown that the inclusion $L(p_S) \subseteq L(P_B)$ does not hold.

As a further optimisation, the set NN is maintained as an antichain w.r.t. the preorder that compares the pairs stored in NN such that the states from Q_S on the left are compared w.r.t. $\preceq$, and the sets from 2^{Q_B} on the right are compared w.r.t. $\succeq^{\exists\forall}$ (line 19). Clearly, there is no need to store a pair (p_S, P_B) that is bigger in the described sense than some other pair (p'_S, P'_B) since every time (p_S, P_B) can be used to prune the search, (p'_S, P'_B) can also be used.

Taking into account Theorem 8.2 and the above presented facts, it is not difficult to see that the following holds.

Theorem 8.3. *When applied on a pair of TAs $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, R_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, R_B)$ s.t. $Q_S \cap Q_B = \emptyset$, Algorithm 8.2 terminates and returns true if and only if $L(\mathcal{A}_S) \subseteq L(\mathcal{A}_B)$.*

Algorithm 8.3: Downward inclusion (antichains + preorder + *IN*)

```
    Input: TAs $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, R_S), \mathcal{A}_B = (Q_B, \Sigma, \Delta_B, R_B)$,
           preorder $\preceq\ \subseteq (Q_S \cup Q_B)^2$
    Output: $true$ if $L(\mathcal{A}_S) \subseteq L(\mathcal{A}_B)$, $false$ otherwise
    Data: $NN := \emptyset$; $IN := \emptyset$;
1   foreach $q_S \in R_S$ do
2       if expand2e$(q_S, R_B, \emptyset) = (false, \_, \_)$ then return $false$;
3   return $true$;
```

8.2.2. Optimisation with Caching of Inclusion Pairs

The algorithm from the previous section can be optimised even more. Recall that the algorithm caches pairs for which the inclusion does not hold, i.e. pairs (p_S, P_B) such that $L(p_S) \not\subseteq L(P_B)$, in the set NN (which is maintained as an antichain). A natural question that arises is whether there is a similar option for pairs for which the inclusion does hold, i.e. pairs (p_S, P_B) such that $L(p_S) \subseteq L(P_B)$. Such an option indeed exists and is presented in the rest of this section.

Let us denote the set of the above-mentioned pairs for which the inclusion holds as IN. Then, when checking the inclusion $L(p_S) \subseteq L(P_B)$, when there is a pair $(p'_S, P'_B) \in IN$ such that $p_S \preceq p'_S$ and $P'_B \preceq^{\forall\exists} P_B$, then we immediately know that the checked inclusion holds because $L(p_S) \subseteq L(p'_S) \subseteq L(P'_B) \subseteq L(P_B)$.

The set IN can again be optimised as an antichain but with the opposite ordering than NN. This means that there are no two pairs $(p_S, P_B), (p'_S, P'_B)$ such that $p_S \preceq p'_S$ and $P'_B \preceq^{\forall\exists} P_B$ in IN. It is easy to understand that a pair (p_S, P_B) does not have to be stored since whenever (p_S, P_B) can be used to prune the search, (p'_S, P'_B) can also be used.

However, adding new pairs to IN is not as straightforward as for NN. Assume that we add a pair (p_S, P_B) to IN immediately when the function call `expand2`$(p_S, P_B, workset)$ at line 15 of function `expand2` returns *true* for some *workset*. This is not correct as shown in the following example.

Suppose that when checking inclusion $L(p'_S) \subseteq L(P'_B)$, a test for inclusion $L(p_S) \subseteq L(P_B)$ where $p_S \preceq p'_S$ and $P'_B \preceq^{\forall\exists} P_B$ is encountered somewhere deep in the recursive calls of `expand2`. As stated previously, the inclusion $L(p_S) \subseteq L(P_B)$ does not need to be tested since if $L(p'_S) \subseteq L(P'_B)$, then $L(p_S) \subseteq L(P_B)$, and if $L(p'_S) \not\subseteq L(P'_B)$, then this cannot be caused solely by $L(p_S) \not\subseteq L(P_B)$. Hence, `expand2`$(p_S, P_B, workset)$ returns *true*, and the result of the query $L(p'_S) \subseteq L(P'_B)$ will be given by other branches of the call tree generated for the $L(p'_S) \subseteq L(P'_B)$ query. However, if we put the pair (p_S, P_B) into IN and later proved that $L(p'_S) \not\subseteq L(P'_B)$, then the set IN would become invalid.

A solution to this issue is given in Algorithm 8.3 (the changes from Algorithm 8.2 are highlighted). The `expand2e` function is a modified version of `expand2` that additionally returns a formula of the form $\bigwedge Ant \to \bigwedge Con$ where Con (*consequents*) is a set of inclusion queries that can be answered positively provided that the inclusion queries in Ant (*antecedents*) are all answered positively.

Function expand2e(p_S, P_B, $workset$)

```
1  if $\exists (p'_S, P'_B) \in IN : p_S \preceq p'_S \wedge P'_B \preceq^{\forall\exists} P_B$ then return $(true, \emptyset, \emptyset)$ ;
2  if $\exists (p'_S, P'_B) \in NN : p'_S \preceq p_S \wedge P_B \preceq^{\forall\exists} P'_B$ then return $(false, \emptyset, \emptyset)$ ;
3  if $\exists p \in P_B : p_S \preceq p$ then return $(true, \emptyset, \emptyset)$ ;
4  if $\exists (p'_S, P'_B) \in workset : p_S \preceq p'_S \wedge P'_B \preceq^{\forall\exists} P_B$ then
5      return $(true, \{(p'_S, P'_B)\}, \emptyset)$;
6  $workset := workset \cup \{(p_S, P_B)\}$; $Ant := \emptyset$; $Con := \emptyset$;
7  foreach $a \in \Sigma$ do
8      if $\#a = 0$ then
9          if $down_a(p_S) \neq \emptyset \wedge down_a(P_B) = \emptyset$ then return $(false, \emptyset, \emptyset)$ ;
10     else
11         $W := down_a(P_B)$;
12         foreach $(r_1, \dots, r_{\#a}) \in down_a(p_S)$ do                // $p_S \to a(r_1, \dots, r_{\#a})$
13             foreach $f \in \{W \to \{1, \dots, \#a\}\}$ do               // $\forall f \in cf(P_B, a)$
14                 $found := false$;
15                 foreach $1 \leq i \leq \#a$ do                           // $\exists 1 \leq i \leq \#a$
16                     $S := \{q_i \mid (q_1, \dots, q_{\#a}) \in W, f((q_1, \dots, q_{\#a})) = i\}$ ;
17                     $(x, Ant', Con') :=$ expand2e$(r_i, S, workset)$;
18                     if $x$ then                                          // if $L(r_i) \subseteq L(S)$
19                         $found := true$; $Ant := Ant \cup Ant'$;
20                         $Con := Con \cup Con'$; break;
21                     if $\nexists (r', H) \in NN : r' \preceq r_i \wedge S \preceq^{\forall\exists} H$ then
22                         $NN := (NN \setminus \{(r', H) \mid H \preceq^{\forall\exists} S, r_i \preceq r'\}) \cup \{(r_i, S)\}$;
23                 if $\neg found$ then return $(false, \emptyset, \emptyset)$ ;
24 $Ant := Ant \setminus \{(p_S, P_B)\}$; $Con := Con \cup \{(p_S, P_B)\}$;
25 if $Ant = \emptyset$ then
26     foreach $(x, Y) \in Con$ do
27         if $\nexists (p'_S, P'_B) \in IN : x \preceq p'_S \wedge P'_B \preceq^{\forall\exists} Y$ then
28             $IN := (IN \setminus \{(r', H) \mid Y \preceq^{\forall\exists} H, r' \preceq x\}) \cup \{(x, Y)\}$;
29     $Con := \emptyset$;
30 return $(true, Ant, Con)$;
```

When the recursive call of expand2e$(p_S, P_B, workset)$ is at the bottom of the call tree and there is $(p'_S, P'_B) \in workset$ such that $p_S \preceq p'_S$ and $P'_B \preceq^{\forall\exists} P_B$ (line 4), then, according to the above, the formula returned from expand2e along with *true* could be $\bigwedge\{L(p'_S) \subseteq L(P'_B)\} \to \bigwedge\{L(p_S) \subseteq L(P_B)\}$ because $L(p_S) \subseteq L(P_B)$ cannot be considered guaranteed before $L(p'_S) \subseteq L(P'_B)$ is positively answered. This formula is, however, simplified to $\bigwedge\{L(p'_S) \subseteq L(P'_B)\} \to \emptyset$ since $L(p_S) \subseteq L(P_B)$ can be forgotten as it is weaker than $L(p'_S) \subseteq L(P'_B)$.

A situation similar to what we have just discussed arises when the recursive call of `expand2e`$(p_S, P_B, \mathit{workset})$ is at the bottom of the call tree and there is $(p'_S, P'_B) \in \mathit{IN}$ such that $p_S \preceq p'_S$ and $P'_B \preceq^{\forall\exists} P_B$ (line 1). In this case, $\bigwedge \emptyset \to \bigwedge \emptyset$ is returned (along with *true*) since the validity of $L(p'_S) \subseteq L(P'_B)$ has already been established. Next, if the recursive call of `expand2e`$(p_S, P_B, \mathit{workset})$ is at the bottom of the call tree and there is $p \in P_B$ such that $p_S \preceq p$ (line 3), $\bigwedge \emptyset \to \bigwedge \emptyset$ is again returned since for any inclusion query $L(p'_S) \subseteq L(P'_B)$ such that $p'_S \preceq p_S$ and $P_B \preceq^{\forall\exists} P'_B$, it will be the case that there is $p' \in P'_B$ such that $p'_S \preceq p'$ (and hence the computation will be immediately stopped without a need to use *IN* for this purpose). Finally, when `expand2e` returns *false* (line 2), it is accompanied by the formula $\bigwedge \emptyset \to \bigwedge \emptyset$, which, however, is ignored in this case and is returned just to make the result of `expand2e` have the same structure.

For inner nodes of the call tree, this is, nodes that correspond to function calls `expand2e`(p_S, P_B) that themselves call `expand2e`, all antecedents and consequents returned from successful nested calls are collected into sets *Ant* and *Con*. Then, the condition $L(p_S) \subseteq L(P_B)$ is removed from *Ant* (if it is there) and added to *Con* since it has just been proved that $L(p_S) \subseteq L(P_B)$ holds provided that the elements from $\mathit{Ant} \setminus \{L(p_S) \subseteq L(P_B)\}$ are later proved to also hold. When the set *Ant* becomes empty, yielding the formula $\bigwedge \emptyset \to \bigwedge \mathit{Con}$, all elements of *Con* can be added to *IN* (while respecting the antichain property of *IN*) and the set *Con* cleared.

Taking into account Theorem 8.3 and the above presented facts, it can be seen that the following holds.

Theorem 8.4. *When applied on a pair of TAs $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, R_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, R_B)$ s.t. $Q_S \cap Q_B = \emptyset$, Algorithm 8.3 terminates and returns true if and only if $L(\mathcal{A}_S) \subseteq L(\mathcal{A}_B)$.*

8.3. Experimental Results

We implemented Algorithm 8.1 (which we mark as `down` in what follows), Algorithm 8.2 with the maximum downward simulation as the input preorder (marked as `down+s`), and Algorithm 8.3 inside the VATA library (about which we give further details in Chapter 10). We provide two configurations of Algorithm 8.3 that differ in the input preorder: The first of them uses identity (we mark this configuration as `down-opt`), while the other also uses the maximum downward simulation (marked as `down-opt+s`. In the experiments, we evaluated the performance of the four algorithms with the algorithm for upward inclusion checking using antichains from [BHH$^+$08] (marked as `up`) and its modification that uses the maximum upward simulation parameterised by identity (proposed in [ACH$^+$10] and marked as `up+s` below), which are provided in VATA. The evaluation was testing language inclusion $L(\mathcal{A}) \overset{?}{\subseteq} L(\mathcal{B})$ of almost 2 000 tree automata pairs of different sizes (ranging from 50 to 1 000 states), including automata from the intermediate steps of abstract regular tree model checking of the algorithm for rebalancing red-black trees after insertion or deletion of a leaf node [BHH$^+$08]. The timeout was set to 30 s.

The results of the experiments are presented in Table 8.1. The table compares the methods according to the percentage of the cases in which they were the fastest when

Table 8.1.: Results of the experiments (timeout 30 s)

Algorithm	All pairs		$L(\mathcal{A}) \nsubseteq L(\mathcal{B})$		$L(\mathcal{A}) \subseteq L(\mathcal{B})$	
	Winner	Timeouts	Winner	Timeouts	Winner	Timeouts
`down`	36.35 %	32.51 %	39.85 %	26.01 %	0.00 %	90.80 %
`down+s`	4.15 %	18.27 %	0.00 %	20.31 %	47.28 %	0.00 %
`down-opt`	32.20 %	32.51 %	35.30 %	26.01 %	0.00 %	90.80 %
`down-opt+s`	3.15 %	18.27 %	0.00 %	20.31 %	35.87 %	0.00 %
`up`	24.14 %	0.00 %	24.84 %	0.00 %	16.85 %	0.00 %
`up+s`	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %

checking inclusion on the same automata pair, and also according to the percentage of timeouts. The set of results in the column labelled with "All pairs" contains data for all pairs.

We also checked the performance of the algorithms for cases when inclusion either *does* or *does not* hold in order to explore the ability of the algorithms to either find a counterexample in the case when inclusion does not hold, or prove the inclusion in case it does. The results below "$L(\mathcal{A}) \nsubseteq L(\mathcal{B})$" in the table are for the pairs $\mathcal{A}, \mathcal{B}$ where the inclusion does not hold, and the column under "$L(\mathcal{A}) \subseteq L(\mathcal{B})$" reports on the cases where the inclusion holds.

The results show that the overhead of computing upward simulation is too high in all the cases that we have considered, causing upward inclusion checking using simulation to be the slowest when the time for computing the simulation used by the algorithm is included[3]. Next, it can be seen that for each of the remaining approaches there are cases in which they win in a significant way. However, the downward approaches are clearly dominating in significantly more of our test cases (with the only exception being the case of small automata when the time of computing simulations is not included). On the other hand, it can be observed that for some particular cases, the more complex structure of the downward algorithms (which resembles an *And-Or* tree) causes an unmanageable state explosion and the algorithms timeout (in contrast to the upward algorithms, which always, though often slowly, terminate).

8.4. Conclusion

In this section, we proposed a new algorithm for checking language inclusion over non-deterministic TAs (based on the one from [HVP05]) that traverses automata in the downward manner and uses both antichains and simulations to optimise its computation. This algorithm is, according to our experimental results, mostly superior to the known upward algorithms.

[3]Note that `up+s` was winning over `up` in the experiments of [ACH+10] even with the time for computing simulation included, which seems to be caused by a much less efficient implementation of the antichains in the original algorithm.

One of the interesting future research directions would be an extension of the techniques used in the optimisations of the downward algorithm to the recently introduced technique for testing language equivalence of nondeterministic finite automata based on the so-called bisimulation up-to congruence [BP13]. Apart from that, it would be interesting to explore an efficient implementation of the data structure used for storing the antichain, e.g. symbolically using some BDD-like data structure, as e.g. in [TH03]. An interesting problem here is how to efficiently encode antichains based not on the subset inclusion but on a simulation relation.

9. Semi-symbolic Tree Automata

Certain important applications of TAs, such as formal verification of programs with complex dynamic data structures [BHRV12] or decision procedures of logics such as WSkS or MSO, require finite (tree) automata with very large alphabets. For instance, the automata manipulated by the decision procedure for WS1S in Chapter 7 use an alphabet of the size 2^n where n is the number of variables in the considered formula. Here, the common choice is to use the tree automata library of MONA [KMS02], which is based on representing transitions of TAs symbolically using *multi-terminal binary decision diagrams* (MTBDDs). The encoding used by MONA is, however, restricted to *deterministic* automata only. This implies a necessity of immediate determinisation after each operation over TAs that introduces nondeterminism and may, in turn, easily lead to a state space explosion. Despite the extensive engineering effort spent to optimise the implementation of MONA, the focus on deterministic automata significantly limits its applicability.

As a way to overcome this issue, in this chapter, we propose a semi-symbolic representation of *nondeterministic* TAs that generalises the one used by MONA, and we develop algorithms implementing the basic operations on TAs (such as computation of union, intersection, etc.) as well as more involved algorithms for computing simulations and for checking language inclusion (using simulations and antichains to optimise it) over the proposed representation.

Outline. The structure of this chapter is the following. In Section 9.1, we give our definitions of BDDs and MTBDDs. The two dual semi-symbolic encodings of TAs are presented in Section 9.2 and the algorithms for operations on TAs over these encodings are described in Section 9.3. Section 9.4 describes our implementation of an MTBDD library. Section 9.5 gives experimental results and, finally, Section 9.6 concludes the chapter.

9.1. Binary Decision Diagrams

Let $\mathbb{B} = \{0, 1\}$ be the set of Boolean values. A *Boolean function* of *arity* k is a function of the form $f : \mathbb{B}^k \to \mathbb{B}$. We extend the notion of Boolean functions to an arbitrary nonempty set S where a k-ary Boolean function extended to the domain set S is a function of the form $f : \mathbb{B}^k \to S$.

A *reduced ordered binary decision diagram* (ROBDD) [Bry86] r over a set of n Boolean variables $x_1, \ldots, x_n$ is a connected directed acyclic graph with a single *source node* (denoted as $r.root$) and at least one of the two *sink nodes* $\mathbf{0}$ and $\mathbf{1}$. We call *internal* the nodes which are not sink nodes. A function *Var* assigns each internal node a Boolean variable from the set $X = \{x_1, \dots, x_n\}$, ordered by the ordering $x_1 < x_2 < \cdots < x_n$. For every internal node v there exists a pair of outgoing edges labelled *low* and *high*. We denote by $v.low$ a node w and by $v.high$ a node z such that there exists a directed edge from v to w labelled by *low* and a directed edge from v to z labelled by *high* respectively. For each internal node v, it must hold that $\mathit{Var}(v) < \mathit{Var}(v.low)$ and $\mathit{Var}(v) < \mathit{Var}(v.high)$, and also $v.low \neq v.high$. A node v represents an n-ary Boolean function $[\![v]\!] : \mathbb{B}^n \to \mathbb{B}$ that assigns to each assignment to the Boolean variables in X a corresponding Boolean value defined in the following way (using $\overline{x}$ as an abbreviation for $x_1 \dots x_n$):

$$\begin{aligned} &[\![\mathbf{0}]\!] = \lambda\,\overline{x}\,.\,0, \\ &[\![\mathbf{1}]\!] = \lambda\,\overline{x}\,.\,1, \\ &[\![v]\!] = \begin{cases} \lambda\,\overline{x}\,.\,[\![v.low]\!](\overline{x}) & \text{if } x_i = 0 \\ \lambda\,\overline{x}\,.\,[\![v.high]\!](\overline{x}) & \text{if } x_i = 1 \end{cases} \quad \text{for} \quad \mathit{Var}(v) = x_i. \end{aligned} \tag{9.1}$$

For every pair of distinct nodes v and w, it further holds that they represent a different function, i.e. $[\![v]\!] \neq [\![w]\!]$. We say that an ROBDD r represents the Boolean function $[\![r]\!]$ defined as $[\![r]\!] = [\![r.root]\!]$. Dually, for a Boolean function f, we use $\langle f \rangle$ to denote the (unique up to isomorphism) ROBDD representing f, i.e. $f = [\![\langle f \rangle]\!]$ and $r = \langle [\![r]\!] \rangle$.

We generalise the standard *Apply* operation for manipulation of Boolean functions represented by ROBDDs in the following way: let op_1, op_2, and op_3 be in turn arbitrary unary, binary, and ternary Boolean functions. Then the functions *Apply1*, *Apply2*, and *Apply3* produce a new ROBDD that is defined for ROBDDs f, g, and h as follows:

$$\begin{aligned} \mathit{Apply1}(f, op_1) &= \langle \lambda\,\overline{x}\,.\,op_1\big([\![f]\!](\overline{x})\big) \rangle, \\ \mathit{Apply2}(f, g, op_2) &= \langle \lambda\,\overline{x}\,.\,op_2\big([\![f]\!](\overline{x}), [\![g]\!](\overline{x})\big) \rangle, \\ \mathit{Apply3}(f, g, h, op_3) &= \langle \lambda\,\overline{x}\,.\,op_3\big([\![f]\!](\overline{x}), [\![g]\!](\overline{x}), [\![h]\!](\overline{x})\big) \rangle. \end{aligned} \tag{9.2}$$

In practice, op_1, op_2, and op_3 can be implemented as functions with side-effects.

The notion of ROBDDs is further generalised to *multi-terminal binary decision diagrams* (MTBDDs) [CMZ+97]. MTBDDs are essentially the same data structures as ROBDDs, the only difference being the fact that the set of sink nodes is not restricted to two nodes. Instead, it can contain an arbitrary number of nodes labelled uniquely by elements of an arbitrary domain set S. All standard notions for ROBDDs can naturally be extended to MTBDDs. An MTBDD m then represents a Boolean function extended to S, $[\![m]\!] : \mathbb{B}^n \to S$. Further, the concept of *shared MTBDDs* is used. A shared MTBDD s is an MTBDD with multiple source nodes (or *roots*) that represents a mapping of every element of the set of roots R to a function induced by the MTBDD corresponding to the given root, $[\![s]\!] : R \to (\mathbb{B}^n \to S)$. We abuse notation and use $f(r)$ for a shared MTBDD f and a root $r \in R$ to denote the MTBDD $\langle [\![f]\!](r) \rangle$.

Apply operations for MTBDDs are extended in such a way that the MTBDDs for *Apply2* and *Apply3* may have different domain sets. Not only this, even the result of the *Apply* operation may be over a different domain set than any of the parameters. Formally, suppose a triple of MTBDDs: f (over a domain F), g (over a domain G),

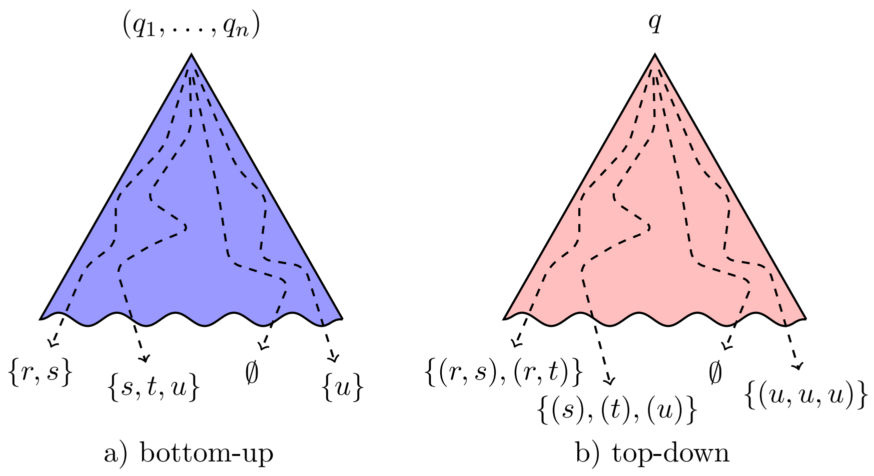


Figure 9.1.: The (a) bottom-up and (b) top-down semi-symbolic encodings of a transition relation. Paths in the MTBDD correspond to symbols from Σ.

and h (over a domain H). Further, assume a domain K for the resulting MTBDD. (Note that some of the considered domains may be identical.) Then, for $op_1 : F \to K$, $op_2 : (F \times G) \to K$, and $op_3 : (F \times G \times H) \to K$, the results of $\mathit{Apply1}(f, op_1)$, $\mathit{Apply2}(f, g, op_2)$, and $\mathit{Apply3}(f, g, h, op_3)$ are all MTBDDs over the domain K.

9.2. Semi-Symbolic Representations of Tree Automata

We next consider a natural, semi-symbolic, MTBDD-based encoding of nondeterministic TAs, suitable for handling automata with huge alphabets. A shared MTBDD is used to encode the transition relation of a TA by connecting states with tuples of states in a particular way, depending on the direction of the encoding.

We fix a tree automaton $\mathcal{A} = (Q, \Sigma, \Delta, R)$ for the rest of the section. We consider both top-down and bottom-up representations of its transition relation Δ, because some operations on $\mathcal{A}$ are easier to do on the former representation while others are easier on the latter. Moreover, we also provide an algorithm for translation between the considered representations. We assume w.l.o.g. that the input alphabet Σ of $\mathcal{A}$ is represented in binary using n bits. Each bit in the binary encoding of Σ is assigned a Boolean variable from the set $\{x_1, \ldots, x_n\}$. We can then use shared MTBDDs with a set of roots R and a domain S for encoding various functions of the form $R \to (\Sigma \to S)$ that we shall need.

9.2.1. Bottom-up Representation

Our *bottom-up* representation of the transition relation Δ of $\mathcal{A}$ uses a shared MTBDD Δ^{bu} over Σ where the set of root nodes is $Q^{\#}$ and the domain of labels of sink nodes is 2^Q (see Figure 9.1a). The shared MTBDD Δ^{bu} then represents the following function $[\![\Delta^{bu}]\!]$:

$$\begin{aligned} &[\![\Delta^{bu}]\!] : Q^{\#} \to (\Sigma \to 2^Q), \\ &[\![\Delta^{bu}]\!] = \lambda\, (q_1, \ldots, q_p)\, a \,.\, \{q \mid a(q_1, \ldots, q_p) \to q\}. \end{aligned} \tag{9.3}$$

Algorithm 9.1: Inversion of a shared MTBDD

```
    Input: Shared MTBDD $f$ such that $[\![f]\!] : R \to (\mathbb{B}^n \to 2^S)$
    Output: Shared MTBDD $g$ such that $[\![g]\!] : S \to (\mathbb{B}^n \to 2^R)$ and
            $r \in [\![g]\!](s,a) \iff s \in [\![f]\!](r,a)$
1   $g := \langle \lambda\, r\, \overline{x} \,.\, \emptyset \rangle$ ;                       // $[\![g]\!] : S \to (\mathbb{B}^n \to 2^R)$
2   foreach $s \in S$ such that $\exists r \in R, \exists \overline{x} \in \mathbb{B}^n : s \in [\![f]\!](r, \overline{x})$ do
3       foreach $r \in R$ such that $f(r) \neq \langle \lambda\, \overline{x} \,.\, \emptyset \rangle$ do
4           $g(s) := Apply_2(f(r), g(s), (\lambda\, X\, Y \,.$ if $s \in X$ then $Y \cup \{r\}$ else $Y))$;
5   return $g$;
```

It is easy to observe that the shared MTBDD Δ^{bu} is a semi-symbolic representation of Δ, in particular $[\![\Delta^{bu}]\!]((q_1, \dots, q_p), a) = up_a((q_1, \dots, q_p))$.

9.2.2. Top-down Representation

Our *top-down* representation of the transition relation Δ of $\mathcal{A}$ uses a shared MTBDD Δ^{td} over Σ where the set of root nodes is Q, and the domain of labels of sink nodes is $2^{Q^\#}$ (see Figure 9.1b). The MTBDD Δ^{td} represents the following function $[\![\Delta^{td}]\!]$:

$$\begin{aligned} &[\![\Delta^{td}]\!] : Q \to (\Sigma \to 2^{Q^\#}), \\ &[\![\Delta^{td}]\!] = \lambda\, q\, a \,.\, \{(q_1, \dots, q_p) \mid q \to a(q_1, \dots, q_p)\}. \end{aligned} \tag{9.4}$$

Again, we can easily see that $[\![\Delta^{td}]\!](q, a) = down_a(q)$.

9.2.3. Conversion Between Bottom-up and Top-down Representations

Sometimes it is necessary to convert between the bottom-up and top-down representation of a TA, for instance, when computing downward simulations (as explained later in the text). The transformation can be done using the generic algorithm given in Algorithm 9.1. The algorithm converts a shared MTBDD f representing a function $[\![f]\!] : R \to (\mathbb{B}^n \to 2^S)$ over n Boolean variables to a shared MTBDD g that represents the function $[\![g]\!] : S \to (\mathbb{B}^n \to 2^R)$ such that $r \in [\![g]\!](s, a) \iff s \in [\![f]\!](r, a)$. The algorithm first initialises g to map all elements of S and all valuations of the Boolean variables to the empty set. Then, for each element of $s \in S$ and $r \in R$ and for each valuation of the Boolean variables, which are implicitly traversed by the $Apply_2$ function, if s is in the sink node of $f(r)$ for some valuation of the Boolean variables, r is added to the sink node of $g(s)$ for the same valuation of the Boolean variables.

9.3. Tree Automata Algorithms over Semi-Symbolic Encoding

In this section, we propose algorithms for removing unreachable states, computing the union, intersection, and (maximum) downward simulation, as well as algorithms for upward and downward inclusion checking on the considered representation of TAs.

9.3.1. Removing Unreachable States

As the performance of many operations on automata depends on the size of the automaton (in the sense of the size of the state set and the size of the transition table), it is often desirable to remove both bottom-up and top-down unreachable states. Indeed, such states are useless: bottom-up unreachable states cannot be used to generate a finite tree and although top-down unreachable states can generate a finite tree, this tree cannot be a subtree of any tree accepted by the automaton.

Removing both bottom-up unreachable states for the bottom-up representation and top-down unreachable states for the top-down representation can be easily done by a single traversal through the automaton. Nevertheless, sometimes, e.g. when checking language inclusion of automata, it is useful to also remove states unreachable in the opposite direction.

The procedure for removing top-down unreachable states from a TA $\mathcal{A} = (Q, \Sigma, \Delta, R)$ represented bottom-up generates a directed graph (Q, E) where E contains the edge (q, r) if there is a transition $q \to a(q_1, \ldots, q_n) \in \Delta$ such that $r = q_i$ for some $1 \leq i \leq n$ and $a \in \Sigma$. When the graph is created, the states that correspond to nodes that are backward unreachable from the nodes corresponding to root states of $\mathcal{A}$ are removed from the automaton in a simple traversal.

Removing bottom-up unreachable states for the top-down semi-symbolic representation of $\mathcal{A}$ is more complex. First, $\mathcal{A}$ is traversed in the top-down manner while creating a directed *And-Or* graph $(N_\forall, N_\exists, E)$ where $N_\forall = Q$ represents the *And* nodes of the graph and $N_\exists \subseteq Q^*$ represents the *Or* nodes. The set of edges E contains the edge $(q, (q_1, \ldots, q_n))$ if there exists the transition $q \to a(q_1, \ldots, q_n) \in \Delta$ for some $a \in \Sigma$, and the edge $((q_1, \ldots, q_n), q)$ if $q_i = q$ for some $1 \leq i \leq n$. The algorithm starts by marking the node labelled by $()$ (which is an *Or* node) and proceeds by marking the nodes of the graph using the following rules: an *Or* node n_o is marked if there exists a marked node n_a such that $(n_o, n_a) \in E$, and an *And* node n_a is marked if all nodes n_o such that $(n_a, n_o) \in E$ are marked. When no new nodes can be marked, the states of $\mathcal{A}$ are reduced to only those that correspond to the marked *And* nodes in the graph.

9.3.2. Union

An algorithm for computing the union of a pair of TAs represented bottom-up follows as Algorithm 9.2. The presented algorithm simply unites the sets of states Q_1 and Q_2, and the sets of root states R_1 and R_2. We slightly abuse the notation and use $\Delta_1^{bu} \cup \Delta_2^{bu}$ to denote the union $\langle [\![\Delta_1^{bu}]\!] \cup [\![\Delta_2^{bu}]\!] \rangle$ of the considered shared MTBDDs. In order to carry out the union operation on the leaf transitions of the automaton (denoted by $()$), a single *Apply* operation needs to be performed. The *Apply* operation is given the lambda expression $\lambda X\, Y \,.\, X \cup Y$ as the function to perform on the sink nodes of the MTBDD. Correspondingly, when the *Apply* operation is evaluated, X and Y are mapped to the sets of states that are the values of the corresponding sink nodes of the first and second argument of the *Apply* operation, producing new sink nodes with the value of $X \cup Y$.

Performing the union on TAs represented top-down is more straightforward: $\mathcal{A}_\cup = (Q_1 \cup Q_2, \Sigma, \Delta_1^{td} \cup \Delta_2^{td}, R_1 \cup R_2)$, provided that $Q_1 \cap Q_2 = \emptyset$.

Algorithm 9.2: Union of TAs represented bottom-up

```
  Input: $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1^{bu}, R_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \Delta_2^{bu}, R_2)$, $Q_1 \cap Q_2 = \emptyset$
  Output: $\mathcal{A}_\cup = (Q_\cup, \Sigma, \Delta_\cup^{bu}, R_\cup)$ s.t. $L(\mathcal{A}_\cup) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$
1 $\Delta_\cup^{bu} := \Delta_1^{bu} \cup \Delta_2^{bu}$;
2 $\Delta_\cup^{bu}(()) := \mathit{Apply_2}(\Delta_1^{bu}(()), \Delta_2^{bu}(()), (\lambda\, X\, Y \,.\, X \cup Y))$;
3 return $\mathcal{A} = (Q_1 \cup Q_2, \Sigma, \Delta_\cup^{bu}, R_1 \cup R_2)$;
```

9.3.3. Intersection

Algorithm 9.3 performs intersection of a pair of TAs $\mathcal{A}^a = (Q^a, \Sigma, \Delta^{bu,a}, R^a)$ and $\mathcal{A}^b = (Q^b, \Sigma, \Delta^{bu,b}, R^b)$ that use the bottom-up representation. It constructs the intersection of $\mathcal{A}^a$ and $\mathcal{A}^b$ by creating a product automaton $\mathcal{A}_\cap = (Q^a \times Q^b, \Sigma, \Delta_\cap^{bu}, R^a \times R^b)$ where

$$\begin{aligned}\Delta_\cap^{bu} = \Big\{ & f((q_1^a, q_1^b), \ldots, (q_n^a, q_n^b)) \to (q^a, q^b) \mid f(q_1^a, \ldots, q_n^b) \to q^a \in \Delta^{bu,a}, \\ & f(q_1^b, \ldots, q_n^b) \to q^b \in \Delta^{bu,b}, \forall 1 \le i \le n : (q_i^a, q_i^b) \text{ is reachable} \Big\},\end{aligned} \tag{9.5}$$

where a product state (q^a, q^b) is *reachable* if there exists a symbol $g \in \Sigma$, states $q_1^a, \ldots, q_n^a \in Q^a$, and states $q_1^b, \ldots, q_n^b \in Q^b$ such that $g(q_1^a, \ldots, q_n^a) \to q^a \in \Delta^{bu,a}$ and $g(q_1^b, \ldots, q_n^b) \to q^b \in \Delta^{bu,b}$, and, further, for all $1 \le i \le n$, the product state (q_i^a, q_i^b) is reachable (note that leaf states are trivially reachable).

The transitions in $\Delta_\cap^{bu}$ basically run the two automata in parallel such that $\mathcal{A}_\cap$ contains only bottom-up reachable states and transitions. The algorithm detects reachable states by starting from leaf transitions of $\mathcal{A}^a$ and $\mathcal{A}^b$, analysing all transitions over leaf symbols, and collecting reachable product states into the set *newStates*. Then, until *newStates* is empty, a pair (q^a, q^b) is removed from *newStates*. For every such pair, we compute the set of product states reachable from any pairs of tuples $((q_1^a, \ldots, q_n^a), (q_1^b, \ldots, q_n^b))$ where $q^a = q_i^a$ and $q^b = q_i^b$ for some i, and at all positions $j \neq i$, it holds that $(q_i^a, q_i^b) \in Q_\cap$. We add the product states of the computed set to *newStates* and continue with the next iteration of the loop.

9.3.4. Downward Simulation

We next give an algorithm for computing the maximum downward simulation relation on the states of the TA $\mathcal{A}$ whose transition relation is encoded using our semi-symbolic representation. The algorithm is inspired by the algorithm of Ilie *et al* [INY04] (which is based on the same ideas as the algorithm of Henzinger *et al* [HHK95], but is more convenient for us because it uses only a single *remove* set) proposed for computing simulations on finite (word) automata. For use in the algorithm, we extend the notion of downward simulation to tuples of states by defining $(q_1, \ldots, q_n) \preceq_D (r_1, \ldots, r_n)$ to hold iff $\forall 1 \le i \le n : q_i \preceq_D r_i$.

Our algorithm for computing downward simulations, shown as Algorithm 9.4, starts with a relation that grossly over-approximates of the maximum downward simulation. The relation is then pruned in a loop, removing pairs that do not satisfy the simulation

Algorithm 9.3: Intersection of TAs represented bottom-up

```
   Input: TAs $\mathcal{A}^a = (Q^a, \Sigma, \Delta^{bu,a}, R^a)$ and $\mathcal{A}^b = (Q^b, \Sigma, \Delta^{bu,b}, R^b)$
   Output: $\mathcal{A}_\cap = (Q_\cap, \Sigma, \Delta^{bu}_\cap, R_\cap)$ s.t. $L(\mathcal{A}_\cap) = L(\mathcal{A}^a) \cap L(\mathcal{A}^b)$
1  $Q_\cap := \emptyset; R_\cap := \emptyset; newStates := \emptyset;$
2  $\Delta^{bu}_\cap := \langle \lambda\,(q_1, \ldots, q_n)\,a\,.\,\emptyset \rangle;$
3  $\Delta^{bu}_\cap(()) := Apply_2(\Delta^{bu,a}(()), \Delta^{bu,b}(()), (\texttt{isect}\ newStates));$
4  while $\exists (q^a, q^b) \in newStates$ do
5      $newStates := newStates \setminus \{(q^a, q^b)\};$
6      if $(q^a, q^b) \in Q_\cap$ then continue;
7      $Q_\cap := Q_\cap \cup \{(q^a, q^b)\};$
8      if $q^a \in R^a \wedge q^b \in R^b$ then $R_\cap := R_\cap \cup \{(q^a, q^b)\}$ ;
9      foreach $(q^a_1, \ldots, q^a_n) \in Q^{a\#}$ such that $\exists 1 \leq i \leq n : q^a = q^a_i$ do
10         foreach $(q^b_1, \ldots, q^b_n) \in Q^{b\#}$ such that $q^b = q^b_i$ do
11             if $\forall 1 \leq i \leq n : (q^a_i, q^b_i) \in Q_\cap$ then
12                 $\Delta^{bu}_\cap((q^a_1, q^b_1), \ldots, (q^a_n, q^b_n)) := Apply_2(\Delta^{bu,a}((q^a_1, \ldots, q^a_n)),$
                       $\Delta^{bu,b}((q^b_1, \ldots, q^b_n)), (\texttt{isect}\ newStates));$
13 return $\mathcal{A}_\cap = (Q_\cap, \Sigma, \Delta^{bu}_\cap, R_\cap);$
```

Function isect(&newStates, up^a, up^b)

```
1  $productSet := up^a \times up^b;$
2  $newStates := newStates \cup productSet;$
3  return $productSet;$
```

condition, until it stabilises and the maximum downward simulation is obtained. The algorithm uses the following main data structures:

- For every $q \in Q$, the set $sim(q) \subseteq Q$ contains states that are considered to simulate q at the current step of the computation. Its value is gradually pruned during the computation. At the end, *sim* encodes the maximum downward simulation being computed.
- The set $remove \subseteq Q^\# \times Q^\#$ contains pairs $\big((q_1, \ldots, q_n), (r_1, \ldots, r_n)\big)$ of tuples of states added there when it is found out that for some i, it holds that $q_i \not\preceq_D r_i$. The pairs are removed from the set and processed in a fixpoint computation.
- Finally, *cnt* is a shared MTBDD encoding the function $[\![cnt]\!] : Q^\# \to (\Sigma \to (Q \to \mathbb{N}))$ that for each $(q_1, \ldots, q_n) \in Q^\#$, $a \in \Sigma$, and $r \in Q$ gives the value $h \in \mathbb{N}$ denoting that r can make a top-down transition to h distinct tuples $(r_1, \ldots, r_n)$ such that $(q_1, \ldots, q_n) \preceq_D (r_1, \ldots, r_n)$ in the current approximation of *sim*.

The algorithm works in two phases. We assume that we start with a TA whose transition relation is represented bottom-up. In the *initialisation* phase, the dual top-down representation of the transition relation is first computed (note that we can also

Algorithm 9.4: Computing downward simulation on a semi-symbolic TA

```
   Input: TA $\mathcal{A} = (Q, \Sigma, \Delta^{bu}, R)$
   Output: Maximum downward simulation $\preceq_D \subseteq Q^2$
1  $\Delta^{td}$ := invertMTBDD($\Delta^{bu}$);
2  $remove := \emptyset$;
3  $initCnt := \langle \lambda\ a\ .\ \emptyset \rangle$ ;                      // $[\![initCnt]\!] : \Sigma \to (Q \to \mathbb{N})$
4  foreach $q \in Q$ do                                  // initialisation loop
5      $sim(q) := \emptyset$;
6      $initCnt := Apply_2(\Delta^{td}(q), initCnt, (\lambda X\, Y . Y \cup \{(q, |X|)\}))$ ;
7      foreach $r \in Q$ do
8          $isSim := true$;
9          $Apply_2(\Delta^{td}(q), \Delta^{td}(r), (\lambda X\, Y$ . if $(X \neq \emptyset \wedge Y = \emptyset)$ then $isSim := false))$ ;
10         if $isSim$ then $sim(q) := sim(q) \cup \{r\}$ ;
11         else
12             foreach $(q_1, \dots, q_n) \in Q^{\#}, (r_1, \dots, r_n) \in Q^{\#}$ do
13                 if $\exists 1 \leq i \leq n : q_i = q \wedge r_i = r$ then
14                     $remove := remove \cup \{((q_1, \dots, q_n), (r_1, \dots, r_n))\}$;
15 $cnt := \langle \lambda\, (q_1, \dots, q_n)\, a\, .\, \emptyset \rangle$ ;          // $[\![cnt]\!] : Q^{\#} \to (\Sigma \to (Q \to \mathbb{N}))$
16 foreach $(q_1, \dots, q_n) \in Q^{\#}$ do $cnt((q_1, \dots, q_n)) := initCnt$ ;
17 while $\exists((q_1, \dots, q_n), (r_1, \dots, r_n)) \in remove$ do          // fixpoint comp.
18     $remove := remove \setminus \{((q_1, \dots, q_n), (r_1, \dots, r_n))\}$;
19     $cnt((q_1, \dots, q_n)) := Apply_3(\Delta^{bu}((r_1, \dots, r_n)), \Delta^{bu}((q_1, \dots, q_n)),$
           $cnt((q_1, \dots, q_n))$, (refine $sim$ $remove$));
20 return $\{(q, r) \mid q \in Q, r \in sim(q)\}$;
```

start with a top-down representation and compute the bottom-up representation since both are needed in the algorithm). The three main data structures are then initialised as follows:

- For each $q \in Q$, the set $sim(q)$ is initialised as the set of states that can make top-down transitions over the same symbols as q, which is determined using the *Apply* operation on line 9. That is, when starting the main computation loop on line 17, the value of *sim* for each $q \in Q$ is $sim(q) = \{r \mid \forall a \in \Sigma : q \to a(q_1, \dots, q_n) \implies r \to a(r_1, \dots, r_n)$ for some $q_1, \dots, q_n, r_1, \dots, r_n \in Q\}$.

- The *remove* set is initialised to contain each pair of tuples of states $\big((q_1, \dots, q_n), (r_1, \dots, r_n)\big)$ such that it holds that the relation $(q_1, \dots, q_n) \preceq_D (r_1, \dots, r_n)$ is broken even for the initial approximation of $\preceq_D$, i.e. for some position $1 \leq i \leq n$, there is a pair (q_i, r_i) such that $r_i \notin sim(q_i)$.

- To initialise the shared MTBDD *cnt*, the algorithm first constructs an auxiliary MTBDD *initCnt* representing a function $[\![initCnt]\!] : \Sigma \to (Q \to \mathbb{N})$. Via the *Apply*

```
Function refine(&sim, &remove, $up_aR$, $up_aQ$, $cnt_aQ$)
1  $newCnt_aQ := cnt_aQ$;
2  foreach $s \in up_aR$ do
3      $newCnt_aQ(s) := newCnt_aQ(s) - 1$;
4      if $newCnt_aQ(s) = 0$ then
5          foreach $p \in up_aQ : s \in sim(p)$ do
6              foreach $(p_1, \dots, p_n) \in Q^{\#}, (s_1, \dots, s_n) \in Q^{\#}$ do
7                  if $\exists 1 \leq i \leq n : p_i = p \wedge s_i = s$ then
8                      if $\forall 1 \leq j \leq n : s_j \in sim(p_j)$ then
9                          $remove := remove \cup \{((p_1, \dots, p_n), (s_1, \dots, s_n))\}$;
10             $sim(p) := sim(p) \setminus \{s\}$;
11 return $newCnt_aQ$;
```

operation on line 6, this MTBDD gradually collects for every symbol $a \in \Sigma$ the set of pairs (q, h) such that q can make a top-down transition over the symbol a to h distinct tuples. This MTBDD is then copied to the shared MTBDD *cnt* for each tuple of states $(q_1, \dots, q_n) \in Q^{\#}$. This is justified by the fact that we start by assuming that the simulation relation is equal to $Q \times Q$, which, for a symbol $a \in \Sigma$ and a pair $(q, h) \in cnt((q_1, \dots, q_n))$, means that $(q_1, \dots, q_n)$ can make a bottom-up transition over a to h distinct states $r \in sim(q)$.

After that, the main *computation* phase proceeds by gradually restricting the initial over-approximation of the maximum downward simulation being computed. As we have said, the *remove* set contains pairs $((q_1, \dots, q_n), (r_1, \dots, r_n))$ for which it holds that $(q_1, \dots, q_n)$ cannot be simulated by $(r_1, \dots, r_n)$, i.e. $(q_1, \dots, q_n) \not\preceq_D (r_1, \dots, r_n)$. When such a pair is processed, the algorithm proceeds by decrementing the counter $[\![cnt]\!]((q_1, \dots, q_n), a, s)$ for each state s for which there exists a bottom-up transition over a symbol $a \in \Sigma$ such that $a(r_1, \dots, r_n) \to s$. The meaning is that s can make one less top-down transition over a to some $(t_1, \dots, t_n)$ such that $(q_1, \dots, q_n) \preceq_D (t_1, \dots, t_n)$. If $[\![cnt]\!]((q_1, \dots, q_n), a, s)$ drops to zero, it means that s cannot make a top-down transition over a to any such $(t_1, \dots, t_n)$. This means, for all $p \in Q$ such that p can make a top-down transition over a to $(q_1, \dots, q_n)$, that s no longer simulates p, i.e. $p \not\preceq_D s$. When the simulation relation between p and s breaks, the simulation relation between all m-tuples $(p_1, \dots, p_m)$ and $(s_1, \dots, s_m)$ such that $\exists 1 \leq j \leq m : p_j = p \wedge s_j = s$ must also be broken. As a result, the pair $((p_1, \dots, p_m), (s_1, \dots, s_m))$ is put to the *remove* set (unless the simulation relation between some other states in the tuples has already been broken before).

Correctness of the algorithm is summarised in the below theorem, which can be proven analogically as correctness of the algorithm proposed in [INY04], taking into account the meaning of the above described MTBDD-based structures and the operations performed on them.

Function expandSymb(p_S, P_B, *workset*)

```
   // $p_S \in Q_S$, $P_B \subseteq Q_B$, and $workset \subseteq Q_S \times 2^{Q_B}$
 1 if $\exists (p'_S, P'_B) \in workset : p_S \preceq p'_S \wedge P'_B \preceq^{\forall\exists} P_B$ then return $true$ ;
 2 if $\exists (p'_S, P'_B) \in NN : p'_S \preceq p_S \wedge P_B \preceq^{\forall\exists} P'_B$ then return $false$ ;
 3 if $\exists p \in P_B : p_S \preceq p$ then return $true$ ;
 4 $workset := workset \cup \{(p_S, P_B)\}$;
 5 $tmp := \langle \lambda\ a\ .\ \emptyset \rangle$ ;                          // $[\![tmp]\!] : \Sigma \to 2^{Q_B^\#}$
 6 foreach $p_B \in P_B$ do
 7     $tmp := Apply_2(tmp, \Delta_B^{td}(p_B), (\lambda X\, Y\, .\, X \cup Y))$;
 8 $doesInclHold := true$;
 9 $Apply_2(\Delta_S^{td}(p_S), tmp,$ (procDown $doesInclHold$ $workset$));
10 return $doesInclHold$;
```

Theorem 9.1. *When applied on a TA $\mathcal{A} = (Q, \Sigma, \Delta, R)$ whose transition relation is encoded semi-symbolically in the bottom-up way as Δ^{bu}, Algorithm 9.4 terminates and returns the maximum downward simulation on Q.*

9.3.5. Downward Inclusion Checking

We now proceed to an algorithm for efficient downward inclusion checking on top-down semi-symbolically represented TAs. The algorithm we propose for this purpose is derived from Algorithm 8.2 by plugging the `expandSymb` function instead of the `expand2` function. It is based on the same basic principle as `expand2`, but it has to cope with the symbolically encoded transition relation. In particular, in order to inspect whether, for a pair (p_S, P_B) and all symbols $a \in \Sigma$, the inclusion between each tuple from $down_a(p_S)$ and the set of tuples $down_a(P_B)$ holds, the *doesInclHold* parameter, initialised to *true*, is passed to the *Apply* operation on line 9 of the `expandSymb` function. If the algorithm finds out that the inclusion does not hold in some execution of the `procDown` function in the context of a single *Apply*, *doesInclHold* is assigned the *false* value, which is later returned by `expandSymb`; otherwise, `expandSymb` returns the original value *true*. Note that the optimisations of `expand2` presented in Section 8.2.2 (function `expand2e`) can be easily adopted also to function `expandSymb`.

9.3.6. Upward Inclusion Checking

We next present an algorithm for upward inclusion checking on semi-symbolically encoded TAs. We present a version that is not combined with a use of simulation since the experiments that we have done with explicitly represented automata in Chapter 8 were not very favourable for upward inclusion checking combined with a use of simulation. We note, however, that for the future, providing such an algorithm and testing it on a broader set of experiments is still useful.

Function procDown(&doesInclHold, &$workset$, $down_a p_S$, $down_a P_B$)

```
if $() \in down_a p_S \wedge () \notin down_a P_B$ then $doesInclHold := false$ ;
else
    $W := down_a P_B$;
    foreach $(r_1, \ldots, r_n) \in down_a p_S$ do                          // $p_S \to a(r_1, \ldots, r_n)$
        foreach $f \in \{W \to \{1, \ldots, n\}\}$ do                        // $\forall f \in cf(P_B, a)$
            $found := false$;
            foreach $1 \le i \le n$ do
                $S := \{q_i \mid (q_1, \ldots, q_n) \in W, f((q_1, \ldots, q_n)) = i\}$;
                if expandSymb($r_i, S, workset$) then                          // if $L(r_i) \subseteq L(S)$
                    $found := true$;
                    break;
                if $\nexists (r', H) \in NN : r' \preceq r_i \wedge S \preceq^{\forall\exists} H$ then
                    $NN := (NN \setminus \{(r', H) \mid H \preceq^{\forall\exists} S, r_i \preceq r'\}) \cup \{(r_i, S)\}$;
            if $\neg found$ then
                $doesInclHold := false$;
                return;
```

Our upward inclusion checking algorithm is based on the algorithm of Bouajjani *et al* [BHH+08]. The intuition behind this algorithm is that when checking inclusion of languages of two automata $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S^{bu}, R_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B^{bu}, R_B)$, the algorithm works with a set $antichain \subseteq Q_S \times 2^{Q_B}$ such that $(q, D) \in antichain$ if q accepts some tree in $\mathcal{A}_S$, and D is the set of all states in $\mathcal{A}_B$ that accept the same tree. If it holds that $q \in R_S$ and $D \cap R_B = \emptyset$, then $\mathcal{A}_S$ can accept a tree that A_B cannot accept, and therefore the inclusion $L(\mathcal{A}_S) \subseteq L(\mathcal{A}_B)$ does not hold. Also, when the algorithm reaches a pair (q, E) such that $D \subset E$ for some $(q, D) \in antichain$, the pair (q, E) is dropped and not added into *antichain*. This is justified by the fact that if a counterexample to inclusion can be shown from (q, E), it can be found from (q, D) too (since the possible moves of $\mathcal{A}_B$ from D are even more limited than from E). Furthermore, when a pair (q, F) is reached such that $F \subset D$ for some $(q, D) \in antichain$, then all pairs (q, D) with $F \subset D$ are removed from *antichain* and (q, F) is added in their place. Hence, the set *antichain* is indeed an antichain in the poset $(Q_S, \mathrm{id}_{Q_S}) \times (2^{Q_B}, \subseteq)$, i.e. for a given state $q_S \in Q_S$ there are no two sets $G, H \in 2^{Q_B}$ in *antichain* such that $G \subset H$.

Our algorithm for upward inclusion checking is shown as Algorithm 9.5. In the algorithm, the *Apply* operation on line 3 first collects into the sets *antichain* and *notProcessed* the pairs $(q, D) \in Q_S \times 2^{Q_B}$ consisting of states accessible through equilabelled leaf transitions in $\mathcal{A}_S$ and $\mathcal{A}_B$. Then, until the *notProcessed* set is empty or a counterexample to inclusion is found, the algorithm removes a pair (q, D) from the set *notProcessed* and processes it. The processing consists of finding a tuple $(q_1, \ldots, q_n) \in Q_S^{\#}$ containing q as some q_i such that all other states of the tuple also appear in the first position of some pair in *antichain*, and also finding all tuples $(s_1, \ldots, s_n) \in Q_B^n$ such that $s_i \in D$

Algorithm 9.5: Upward antichain-based inclusion for semi-symbolic TAs

```
Input: TAs $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S^{bu}, R_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B^{bu}, R_B)$
Output: $true$ if $L(\mathcal{A}_S) \subseteq L(\mathcal{A}_B)$, $false$ otherwise
1  $notProcessed := \emptyset$;
2  $antichain := \emptyset$;
3  $Apply_2(\Delta_S^{bu}(()), \Delta_B^{bu}(()), ($collectProducts $antichain\ notProcessed))$ ;
4  while $\exists(q, D) \in notProcessed$ do
5      $notProcessed := notProcessed \setminus \{(q, D)\}$;
6      if $q \in R_S \land D \cap R_B = \emptyset$ then return $false$ ;
7      foreach $(q_1, \ldots, q_n) \in Q_S^{\#}$ such that $\exists 1 \leq i \leq n : q_i = q$ do
8          if $\forall 1 \leq j \leq n : \exists R_j \subseteq Q_B : (q_j, R_j) \in antichain$ then
9              $tmp := \langle \lambda\ a\ .\ \emptyset \rangle$;
10             foreach $(s_1, \ldots, s_n) \in Q_B^n$ such that $s_i \in D$ do
11                 if $\forall 1 \leq j \leq n : s_j \in R_j$ then
12                     $tmp := Apply_2(tmp, \Delta_B^{bu}((s_1, \ldots, s_n)), (\lambda\ X\ Y\ .\ X \cup Y))$;
13             $Apply_2(\Delta_S^{bu}((q_1, \ldots, q_n)), tmp,$
                   $($collectProducts $antichain\ notProcessed))$;
14 return $true$;
```

Function collectProducts(&antichain, &notProcessed, up_S, up_B)

```
1  foreach $q \in up_S$ do
2      if $\nexists(q, E) \in antichain$ such that $E \subseteq up_B$ then
3          $antichain := (antichain \setminus \{(q, F) \mid up_B \subset F\}) \cup \{(q, up_B)\}$;
4          $notProcessed := (notProcessed \setminus \{(q, F) \mid up_B \subset F\}) \cup \{(q, up_B)\}$;
```

and all states s_j, for $i \neq j$, appear in the second position of some pair from *antichain*. The transition relations of the said tuples are united by the *Apply* operation on line 12. (Note that it is possible to optimise the computation of the set of tuples by a technique similar to the one proposed in Section 10.2.3 for explicitly represented TAs.) The *Apply* operation on line 13 then collects the reachable pairs, and the loop continues.

9.4. An MTBDD Library

Efficient algorithms over a symbolic representation of the transition relation of a TA require an efficient implementation of the underlying MTBDD library. Our first implementation of algorithms for handling semi-symbolically represented tree automata used a customisation of the CUDD library [Som11] for manipulating MTBDDs. The experiments in [HLŠV11] and profiling of the code showed that the overhead of the customised library is too large. Moreover, the customisation of CUDD did not provide an easy and transparent way of manipulating MTBDDs. These two facts hinted that the implementation of the algorithms would greatly benefit from a major redesign of the MTBDD back-end. Therefore, we created our own generic implementation of MTBDDs with a clean and simple-to-use interface.

The new MTBDD library uses *shared* MTBDDs for every domain. In order to prevent memory leaks, each node of the MTBDD contains a reference counter of other nodes or variables pointing to it. In case the counter reaches zero, the node is deleted from the memory. Because of these implementation choices, copying an MTBDD can be easily done by simply copying the pointer to the root node of the copied MTBDD and incrementing its reference counter.

There are two types of nodes of the MTBDD: *internal nodes* and *leaf nodes.* A leaf node contains a value from the domain of the MTBDD, while an internal node contains a variable name and pointers to the *low* and *high* children of the node. In addition, nodes of both types also contain the aforementioned reference counter. The nodes are manipulated using pointers to them only, and the distinction between a leaf node and an internal node is done according to the least significant bit of the pointer (the compiler aligns these data structures to addresses which are multiples of 4, this bit can therefore be neglected and simply masked out when accessing the value of a node pointer).

For our use, we implemented unary, binary, and ternary *Apply* operations. Further, we also provide *VoidApply* operations, which are Apply operations that do not build a new MTBDD but have a side-effect only. For operations that do not need to build new MTBDDs but rather e.g. only collect data from the leaf nodes, using *VoidApply* saves a considerable and unnecessary overhead. During the execution of an *Apply* operation, both internal and leaf nodes are cached in hash tables.

The newly implemented MTBDD library does not support MTBDD reordering so far, yet the library performs better when compared to our first implementation of a semi-symbolic encoding that used customised CUDD (the speed-up was over 300 times for upward inclusion checking and over 3 000 times for downward inclusion checking). Note that for some applications, e.g. the decision procedure for the WS1S logic presented in Chapter 7, reordering is not really necessary, because a good variable ordering can be chosen in advance, e.g. in the particular decision procedure, it can follow the order of quantifiers in the prefix of the decided formula.

9.5. Experimental Results

We implemented and evaluated the algorithms proposed in this chapter inside the framework of the VATA library, which is presented in more detail in Chapter 10. We focused on an evaluation of the various language inclusion checking algorithms presented in Section 9.3. The row `down` gives results for our implementation of the downward algorithm, Algorithm 8.2, with the function `expandSymb` from Section 9.3.5 plugged in and the identity relation used as the input preorder. The row `down+s` gives results for the same algorithm with the only exception that the maximum downward simulation is used instead of the identity as the input preorder. The rows `down-opt` and `down-opt+s` are based on Algorithm 8.2, the optimised version of the algorithm. Results of the upward

Table 9.1.: Results of the experiments (timeout 30 s)

Algorithm	All pairs		$L(\mathcal{A}) \not\subseteq L(\mathcal{B})$		$L(\mathcal{A}) \subseteq L(\mathcal{B})$	
	Winner	Timeouts	Winner	Timeouts	Winner	Timeouts
`down`	44.02 %	5.87 %	45.03 %	2.48 %	19.74 %	72.37 %
`down+s`	0.00 %	77.93 %	0.00 %	80.03 %	0.00 %	36.84 %
`down-opt`	31.73 %	5.87 %	33.06 %	2.48 %	0.00 %	72.37 %
`down-opt+s`	0.00 %	78.00 %	0.00 %	80.09 %	0.00 %	36.84 %
`up`	24.25 %	22.26 %	21.91 %	23.39 %	80.26 %	0.00 %

inclusion checking algorithm (Algorithm 9.5) are in the table represented in the row labelled with `up`. For the algorithms that use simulation, the simulation is computed using Algorithm 9.4 and the time of computation of the simulation relation is included in the running time of the algorithm.

The table compares the methods according to the percentage of the cases in which they were the fastest when checking inclusion on the same automata pair, and also according to the percentage of timeouts (the timeout was set to 30 s). The results for runs of the inclusion checking algorithms on almost 2 000 pairs of TAs are in the column labelled with "All pairs". We also checked the performance of the algorithms for cases when inclusion either *does* or *does not* hold in order to explore the ability of the algorithms to either find a counterexample in the case when inclusion does not hold, or prove the inclusion in case it does. The results below "$L(\mathcal{A}) \not\subseteq L(\mathcal{B})$" in the table are for the pairs $\mathcal{A}, \mathcal{B}$ of the test set where the inclusion does not hold, and the column under "$L(\mathcal{A}) \subseteq L(\mathcal{B})$" reports on the cases where the inclusion holds.

The output of the experiments shows (again, cf. the results in Chapter 8) the domination of the downward approach. It can be, however, noted that the downward simulation did not help much (in rows `down+s` and `down-opt+s`). This was caused by the overhead of the computation of the simulation relation. Our symbolic downward simulation algorithm is still immature when compared to the one used for the explicit encoding. Despite this, we can observe that for the cases inclusion holds, the use of simulation can significantly decrease the number of timeouts.

9.6. Conclusion

This chapter presented a semi-symbolic MTBDD-based representation of nondeterministic TAs generalising the one used by MONA, together with important tree automata algorithms working over this representation, most notably an algorithm for computing downward simulations over TAs inspired by [INY04] and the downward language inclusion algorithm improved by simulations and antichains proposed in Chapter 8. We have experimentally justified usefulness of the symbolic encoding for nondeterministic TAs with large alphabets.

In the future, we wish to advance the algorithms presented in this chapter even further. A particular candidate would be the algorithm for computing the downward simulation relation on a semi-symbolically encoded automaton, which is still quite immature when compared with current state-of-the-art algorithms for explicitly represented automata [RT07, HŠ09]. Moreover, we plan to also explore other symbolic representations of finite automata on both words and trees, and advanced algorithms on these representations. We are currently working on an algorithm for computing forward simulation on fully-symbolically represented finite automata. The representation stores the entire transition function of the automaton in a single BDD. Using this representation, we wish to avoid the issue with storing the counters from the algorithm presented in Section 9.3.4 inside an MTBDD, which is one of the main bottlenecks of the algorithm, due to the counters being frequently accessed.

10. An Efficient Tree Automata Library

The techniques presented in Chapters 3–7, as well as many other formal verification techniques, rely on an efficient underlying implementation of tree automata, and their success can be hindered by a poor performance of a naïve TA implementation. Some of these techniques are: abstract regular tree model checking (ARTMC) [AJMd02, BHRV12] applied e.g. for verification of programs with complex dynamic data structures [BHRV06], verification of programs handling dynamically linked structures with data [MPQ11], or decision procedures for separation logic [IRV14].

Currently, there exist several available tree automata libraries; they are, however, mostly written in OCaml (e.g. Timbuk/Taml [Gen03]) or Java (e.g. LETHAL [CJH$^+$09]) and they do not always use the most advanced algorithms known to date. Therefore, they are not suitable for tasks that require the available processing power be utilised as efficiently as possible. An exception from these libraries is MONA [KMS02] implementing decision procedures over WS1S/WS2S. MONA contains a highly optimised TA package written in C, but, alas, it supports only binary deterministic tree automata. At the same time, it turns out that determinisation is often a very significant bottleneck of using TAs, and a lot of effort has therefore been invested into developing efficient algorithms for handling nondeterministic tree automata without a need to ever determinise them (e.g. the techniques presented in Chapters 8 and 9).

In order to allow researchers focus on developing verification techniques rather than reimplementing and optimising a TA package, we provide VATA[1], an easy-to-use open source library for efficient manipulation of nondeterministic TAs. VATA supports many of the operations commonly used in automata-based formal verification techniques over two complementary encodings: explicit and semi-symbolic. The *explicit* encoding is suitable for most applications that do not need to use alphabets with a large number of symbols. On the other hand, the semi-symbolic encoding (described in more detail in Chapter 9) is suitable for applications that make use of such alphabets, e.g. the ARTMC-based verification of programs with complex dynamic data structures [BHRV12] or decision procedures of the MSO or WSkS logics [KMS02] (cf. the procedure in Chapter 7).

At the present time, the main application of the structures and algorithms implemented in VATA for handling explicitly encoded TAs are the Forester tool for verification of programs with complex dynamic data structures (see Chapters 3–5) and the tools implementing TA-based decision procedures for separation logic: SPEN (see Chapter 6) and SLIDE [IRV14]. The semi-symbolic encoding of TAs has been used in our decision procedure for WS1S in Chapter 7 (where we use unary tree automata in the place of finite automata).

[1] `http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/`

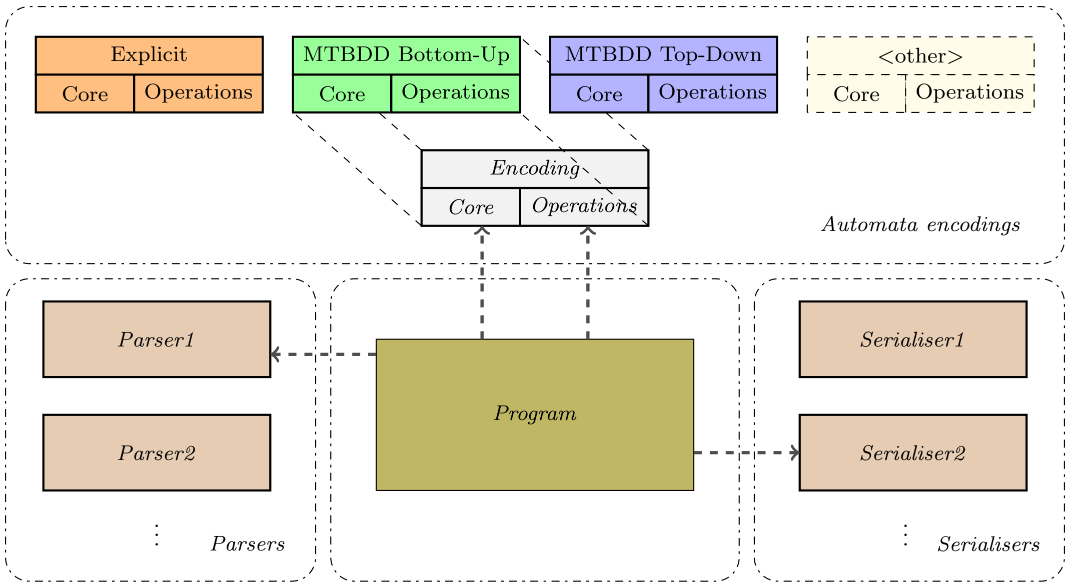


Figure 10.1.: The architecture of the VATA library

In this chapter, we give an overview of the algorithms available and mention various interesting optimisations that we used when implementing them. Based on experimental evidence, we argue that these optimisations are important for the performance of the library.

Outline. The structure of this chapter is the following. In Section 10.1, we describe the design of VATA. Section 10.2 gives a description of the operations that we support. In Section 10.3, we report on our experiments with the implementation. Section 10.4 briefly concludes the chapter.

10.1. Design of the Library

The library is designed in a modular way (see Figure 10.1). The user can choose a module encapsulating his preferred automata encoding and its corresponding operations. Various encodings share the same general interface so it is easy to swap one encoding for another, unless the user takes advantage of encoding-specific functions or operations.

Thanks to the modular design of the library, it is easy to provide an own encoding of tree (or word) automata and effectively exploit the remaining parts of the infrastructure, such as parsers and serialisers from/to different formats, the unit testing framework, performance tests, etc.

The VATA library is implemented in C++ using the C++11 standard library. In order to avoid expensive look-ups of entry points of virtual methods in the *virtual-method table* of an object and to fully exploit compiler's capabilities of code inlining and optimisation of code according to static analysis, the library heavily exploits polymorphism using C++ function templates instead of using virtual methods for core functions. We believe that this is one of the reasons why the performance of the optimised code (the `-O3` flag of `gcc`) is up to 10 times better than the performance of the non-optimised code (the `-O0` flag of `gcc`).

10.1.1. Explicit Encoding

In the explicit representation of TAs used in Vata, transitions are stored in the top-down manner inside a hierarchical data structure similar to a hash table. More precisely, the *top-level lookup table* maps states to *transition clusters.* Each such cluster is itself a lookup table that maps alphabet symbols to a *set of pointers to tuples* of states. The set of pointers to tuples of states is represented using a red-black tree. The tuples of states are stored in a designated hash table to further reduce the required amount of space (by not storing the same tuples of states multiple times). An example of the encoding is depicted in Figure 10.2.

In order to insert a transition $q \to a(q_1, \ldots, q_n)$ into the transition table, one proceeds using the following algorithm:

1. Find a transition cluster that corresponds to the state q in the top-level lookup table. If such a cluster does not exist, create one.

2. In the given cluster, find a set of pointers to tuples of states reachable from q over a. If the set does not exist, create one.

3. Obtain the pointer to the tuple $(q_1, \ldots, q_n)$ from the tuple lookup table and insert it into the found or created set of pointers.

If one ignores the worst case time complexity of the underlying data structures (which, according to our experience, has usually a negligible real impact only), then inserting a single transition into the transition table requires a constant number of steps. Yet the representation provides a more efficient encoding than a plain list of transitions because some transitions share the space required to store the parent states (e.g. state q in the transition $q \to a(q_1, \ldots, q_n)$). Moreover, some transitions also share the alphabet symbol and each tuple of states appearing in the set of transitions is stored only once. Additionally, the encoding allows us to easily perform certain critical operations, such as finding a set of transitions $q \to a(q_1, \ldots, q_n)$ for a given state q. This is useful e.g. during the elimination of (top-down) unreachable states or for the downward inclusion checking algorithm.

In some situations, one needs to manipulate many tree automata at the same time. To give an example, in the forest automata-based program analysis framework considered in Chapters 3 and 5, where (in theory) one needs to store one automaton representing the content of a heap for each reachable state of the program. To improve the performance of our library in such scenarios, we adapt the *copy-on-write* principle. Every time one needs to create a copy of an automaton A to be subsequently modified, it is enough to create a new automaton A' that obtains a pointer to the transition table of A (which

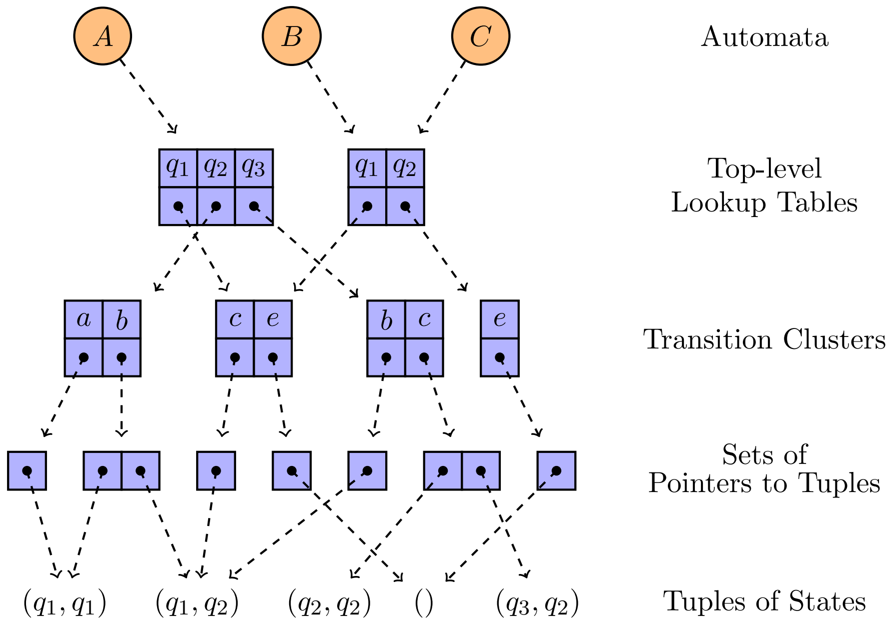


Figure 10.2.: An example of the Vata's explicit encoding of transition functions of three automata A, B, C. In particular, one can see that A contains a transition $q_1 \to c(q_1, q_2)$: it suffices to follow the corresponding arrows. Moreover, B also contains the same transition (and the corresponding part of the transition table is shared with A). Finally, C shares its transitions with B.

requires constant time). Subsequently, as more transitions are inserted into A' (or A), only the part of the shared transition table that gets modified is copied (Figure 10.2 also provides an illustration of this feature).

10.2. Supported Operations

Vata allows the user to choose one of three available encodings: explicit, semi-symbolic top-down, and semi-symbolic bottom-up. Depending on the choice, certain TA operations may or may not be available. Here we describe only operations for the explicit encoding; the description of the operations for the two semi-symbolic encodings is provided in Chapter 9. The supported operations for the explicit representation are the following: union, intersection, elimination of (bottom-up, top-down) unreachable states, inclusion checking (both upward and downward), computation of the maximum simulation relations (both upward and downward simulations), and language-preserving size reduction based on the simulation equivalence. In the case of testing language inclusion, we provide several implementations of the operation, because, as observed in the experimental section of Chapter 8, the performance of different approaches varies on different use cases.

Below, we do not discuss the relatively straightforward implementation of the basic operations on TAs and we comment on the more advanced operations only, in particular on computing the (maximum) simulation relations and upward testing of language inclusion (the used algorithms for downward inclusion testing are described in Chapter 8.

10.2.1. Downward and Upward Simulation

Computation of the (maximum) downward and upward simulation relations for the explicit representation of the TAs is in Vata performed in three steps. For the downward simulation, the input TA is first translated into a labelled transition system (LTS) using the technique described in [ABH+08]. In the second step, the simulation relation for the LTS is computed using an implementation of the state-of-the-art algorithms for computing simulations on LTS's [RT07, HŠ09], with some further optimisations mentioned in Section 10.2.4. Finally, the result is projected back to the set of states of the original automaton.

For the upward simulation, the steps are the same, with the exception of the translation of a TA into an LTS, which is in this case performed using the algorithm from [ABH+08].

10.2.2. Simulation-based Size Reduction

In a typical setting, one often wants to use a representation of tree automata that is as small as possible in order to reduce the memory consumption and speed up operations on the automata (especially the potentially costly ones, such as inclusion testing). To achieve that, the classical approach is to use determinisation and minimisation. However, the minimal deterministic tree automata can still be much bigger than the original nondeterministic ones. Therefore, Vata offers a possibility to reduce the size of tree automata without determinisation by their quotienting w.r.t. an equivalence relation—currently, only the downward simulation equivalence is supported.

The procedure works as follows: first, the downward simulation relation $\preceq_D$ is computed for the automaton. Then, the symmetric fragment of $\preceq_D$ (which is an equivalence relation) is extracted, and each state appearing within the transition function is replaced by a representative of the corresponding equivalence class. A further reduction is then based on the following observation: if an automaton contains a transition $q \to a(q_1, \ldots, q_n)$, any additional transition $q \to a(r_1, \ldots, r_n)$ can be omitted if $r_i \preceq_D q_i$ for all $1 \leq i \leq n$: such a transition does not contribute to the language of the result (recall that, for the downward simulation preorder $\preceq_D$, it holds that $q \preceq_D r \implies L(q) \subseteq L(r)$).

10.2.3. Upward Inclusion

The algorithm for upward inclusion testing using the explicit encodings of TAs of Vata (which was used in the experiments of Chapter 8) is based on, as its name suggests, upward traverse through the TAs. Our top-down representation of the transition relation is therefore not very suitable here. We can, however, afford to build a temporary bottom-up encoding, since the overhead of a translation into this encoding is negligible compared to the complexity of the subsequent operations.

The upward algorithm for language inclusion testing is based on the approach introduced in [BHH+08]. Here, the main principle used for checking whether $L(\mathcal{A}) \subseteq L(\mathcal{B})$ is to search for a tree that is accepted by $\mathcal{A}$ and not by $\mathcal{B}$ (thus being a witness for $L(\mathcal{A}) \nsubseteq L(\mathcal{B})$). This is done by simultaneously traversing both $\mathcal{A}$ and $\mathcal{B}$ from their leaf transitions while generating pairs $(p_\mathcal{A}, P_\mathcal{B}) \in Q_\mathcal{A} \times 2^{Q_\mathcal{B}}$ where $p_\mathcal{A}$ represents a state into which $\mathcal{A}$ can get on some input tree and $P_\mathcal{B}$ is the set of *all* states into which $\mathcal{B}$ can get over the same tree. The inclusion then does clearly not hold iff it is possible to generate a pair consisting of an accepting state of $\mathcal{A}$ and of exclusively non-accepting states of $\mathcal{B}$.

The algorithm collects the so far generated pairs $(p_\mathcal{A}, P_\mathcal{B})$ in a set called *Visited*. Another set called *Next* is used to store the generated pairs whose successors are still to be explored. One can then observe that whenever one can reach a counterexample to inclusion from $(p_\mathcal{A}, P_\mathcal{B})$, one can also reach a counterexample from any $(p_\mathcal{A}, P'_\mathcal{B} \subseteq P_\mathcal{B})$ as $P'_\mathcal{B}$ allows even less runs of $\mathcal{B}$ than $P_\mathcal{B}$. Using this observation, both mentioned sets can be represented using antichains. In particular, one does not need to store and further explore any two elements comparable w.r.t. $(=, \subseteq)$, i.e. by equality on the first component and inclusion on the other component.

Clearly, the running time of the above algorithm strongly depends on the total number of pairs $(p_\mathcal{A}, P_\mathcal{B})$ taken from *Next* for further processing. Indeed, this is one of the reasons why the antichain-based optimisations helps. According to our experience, the number of pairs that need to be processed can further be reduced when processing the pairs stored in *Next* in a suitable order. Our experimental results have shown that we can achieve a very good improvement by preferring those pairs $(p_\mathcal{A}, P_\mathcal{B})$ that have a smaller (w.r.t. the size of the set) second component.

Yet another way that we found useful when improving the above algorithm is to optimise the way the algorithm computes the successors of a pair from *Next*. The original algorithm picks a pair $(p_\mathcal{A}, P_\mathcal{B})$ from *Next* and puts it into *Visited*. Then, it finds all transitions of the form $a(p_{\mathcal{A},1}, \ldots, p_{\mathcal{A},n}) \to p$ in $\mathcal{A}$ such that $(p_{\mathcal{A},i}, P_{\mathcal{B},i}) \in$ *Visited* for all $1 \leq i \leq n$ and $(p_{\mathcal{A},j}, P_{\mathcal{B},j}) = (p_\mathcal{A}, P_\mathcal{B})$ for some $1 \leq j \leq n$. For each such transition, it finds all transitions of the form $a(q_1, \ldots, q_n) \to q$ in $\mathcal{B}$ such that $q_i \in P_{\mathcal{B},i}$ for all $1 \leq i \leq n$. Here, the process of finding the needed $\mathcal{B}$ transitions is especially costly. In order to speed it up, we cache for each alphabet symbol a, each position i, and each set $P_{\mathcal{B},i}$, the set of transitions $\{a(q_1, \ldots, q_n) \to q \in \Delta_\mathcal{B} : q_i \in P_{\mathcal{B},i}\}$ at the first time it is used in the computation of successors. Then, whenever we need to find all transitions of the form $a(q_1, \ldots, q_n) \to q$ in $\mathcal{B}$ such that $q_i \in P_{\mathcal{B},i}$ for all $1 \leq i \leq n$, we find them simply by intersecting the sets of transitions cached for each $(P_{\mathcal{B},i}, i, a)$.

Next, we propose another modification of the algorithm that aims to improve the performance especially in those cases where finding a counterexample to inclusion requires us to build representatives of trees with higher depths or in the cases where the inclusion holds. Unlike the original approach that moves only one pair from *Next* to *Visited* at the beginning of each iteration of the main loop, we add the newly created pairs $(p_\mathcal{A}, P_\mathcal{B})$ into *Next* and *Visited* at the same time (immediately after they are generated). Our experiments showed that this allows *Visited* converge faster towards the fixpoint.

Finally, yet another optimisation of the algorithm presented in [BHH+08] appeared in [ACH+10]. This optimisation maintains the sets *Visited* and *Next* as antichains w.r.t. $(\preceq_U, \succeq_U^{\exists\forall})^2$. Hence, more pairs can be discarded from these sets. Moreover, for pairs that cannot be discarded, one can at least reduce the sets on their right-hand side by removing states that are simulated by some other state in these sets (this is based on the observation that any tree accepted from an upward-simulation-smaller state is accepted from an upward-simulation-bigger state too). Finally, one can also use upward simulations between states of the two automata being compared. Then, one can discard any pair $(p_\mathcal{A}, P_\mathcal{B})$ such that there is some $p_\mathcal{B} \in P_\mathcal{B}$ that upward-simulates p_A because it is then clear that no tree can be accepted from $p_\mathcal{A}$ that could not be accepted from $p_\mathcal{B}$. All these optimisations are also available in VATA and can optionally be used—they are not used by default since the computation of the upward simulation can be quite costly (as observed by the experimental results of Chapter 8.

10.2.4. Computing Simulation over LTS

The explicit part of VATA uses a highly optimised LTS simulation algorithm proposed in [RT07] and further improved in [HŠ09]. The main idea of the algorithm is to start with an over-approximation of the simulation preorder (a possible initial approximation is the relation $Q \times Q$), which is then iteratively pruned whenever it is discovered that the simulation relation cannot hold for certain pairs of states. For a better efficiency, the algorithm represents the current approximation R of the simulation using a so-called *partition-relation pair*. The partition splits the set of states into subsets (called *blocks*) whose elements are equivalent w.r.t. R, and the relation R is lifted to these blocks.

In order to be able to deal with the partition-relation pair efficiently, the algorithm needs to record for each block a matrix of counters of size $|Q||\Sigma|$ where, for the given LTS, Q is the set of states and Σ is the set of labels. The counters are used to count how many transitions going from the given state via a given symbol a lead to states in the given block (or blocks currently considered to be bigger w.r.t. the simulation). This information is then used to optimise re-computation of the partition-relation pair when pruning the current approximation of the simulation relation being computed (for details see e.g. [RT07]). Since the number of blocks can (and often does) reach the number of states, the naïve solution requires $|Q|^2|\Sigma|$ counters in the worst case. It turns out that this is one of the main barriers which prevents the algorithm from scaling to systems with large alphabets and/or large sets of states.

Working towards a remedy for the above problem, one can observe that the mentioned algorithm actually works in several phases. At the beginning, it creates an initial estimation of the partition-relation pair, which typically contains large equivalence classes. Then it initialises the counters for each element of the partition. Finally, it starts the iterative partition splitting. During this last phase, the counters are only decremented or copied to the newly created blocks. Moreover, the splitting of some block is itself triggered by decrementing some set of counters to 0. In practice, late phases of the iteration typically witness a lot of small equivalence classes having very sparsely populated counters with 0 being the most abundant value.

[2]One says that $P \preceq_U^{\exists\forall} Q$ holds iff $\forall p \in P \; \exists q \in Q : \; p \preceq_U q$. Note also that the upward simulation must be parameterised by the identity in this case [ACH+10].

This suggests that one could use sparse matrices containing only the non-zero elements. Unfortunately, according to our experience, this turns out to be the worst possible solution which strongly degrades the performance. The reason is that the algorithm accesses the counters very frequently (it either increments them by one or decrements them by one), hence any data structure with non-constant time access causes the computation to stall. A somewhat better solution is to record the non-zero counters using a hash table, but the memory requirements of such representation are not yet reasonable.

Instead, we are currently experimenting with storing the counters in blocks, using a copy-on-write approach and a zeroed-block deallocation. In short, we divide the matrix of counters into a list of blocks of some fixed size. Each block contains an additional counter (a block-level counter) that sums up all the elements within the block. As soon as a block contains a single non-zero counter only, it can safely be deallocated—the content of the non-zero counter is then recorded in the block-level counter.

Our initial experiments show that, using the above approach, one can easily reduce the memory consumption by the factor of 5 for very large instances of the problem compared to the array-based representation used in [HŠ09]. The best value to be used as the size of blocks of counters is still to be studied—after some initial experiments, we are currently using blocks of size $\sqrt{|Q|}$.

10.3. Experimental Evaluation of VATA

In order to illustrate the level of optimisation that has been achieved in Vata and that can be exploited in its applications (such as the Forester tool considered in Chapters 3–5), we compared its performance against Timbuk and the prototype library considered in [HLŠV11], which—despite its prototype status—already contained a quite efficient TA implementation.

We compared the performance of the explicit encoding of Vata with Timbuk for union and intersection of more than 3 000 pairs of TAs. On average, Vata was over 20 000 times faster on union and over 100 000 times faster on intersection. The comparison of the implemented inclusion checking algorithms can be found in Chapters 8 and 9.

10.4. Conclusion

In this chapter, we gave a description of Vata, a new efficient and open-source nondeterministic tree automata library, which supports both explicit and semi-symbolic encoding of the tree automata transition function. Up to our knowledge, it is currently the most efficient library for manipulating tree automata. Since its introduction, it has already been used by a few researchers around the world as an efficient underlying library for handling nondeterministic automata for their own techniques (such as for testing language inclusion of TAs in the decision procedure for separation logic of [IRV14], or for the computation of the simulation relation in algorithm for checking language equivalence of nondeterministic finite automata of [BP13]).

In the future, we wish to extend the library with more representations of automata (e.g. with a fully-symbolic representation) and support more operations, such as determinisation (which, however, is generally desired to be avoided), or complementation (which we so far do not know how to compute without first determinising the automaton).

11. Conclusions and Future Directions

Each of the main chapters contains detailed conclusions concerning the specific topic. Here, we summarise once more the main points and discuss possible further research directions.

11.1. A Summary of the Contributions

The main focus of this thesis was on improving the state of the art in shape analysis. This high-level goal was addressed by contributions in the following three areas. In the first area of forest automata-based shape analysis, we developed an extension of the analysis proposed in [HHR$^+$12] that allows it to run fully automatically, without user intervention. The extension is based on learning boxes, i.e. lower-level forest automata describing repeated substructures of the considered complex dynamic data structure, which needed to be manually provided by the user in the original analysis. The boxes are inferred automatically from the structure of the sets of heap graph that occur during the run of the analysis. Moreover, we extended the analysis even further by considering the relations between the data stored in the heap cells. We trace ordering relations between the data stored, which allows us to verify programs such as various sorting algorithms (bubblesort and insertsort over lists), programs with binary search trees, or programs with skip lists of two and three levels.

In the second area, which focused on the development of decision procedures for various logics, we proposed the following two procedures: First, we proposed a decision procedure for testing entailment in a fragment of separation logic that contains various flavours of lists that appear in practice. The decision procedure is based on decomposing the whole entailment query into several lower-level queries and deciding those by translating them into the tree automata membership problem. Second, we proposed a decision procedure for testing validity of WS1S formulae. The decision procedure is based on transforming the formula to be decided into the prenex normal form, constructing a finite automaton for the matrix of the formula, and, finally, processing the prefix of the formula using a technique that is a generalisation of the antichain principle from testing universality and language inclusion of finite automata.

In the third area focusing on finding new and improving existing techniques for manipulating nondeterministic tree automata, we contributed by the following results. We developed a new technique for testing language inclusion that is based on a downward traversal through the automaton. We further augmented the basic technique with the use of antichains and simulations, and also proposed more advanced optimisations. According to our experiments, the technique performs often better than the so far used upward inclusion checking, which is based on upward determinisation of the automaton.

Moreover, we also proposed a semi-symbolic encoding of nondeterministic tree automata and developed algorithms for automata operations (including some more advanced like computing the maximum downward simulation relation on the states of the automaton, or checking language inclusion of a pair of automata) over this encoding. Our work in exploring efficient techniques for handling nondeterministic tree automata culminated in the development of the VATA library, where these techniques are implemented, and which is, as far as we know, currently the most efficient library for manipulating nondeterministic tree automata available.

11.2. Further Directions

There are many interesting directions of further work. In the area of automata-based shape analysis, an interesting direction is to consider a more general notion than the currently used formalism of forest automata. One option would be to remove the restriction that the boxes cannot be recursive. Such a change would increase the expressive power of forest automata, allowing them to express such data structures as trees with linked leaves or skip lists of an arbitrary height. On the other hand, the box folding and learning algorithms would need to be significantly re-designed. Another option would be to adopt a different model, based e.g. on the encoding of inductive higher-order predicates used in the decision procedure for separation logic of Iosif *et al* [IRV14]. Yet another option, this one relating to the data-related component of the analysis, is extension of the abstract data domain to more general relations than currently considered, or even creating a generalised framework that would allow one to plug in any abstract domain that meets certain requirements. In any case, we wish to extend the forest automata-based shape analysis with a counterexample-guided abstraction refinement (CEGAR) loop and use predicate language abstraction on the forest automata instead of the coarse finite height abstraction used now. We believe that the use of the more refined abstraction should allow us to verify some data structures that we currently cannot handle due to the abstraction used, such as red-black trees.

A further interesting future direction is the development of an approach that would allow verification of memory allocators (such as the `ptmalloc()` allocator used in the `glibc` library), which is a truly challenging task due to the complex overlaid shape of the used data structures. A more general representation would also be needed for the verification of some concurrent programs with dynamic memory, e.g. lock-free implementations of concurrent skip lists. In this setting, the invariant of the sequential skip list, which we are currently able to infer, is broken in this particular lock-free concurrent setting, and forest automata, as defined, cannot represent it (because the pointers in the structure overlap and do not create the nested hierarchy from the sequential algorithm, we cannot fold the lower levels into boxes any more). Nevertheless, we plan to apply the shape analysis to verification of concurrent programs, combining it e.g. with the ideas of Abdulla *et al* [AHH$^+$13].

Regarding our decision procedure for separation logic, in future, we wish to continue with extending its generality. In particular, we would like to weaken the limitations on the Boolean structure of the formulae, and, moreover, we would also like to explore whether it is possible to combine it with the decision procedure from [IRV14], which considers more general inductive definitions. For the decision procedure for WS1S, there are several possibilities. We wish to extend the decision procedure to WSkS for an arbitrary k by the use of tree automata and, probably, an algorithm with a structure similar to the structure of the algorithms for downward language inclusion testing of nondeterministic tree automata that were presented in this thesis. We also plan to generalise our notion of symbolic terms in the algorithm to reduce the number of states of the automaton for the matrix of a formula. We believe that our proposed decision procedure is only the start of a new research direction searching for techniques for efficiently deciding WSkS formulae, combining heuristics from both automata theory and formal logic.

Even though the methods for manipulating nondeterministic finite tree (and word) automata have seen a great advance in the recent years, as shown by a recent algorithm for testing equivalence and inclusion of nondeterministic finite word automata of Bonchi and Pous [BP13], there is still a space for improvement. We wish to generalise their algorithm to testing inclusion of nondeterministic tree automata, both for the upward and downward direction of traversal through the automata. We also wish to keep exploring yet other possibilities for reducing the state space in checking language inclusion of nondeterministic finite and tree automata. Furthermore, one of our future goals is to develop an efficient technique for reducing nondeterministic finite automata, both for words and trees, going beyond the capabilities of the techniques based on the simulation equivalence. In the area of symbolic representation of finite word and tree automata, we wish to explore different encodings, suitable for particular needs, such as for the use in the decision procedures of various logics (e.g. WSkS) or for the verification of hardware.

11.3. Publications Related to this Thesis

The results presented in this thesis were originally published in the following papers. The automated approach for learning boxes in the forest automata-based shape analysis, together with the refined technique for abstraction, appeared in [HLR$^+$13]. The data extension of the forest automata-based shape analysis was published as [AHJ$^+$13] (and later extended in [AHJ$^+$15]). The decision procedures for separation logic with list predicates was published in [ELSV14a], and the decision procedure for WS1S has been accepted to appear as [FHLV15]. Our algorithms for manipulating nondeterministic tree automata were published in [HLŠV11] (the downward inclusion checking and the algorithms for the semi-symbolic representation) and the description of our tree automata library appeared in [LŠV12].

Bibliography

[AACJ09] Parosh Aziz Abdulla, Muhsin Atto, Jonathan Cederberg, and Ran Ji. Automated analysis of data-dependent programs with dynamic memory. In *Proc. of ATVA'09*, volume 5799 of *LNCS*, pages 197–212. Springer, 2009.

[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.

[AHH$^+$13] Parosh Aziz Abdulla, Frédéric Haziza, Lukáš Holík, Bengt 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.

[AHJ$^+$13] Parosh Aziz Abdulla, Lukáš Holík, Bengt Jonsson, Ondřej Lengál, Cong Quy Trinh, and Tomáš Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. In *Proc. of ATVA'13*, volume 8172 of *LNCS*, pages 224–239. Springer, 2013.

[AHJ$^+$15] Parosh Aziz Abdulla, Lukáš Holík, Bengt Jonsson, Ondřej Lengál, Cong Quy Trinh, and Tomáš Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. *Acta Informatica*, 2015. Accepted for publication.

[AJMd02] Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso. Regular tree model checking. In *Proc. of CAV'02*, volume 2404 of *LNCS*, pages 555–568. Springer, 2002.

[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):158–192, 2011.

[BCC$^+$07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *Proc. of CAV'07*, volume 4590 of *LNCS*, pages 178–192. Springer, 2007.

[BCI11] Josh Berdine, Byron Cook, and Samin Ishtiaq. SLAYER: Memory safety for systems-level code. In *Proc. of CAV'11*, volume 6806 of *LNCS*, pages 178–183. Springer, 2011.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *Proc. of FSTTCS'04*, volume 3328 of *LNCS*, pages 97–109. Springer, 2005.

[BCO06] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proc. of FMCO'05*, volume 4111 of *LNCS*, pages 115–137. Springer, 2006.

[BDES12] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In *Proc. of ATVA'12*, volume 7561 of *LNCS*, pages 167–182. Springer, 2012.

[BFGP14] James Brotherston, Carsten Fuhs, Nikos Gorogiannis, and Juan Navarro Pérez. A decision procedure for satisfiability in separation logic with inductive predicates. In *Proc. of CSL-LICS'14*, pages 25:1–25:10. ACM, 2014.

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

[BGP12] James Brotherston, Nikos Gorogiannis, and Rasmus Lerchedahl Petersen. A generic cyclic theorem prover. In *Proc. of APLAS'12*, volume 7705 of *LNCS*, pages 350–367. Springer, 2012.

[BH$^+$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.

[BHRV06] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *Proc. of SAS'06*, volume 4134 of *LNCS*, pages 52–70. Springer, 2006.

[BHRV12] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. Abstract regular (tree) model checking. *International Journal on Software Tools for Technology Transfer*, 14(2):167–191, 2012.

[BHT06] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Lazy shape analysis. In *Proc. of CAV'06*, volume 4144 of *LNCS*, pages 532–546. Springer, 2006.

[Bou11] Tony Bourdier. Tree automata based semantics of firewalls. In *Proc. of SAR-SSI'11*, pages 1–8. IEEE, 2011.

[BP13] Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In *Proc. of POPL'13*, pages 457–468. ACM, 2013.

[BR06] Jesse Bingham and Zvonimir Rakamarić. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *Proc. of VMCAI'06*, volume 3855 of *LNCS*, pages 207–221. Springer, 2006.

[Bry86] Randall E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, C-35:677–691, 1986.

[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` (May 2010).

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM, 1977.

[CDG$^+$07] Hubert Comon, Max Dauchet, Remi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications.* 2007. released October 12th, 2007.

[CDNQ12a] 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. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.

[CDNQ12b] 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.

[CDOY09] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proc. of POPL'09*, pages 289–300. ACM, 2009.

[CHO$^+$11] Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. Tractable reasoning in a fragment of separation logic. In *Proc. of CONCUR'11*, volume 6901 of *LNCS*, pages 235–249. Springer, 2011.

[CJH$^+$09] Philipp Claves, Dorothea Jansen, Sezar Jarrous Holtrup, Martin Mohr, Anton Reis, Maria Schatz, and Irene Thesing. The LETHAL library, 2009. Available from `http://lethal.sourceforge.net/`.

[CLQR07] Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakaramarić. A reachability predicate for analyzing low-level software. In *Proc. of TACAS'07*, volume 4424 of *LNCS*, pages 19–33. Springer, 2007.

[CMZ$^+$97] Edmund M. Clarke, Kenneth L. McMillan, Xudong Zhao, Masahiro Fujita, and Jerry Chih Yuan Yang. Spectral transforms for large Boolean functions with applications to technology mapping. *FMSD*, 10, 1997.

[CR08] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Proc. of POPL'08*, pages 247–260. ACM, 2008.

[CRN07] Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In *Proc. of SAS'07*, volume 4634 of *LNCS*, pages 384–401. Springer, 2007.

[CYO01] Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *Proc. of FSTTCS'01*, volume 2245 of *LNCS*, pages 108–119. Springer, 2001.

[DEG06] Jyotirmoy V. Deshmukh, Ernest Allen Emerson, and Prateek Gupta. Automatic verification of parameterized data structures. In *Proc. of TACAS'06*, volume 3920 of *LNCS*, pages 27–41. Springer, 2006.

[DPV13] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Byte-precise verification of low-level list manipulation. In *Proc. of SAS'13*, volume 7935 of *LNCS*, pages 215–237. Springer, 2013.

[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.

[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.

[ELSV14a] Constantin Enea, Ondřej Lengál, Mihaela Sighireanu, and Tomáš Vojnar. Compositional entailment checking for a fragment of separation logic. In *Proc. of APLAS'14*, volume 8858 of *LNCS*, pages 314–333. Springer, 2014.

[ELSV14b] Constantin Enea, Ondřej Lengál, Mihaela Sighireanu, and Tomáš Vojnar. Spen, 2014. Available from `http://www.liafa.univ-paris-diderot.fr/spen`.

[ESS13] Constantin Enea, Vlad Saveluc, and Mihaela Sighireanu. Compositional invariant checking for overlaid and nested linked lists. In *Proc. of ESOP'13*, volume 7792 of *LNCS*, pages 129–148. Springer, 2013.

[FHLV14] Tomáš Fiedor, Lukás 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*, pages 658–674. Springer, 2015.

[GBC11] Alexey Gotsman, Josh Berdine, and Byron Cook. Precision and the conjunction rule in concurrent separation logic. *Electronic Notes in Theoretical Computer Science*, 276:171–190, 2011.

[Gen03] Thomas Genet. Timbuk/Taml: A tree automata library, 2003. Available from `http://www.irisa.fr/lande/genet/timbuk`.

[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.

[GVA07] Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *Proc. of PLDI'07*, pages 256–265. ACM, 2007.

[HHK95] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. of FOCS'95*, pages 453–462. IEEE, 1995.

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

[HIOP13] Christoph Haase, Samin Ishtiaq, Joël Ouaknine, and Matthew J. Parkinson. SeLoger: A tool for graph-based reasoning in separation logic. In *Proc. of CAV'13*, volume 8044 of *LNCS*, pages 790–795, 2013.

[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.

[HLR$^+$13] Lukáš Holík, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Fully automated shape analysis based on forest automata. In *Proc. of CAV'13*, volume 8044 of *LNCS*, pages 740–755. Springer, 2013.

[HLŠV11] Lukáš Holík, Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. Efficient inclusion checking on explicit and semi-symbolic tree automata. In *Proc. of ATVA'11*, volume 6996 of *LNCS*, pages 243–258. Springer, 2011.

[HNR10] Jonathan Heinen, Thomas Noll, and Stefan Rieger. Juggrnaut: Graph grammar abstraction for unbounded heap structures. In *Proc. of TTSS'09*, volume 266 of *ENTCS*, pages 93–107. Elsevier, 2010.

[Hos11] Haruo Hosoya. *Foundations of XML Processing: The Tree Automata Approach.* Cambridge University Press, 2011.

[HŠ09] Lukáš Holík and Jiří Šimáček. Optimizing an LTS-simulation algorithm. In *Proc. of MEMICS'09*, pages 93–101. Faculty of Informatics MU, 2009.

[HVP05] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27, 2005.

[INY04] Lucian Ilie, Gonzalo Navarro, and Sheng Yu. On NFA reductions. In *Theory Is Forever*, volume 3113 of *LNCS*, pages 112–124. Springer, 2004.

[IO01] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. of POPL'01*, pages 14–26. ACM, 2001.

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

[IRV14] Radu Iosif, Adam Rogalewicz, and Tomáš Vojnar. Deciding entailments in inductive separation logic with tree automata. In *Proc. of ATVA'14*, volume 8837 of *LNCS*, pages 201–218. Springer, 2014.

[JJSK97] Jakob L. Jensen, Michael E. Jørgensen, Michael I. Schwartzbach, and Nils Klarlund. Automatic verification of pointer programs using monadic second-order logic. In *Proc. of PLDI'97*, pages 226–234. ACM, 1997.

[KM01] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual.* BRICS, Department of Computer Science, Aarhus University, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3.

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

[LGQC14] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In *Proc. of CAV'14*, volume 8559 of *LNCS*, pages 52–68. Springer, 2014.

[LQ08] Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: Revisiting precise program verification using SMT solvers. In *Proc. of POPL'08*, pages 171–182. ACM, 2008.

[LRS05] Alexey Loginov, Thomas Reps, and Mooly Sagiv. Abstraction refinement via inductive learning. In *Proc. of CAV'05*, volume 3576 of *LNCS*, pages 519–533. Springer, 2005.

[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.

[LYP11] Oukseh Lee, Hongseok Yang, and Rasmus Petersen. Program analysis for overlaid data structures. In *Proc. of CAV'11*, volume 6806 of *LNCS*, pages 592–608. Springer, 2011.

[Mey72] Albert R. Meyer. Weak monadic second order theory of successor is not elementary-recursive. In Rohit Parikh, editor, *Proc. of Logic Colloquium—Symposium on Logic Held at Boston, 1972–73*, volume 453 of *Lecture Notes in Mathematics*, pages 132–154. Springer, 1972.

[MN05] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In *Proc. of CAV'05*, volume 3576 of *LNCS*, pages 476–490. Springer, 2005.

[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.

[MS01] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Proc. of PLDI'01*, pages 221–231. ACM, 2001.

[MTLT10] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proc. of POPL'10*, pages 211–222. ACM, 2010.

[PR11] Juan Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *Proc. of PLDI'11*, pages 556–566. ACM, 2011.

[Pug90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[PWZ13] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using SMT. In *Proc. of CAV'13*, volume 8044 of *LNCS*, pages 773–789. Springer, 2013.

[QGSM13] Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and Parthasarathy Madhusudan. Natural proofs for structure, data, and separation. In *Proc. of PLDI'13*, pages 231–242. ACM, 2013.

[QHL$^+$13] Shengchao Qin, Guanhua He, Chenguang Luo, Wei-Ngan Chin, and Xin Chen. Loop invariant synthesis in a combined abstract domain. *Journal of Symbolic Computation*, 50:386–408, 2013.

[RBHC07] Zvonimir Rakamarić, Roberto Bruttomesso, Alan J. Hu, and Alessandro Cimatti. Verifying heap-manipulating programs in an SMT framework. In *Proc. of ATVA'07*, volume 4762 of *LNCS*, pages 237–252. Springer, 2007.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS'02*, pages 55–74. IEEE, 2002.

[RT07] Francesco Ranzato and Francesco Tapparo. A new efficient simulation equivalence algorithm. In *Proc. of LICS'07*, pages 171–180. IEEE, 2007.

[sl-14] SL-COMP'14, 2014. Available from `http://smtcomp.sourceforge.net/2014/results-SLCOMP2.shtml`.

[Som11] Fabio Somenzi. CUDD: CU decision diagram package release 2.4.2, 2011. Available from `http://vlsi.colorado.edu/~fabio/CUDD/`.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.

[TH03] Akihiko Tozawa and Masami Hagiya. XML schema containment checking based on semi-implicit techniques. In *Proc. of CIAA'03*, volume 2759 of *LNCS*, pages 213–225. Springer, 2003.

[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*, pages 17–30. Springer, 2006.

[Wei12] Alexander Dominik Weinert. *Inferring Heap Abstraction Grammars.* RWTH Aachen, 2012. BSc. thesis.

[WKZ$^+$07] Thomas. Wies, Viktor Kuncak, Karen Zee, Andreas Podelski, and Martin Rinard. On verifying complex properties using symbolic shape analysis. In *Proc. of HAV'07*, 2007.

[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.

[WP10] Thomas Wies and Andreas Podelski. Counterexample-guided focus. In *Proc. of POPL'10*, pages 249–260. ACM, 2010.

[YLB$^+$08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *Proc. of CAV'08*, volume 5123 of *LNCS*, pages 385–398. Springer, 2008.

[ZKR08] Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In *Proc. of PLDI'08*, pages 349–361. ACM, 2008.