DISJOINT FRACTIONAL PERMISSIONS IN VERIFICATION: APPLICATIONS, SYSTEMS AND THEORY

XUAN-BACH LE
Bachelor in Computer Science (NUS 2012, First-class Honour)
Bachelor in Mathematics (NUS 2012, First-class Honour)

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2017

Supervisor:
Dr Aquinas Hobor
Mentor:
Dr Anthony W. Lin, University of Oxford
Examiners:
Professor Joxan Jaffar
Professor Olivier Gerard Henri Marie Danvy
Dr James Brotherston, University College London

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Xuan-Bach Le
14 November 2017

Acknowledgments

First of all, I am grateful to my supervisor, Aquinas Hobor, for his delicate supervision during my PhD study. I am very fortunate to be his first PhD student and thus I have received enormous supports and motivations from him so that I can become an independent researcher. My PhD journey with him, in retrospect, has been so much relaxing, enjoyable and memorable.

I would like to express my sincere gratitude to my mentor, Anthony W. Lin, for his constant counseling and supports. He has taught me many valuable lessons in term of technical research and social life that have shaped me to become the person I am now. Moreover, this thesis would not be possible without his guidance during my PhD.

I want to thank Prof. Frank Stephan, Prof. Sanjay Jain and Prof. Yang Yue for their wonderful courses on automata theory, complexity and logic. The materials from their courses have helped me significantly for my PhD topic. I also thank my (other) collaborators –Cristian Gherghina, Thanh-Toan Nguyen and Prof. Wei-Ngan Chin– for their helps to get the papers published.

I wish to thank Prof. Olivier Danvy, Prof. Joxan Jaffar and Prof. James Brotherston, for being my examiners as well as their precious suggestions that has shaped several directions for this thesis during its early form.

I take this opportunity to record my sincere thanks to Anshuman Mohan for his comprehensive review over my thesis, and to Vinh Ho, Shengyi Wang, Andreea Costea, for their partial reviews and helpful suggestions.

I would like express my sense of gratitude to my parents, Van-Tuong Le and Kim-Cuc Pham, for their mental support and unceasing encouragement. I also

want to thank my seniors Quang-Loc Le and Duc-Hiep Chu for their honest advices and recommendations. Last but not least, I want to thank Yu-Fang Chen and his colleges for their hospitality during my stay at Academia Sinica, Taiwan; and Tan, Than, Bao, Vu, Long, Dai, Quang for their friendship and willingness to listen and give advice to me during my PhD study.

Abstract

Fractional permissions enable sophisticated management and reasoning of resource ownership and sharing in Separation Logic (SL). One of the most common models of permission consists of rational numbers in $[0, 1]$ and uses addition for splitting/joining permissions. To support the verification task, rational permissions are embedded into SL formulae through the *fractional maps-to* $x \overset{p}{\mapsto} v$ which asserts that the value v is stored at address x with permission p. Using such notation, it is convenient to express the notion of resource sharing, *i.e.*, a thread that possesses the resource $x \overset{p_1+p_2}{\longmapsto} v$ can split it into $x \overset{p_1}{\longmapsto} v$ and $x \overset{p_2}{\longmapsto} v$ and pass the latter to its child thread. While the rational model is simple, it poses a technical challenge to the core *separation property* of SL that allows smooth compositional reasoning. In particular, SL has a special operator $*$ called *separating conjunction* to specify the disjointness of resources, *e.g.*, $x \mapsto 1 * y \mapsto 1$ is only satisfiable if x and y are two different addresses. On the other hand, the two addresses x, y in the predicate $x \overset{0.5}{\longmapsto} 1 * y \overset{0.5}{\longmapsto} 1$ could be aliasing because $x \overset{0.5}{\longmapsto} 1 * x \overset{0.5}{\longmapsto} 1$ is equivalent to $x \overset{1}{\mapsto} 1$ which is satisfiable. As a result, there has been substantial work in proposing better models for fractional permissions in the last twenty years.

In this thesis, we study the fractional permission model of tree shares proposed by Dockins *et al.* as a novel treatment to the disjointness problem. The tree domain consists of boolean binary trees in canonical form; and instead of addition, we have the "join" operator $\oplus$ to regulate resource sharing. Furthermore, we have a special tree multiplication-like operator $\bowtie$ called "bowtie" that is useful to assign permissions over arbitrary predicates. Our main contribution is to investigate and extend the research knowledge of tree shares via three pillars: applications, system, and theory.

In term of applications, we demonstrate the embedding of tree shares into SL formulae to reason about shared resources in concurrent context. The demonstration is two-fold: first, we show how to embed tree shares into SL assertion language as well as how to extract tree share constraints from SL formulae. Second, we use tree shares as the underlying structure to develop a general logic framework with predicate multiplication that allows permission reasoning over arbitrary predicates. Our approach can handle sophisticated verification tasks such as bi-abduction inference, inductive predicates and precision reasoning.

Second, we achieve the systems pillar by developing a set of decision procedures over tree equations drawn from the program proof. Our decision procedures are sound and complete and benchmarked in the HIP/SLEEK verification toolset. Subsequently, we refine and improve the procedures so that they can also handle negative constraints with reasonable performance. Furthermore, the procedures are certified in the theorem prover Coq, which can be extracted to OCaml using the Coq extraction feature.

Lastly, we investigate the decidability and complexity of the tree share structure. We obtain a detailed view of the complexity by studying different overlapped sub-structures. Using this approach, we manage to find interesting complexity results that vary from polynomial time to non-elementary. Along the way, we establish several sophisticated connections between the tree share structure and other well-known domains such as automatic structures, word equation and Boolean Algebra. Such resemblances suggest certain problems in these domains can be reduced to tree structure if the tree encoding is more pleasant to handle or vice-versa. For instance, we can transform tree share constraints into word equation constraints and then use standard string solvers to handle them.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

> "*Our lives are not our own. We are bound to others, past and present, and by each crime and every kindness, we birth our future.*"
>
> David Mitchell, *Cloud Atlas.*

1.1 Motivation

In the last decade, there has been substantial progress in the study of formal methods for concurrency reasoning in both theory ([dRPDYG14, DYDG+10, HMP17, JSS+15, SB14, TDB13, ORY01, IO01]) and verification tools ([FLLV15, HG12, KLVU10, LCT15, MHWL12, SNB15, DYdAB17]). The main challenge of this topic is that shared resources can lead to race condition among threads, which results in nondeterministic outcomes. One standard solution is the use of locks or semaphores to protect the shared resources from interference, *i.e.*, by establishing mutual exclusion in critical sections. As a result, it is desirable to provide a foundational semantics that is capable of reasoning about the race-free condition and a formal proof system to assist verification tools. O'Hearn approached this problem by proposing Concurrent Separation Logic (CSL) [OHe07], which is an extension of Separation Logic (SL) [Rey02]. A model for this logic was first invented by Brookes [Bro07a] using trace semantics in which traces are sequences of transition machine states to bookkeep resources. CSL has received enormous attention from researchers and has become one of the central topics in program verification ([BCOP05, Boy03, DHA09, HHH08, PBC06, GBC11, HW06, Hob08, HAZ08, Vaf11, Vaf07]).

Reynolds *et al.* ([Rey02, IO01, ORY01]) developed SL as a formal tool to prove correctness of programs with resources. One key feature of SL is the separating conjunction $*$ that partitions program heap into disjoint components. For example, predicate $x \mapsto v_1 * y \mapsto v_2$ expresses the fact that addresses x and y are disjoint and thus modifying the content of one address will not affect the content of the other. Such disjointness property helps prevent any further pointer aliasing complications that a verifier has to consider. Using $\star$, one has the Frame rule (Eqn. 1.1) which is a powerful tool for local reasoning. Here c is the executing command whereas P, Q, F are predicates describing the heap states. Specifically, P is the precondition, Q is the postcondition, F is the frame that represents irrelevant parts of the heap and the triple $\{P\}\mathsf{c}\{Q\}$ represents the transition of the heap state from P to Q when c is executed. Briefly speaking, the rule says that it suffices to consider the local state P when reasoning about c if any variable modified by c is not a free variable of F, or in other words, c is independent of F.

$$\frac{\{P\}\ \mathsf{c}\ \{Q\} \qquad \mathsf{mod}(\mathsf{c}) \cap \mathsf{fv}(F) = \emptyset}{\{F * P\}\ \mathsf{c}\ \{F * Q\}}\ \mathsf{Frame} \tag{1.1}$$

The CSL developed by O'Hearn [OHe07] contains a crucial assumption that if threads do not share resources then they should not interfere with each others. Consequently, the heap can be seen as a combination of disjoint components possessed by individual threads. The parallel composition $\mathsf{c}_1||\mathsf{c}_2$ expresses the concurrent execution of two programs c_1 and c_2. Its behavior is portrayed as the Parallel rule (Eqn. 1.2).

$$\frac{\begin{array}{ll}\{P_1\}\ \mathsf{c}_1\ \{Q_1\} & \mathsf{fv}(\mathsf{c}_1, P_1, Q_1) \cap \mathsf{mod}(\mathsf{c}_2) = \emptyset \\ \{P_2\}\ \mathsf{c}_2\ \{Q_2\} & \mathsf{fv}(\mathsf{c}_2, P_2, Q_2) \cap \mathsf{mod}(\mathsf{c}_1) = \emptyset\end{array}}{\{P_1 * P_2\}\ \mathsf{c}_1\ ||\ \mathsf{c}_2\ \{Q_1 * Q_2\}}\ \mathsf{Parallel} \tag{1.2}$$

In short, the rule says if two commands c_1 and c_2 do not interfere with each other then running c_1 and c_2 concurrently with the combined precondition $P_1 * P_2$ will result in the combined postcondition $Q_1 * Q_2$. Although this rule is useful to verify race-free programs, its usage is significantly limited by the fact that a number of concurrent programs are purposely designed to not be race-free. As a result, the language and semantics of CSL need to be

extended to capture the reasoning of resource sharing. One solution is to tag resources with *permissions* (ownerships) that dictate certain actions to be applied to them, *e.g.*, read and write permission. Hence the *fractional maps-to* $x \overset{p}{\mapsto} v$ indicates the address x with value v and non-empty permission p (*e.g.* $p \neq 0$ for rational permissions). One of the original uses of fractional permissions was to assert resource sharing via locks [OHe07]. Furthermore, it is desirable to have some mechanisms that regulate the distribution of permissions, *i.e.*, splitting and joining. Generally, a permission model $\mathcal{P} = \langle \mathbb{P}, \oplus \rangle$ consists of the domain $\mathbb{P}$ equipped with a partial *join* operator $\oplus$ to monitor the splitting and combining of permissions.

Example 1.1.1. The rational model $\mathcal{Q} = \langle [0,1], + \rangle$ proposed by Boyland [Boy03] contains all rationals in $[0,1]$ and $p_1 + p_2$ is defined if their sum is at most 1. Here 0 indicates the lack of permission, 1 is the *full permission* and the remaining rationals are called *fractional*. A fractional mapping $x \overset{p}{\mapsto} v$ can be split into two smaller fractional mapping $x \overset{p_1}{\mapsto} v * x \overset{p_2}{\mapsto} v$ s.t. $p = p_1 + p_2$, *e.g.*, $1 \overset{0.8}{\mapsto} 2$ is equivalent to $1 \overset{0.6}{\mapsto} 2 * 1 \overset{0.2}{\mapsto} 2$. ◁

The advantages of $\mathcal{Q}$ are its simple, intuitive representation and efficient computation. In addition, permissions in $\mathcal{Q}$ can be split infinitely and this property is useful to reason about fork-join and recursive programs. Its main disadvantage is the loss of disjointness property that is fundamental to SL. Simply put, while $x \mapsto v * x \mapsto v$ is unsatisfiable in classical SL, this is not the case when fractional permissions in $\mathcal{Q}$ are introduced into the language. In particular, the predicate $x \overset{p}{\mapsto} v * x \overset{p}{\mapsto} v$ can be satisfiable, *e.g.*, $x \overset{0.25}{\mapsto} v * x \overset{0.25}{\mapsto} v$ is equivalent to $x \overset{0.5}{\mapsto} v$ which is satisfiable. The consequence of such unexpected behavior can be clearly visualized using the following recursive predicate definition for fractionally-owned binary trees:

$$\begin{aligned} \mathsf{tree}(\tau, p) \overset{\text{def}}{=} \; & (\tau = \mathsf{null} \;\wedge\; \mathsf{emp}) \;\vee \\ & \exists \tau_l, \tau_r.\; (\tau \overset{p}{\mapsto} (\tau_l, \tau_r) \;*\; \mathsf{tree}(\tau_l, p) \;*\; \mathsf{tree}(\tau_r, p)). \end{aligned} \tag{1.3}$$

This tree predicate is generalized from the standard binary tree definition in SL by asserting only a fraction p ownership of the root and recursively doing the same for the left and right substructures, and so at first glance looks obviously correct. Indeed, when $p \in (0.5, 1]$ then every tree predicate $\mathsf{tree}(\tau, p)$ is actually a tree. Interestingly, when $p \in (0, 0.5]$ then tree

can describe some unintended directed acyclic graphs (dag) such as $\mathsf{tree}(\mathtt{root}, 0.3)$ in Fig.1.1 where `grand` is owned with share $0.3 + 0.3 = 0.6$. One serious consequence is that P and Q in $P * Q$ could share common memory addressees and thus this behavior limits the local reasoning power of SL. We will discuss the disadvantages of rational model closely in §3.6.1.

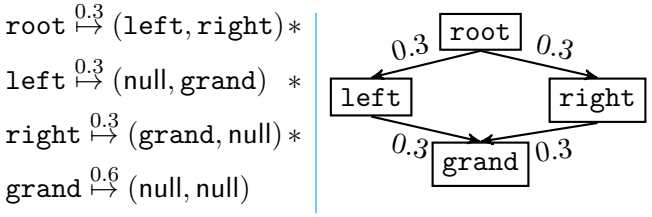


Figure 1.1: Graphical representation of an instance of the predicate $\mathsf{tree}(\tau, 0.3)$

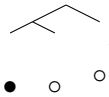
Since then, there has been substantial research to improve the model $\mathcal{Q}$ or replace a better model ([Par05, LCT15, BCOP05, HM15, BMSS14]). Yet none of them are considered adequately reasonable as they suffer at least one of the three main pitfalls: disjointness problem, undecidability or not infinitely-splittable. Dockins *et al.* [DHA09] proposed the following "tree share" model $\mathcal{T} = \langle \mathbb{T}, \oplus \rangle$, which, as we will show, remedies all of the aforementioned issues. A tree share $\tau \in \mathbb{T}$ is simply a binary tree with boolean leaves:

$$\tau \quad \stackrel{\text{def}}{=} \quad \circ \mid \bullet \mid \begin{array}{c} \frown \\ \tau \quad \tau \end{array} \tag{1.4}$$

Here $\circ$ and $\bullet$ denote the empty and full share respectively (similar to 0 and 1 in the rational model $\mathcal{Q}$). We require that all tree shares are in *canonical* form, *i.e.*, they do not contain sub-trees $\begin{array}{c} \frown \\ \bullet \quad \bullet \end{array}$ and $\begin{array}{c} \frown \\ \circ \quad \circ \end{array}$. More precisely, a tree is in canonical form when its representation is the most compact under the equivalence relation $\cong$:

$$\frac{}{\circ \cong \circ} \qquad \frac{}{\bullet \cong \bullet} \qquad \frac{}{\circ \cong \begin{array}{c} \frown \\ \circ \quad \circ \end{array}} \qquad \frac{}{\bullet \cong \begin{array}{c} \frown \\ \bullet \quad \bullet \end{array}} \qquad \frac{\tau_1 \cong \tau_1' \qquad \tau_2 \cong \tau_2'}{\begin{array}{c} \frown \\ \tau_1 \quad \tau_2 \end{array} \cong \begin{array}{c} \frown \\ \tau_1' \quad \tau_2' \end{array}}$$

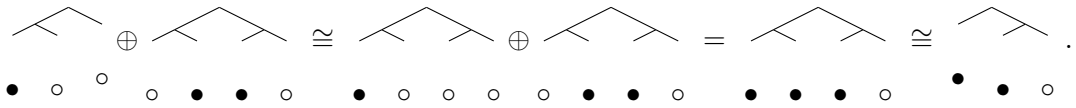
By unfolding the definition 1.4, we obtain two 'half' shares $\begin{array}{c} \frown \\ \circ \quad \bullet \end{array}$ and $\begin{array}{c} \frown \\ \bullet \quad \circ \end{array}$, and four 'quarter'

shares, beginning with  . In contrast, the rational model $\mathcal{Q}$ only has one half permission 0.5 and one quarter permission 0.25. It is a feature that the two half shares are distinct from each other, which explains why $\mathcal{T}$ can solve the disjointness problem.

The join operator $\oplus$ on tree shares requires some maneuvers of tree unfolding/folding to temporarily escape the canonical form. In brief, we unfold two trees τ_1, τ_2 under $\cong$ into the same shape, join them leaf-wise and then fold them back to canonical form. At the leaf level, $\oplus$ behaves similarly as partial addition in which $\circ$ and $\bullet$ are interpreted as 0 and 1:

$$\bullet \oplus \circ \;=\; \circ \oplus \bullet \overset{\text{def}}{=} \bullet \qquad \circ \oplus \circ \overset{\text{def}}{=} \circ \qquad \bullet \oplus \bullet \text{ not defined.}$$

Example 1.1.2.



◁

Because $\bullet \oplus \bullet$ is undefined, the join relation on trees is a partial operation. Dockins *et al.* [DHA09] prove that the join relation satisfies a number of useful axioms *e.g.* associativity and commutativity. One key axiom, not satisfied by $\mathcal{Q} = \langle [0,1], + \rangle$, is "disjointness":

$$x \oplus x = y \;\rightarrow\; x = \circ \,.$$

The corresponding axiom for the rational model would be $x + x = y \;\rightarrow\; x = 0$ which is clearly false. To be precise, if τ is a *positive share*, *i.e.* $\tau \neq \circ$, then the predicate $x \overset{\tau}{\mapsto} v * x \overset{\tau}{\mapsto} v$ is not satisfiable because $\tau \oplus \tau$ is not defined. Interestingly, this property forces the tree predicate in equation 1.3 to behave properly. As tree shares satisfy the disjointness axiom, we will usually refer them as *disjoint permissions* to distinguish them from the non-disjoint rational permissions.

On the other hand, Dockins *et al.* [DHA09] also defined Boolean-like operators for tree shares: $\sqcup$ (union), $\sqcap$ (intersection) and $\bar{\cdot}$ (complement). These operators are generalized

from their standard Boolean counterparts in which the computation is done leaf-wise with the help of $\cong$ to unfold/fold the trees temporarily. Just like rational numbers, tree shares are also equipped with a multiplicative operator $\bowtie$ (bowtie). Briefly speaking, $\tau_1 \bowtie \tau_2$ is defined by replacing all black leaves of τ_1 with an instance of τ_2. We will heavily discuss the formal definitions and properties of the above operators in §2.2.

The appealing theoretical aspect of the tree-share model has been greatly appreciated as a reasonable fractional permission for SL. Hence it is used to design and reason about the soundness proofs of several CSL domains [Hob08, HG11]. Moreover, the pleasant computability of $\oplus$ has led to it being incorporated into several verification tools such as HIP/SLEEK [NDQC07, HG12], VST [App11b] and Heap-Hop [VLC10, Vil11]. Hobor and Gherghina [HG12] showed how to verify entailment between SL formulas with tree shares by splitting it into two independent components, namely a fraction-free SL entailment and an entailment between systems of share equations.

Example 1.1.3. The entailment $x \overset{\pi_1}{\mapsto} a * x \overset{\pi_2}{\mapsto} b \vdash \exists \pi.\ x \overset{\pi}{\mapsto} c$ is divided into the fraction-free $x \mapsto a \wedge a = b \vdash x \mapsto c \wedge c = a$ and the share entailment $\pi_1 \oplus \pi_2 = \pi_3 \vdash \exists \pi.\ \pi = \pi_3$. Here we use the *entailment* symbol $\vdash$ as an alternative for implication $\Rightarrow$. $\triangleleft$

An important technical issue with the tree shares is the absence of algorithms and automatic tools to reason about share constraints. Heap-Hop employed a simplistic heuristic to prove entailments involving tree shares, and although HIP/SLEEK did better by using bounding heuristics [HG12], their tool is still significantly incomplete. For example, it cannot verify the trivial entailment $v_1 \oplus v_2 = v_3 \vdash v_2 \oplus v_1 = v_3$. Moreover, even small programs can generate hundreds of share entailment checks, and this is the main barrier that prevents the tree shares from being widely used in those systems. As a result, **the main goal of this thesis is to conduct a comprehensive study about the tree share model so that our results can provide useful applications and efficient algorithms to reason about permissions in concurrent programs**. The rest of this chapter is organized as follow:

1. In §1.2, we propose three main goals for this thesis, namely the applications, theory and system of tree shares.

2. In §1.3, we briefly introduce the contents of each remaining chapter in the thesis.

1.2 Contributions

In this thesis, we conduct a comprehensive study of the tree share structure and apply our results to solve practical problems in program verification. Our main motivation comes from the observation that tree share structure $\mathcal{T}$ is a good candidate for fractional permissions in concurrency and yet there is little attempt to use it for practical applications. A straightforward application of $\mathcal{T}$ is an upgrade of the rational model $\mathcal{Q} = \langle \mathbb{Q}, +, \times \rangle$ in which $+$ is replaced by $\oplus$ and $\times$ is replaced by $\bowtie$. More importantly, we discover an interesting application of bowtie in constructing scaling permissions as an effective way to manipulate permissions at large scale, *e.g.*, assigning permission to user-defined recursive predicates. However, without reasonable decision procedures over $\mathcal{T}$, those applications are not adequately convincing for automatic SL verifiers such as HIP/SLEEK [NDQC07] and Caper [DYdAB17] to integrate tree shares into their infrastructural core. Hence, another main goal of this thesis is to establish a concrete foundation over decidability and complexity of $\mathcal{T}$ which will ultimately be the guidance to develop decision procedures for $\mathcal{T}$. In summary, there are three main targets that we aim to achieve: applications, systems and theory of $\mathcal{T}$.

1.2.1 Applications of tree shares

When $\mathcal{T} = \langle \mathbb{T}, \oplus, \bowtie \rangle$ was first introduced by Dockins *et al.* [DHA09], it was used to construct a semantic model for fractional permissions. More precisely, $\oplus$ satisfies unique properties such as disjointness (to avoid the deformation of structures) and infinite split-ability (for recursive programs). Furthermore, the computation of $\oplus$ is easy to handle by recursion. On the other hand, $\bowtie$ was briefly mentioned as a helper operator for token counting to split a token tree τ into two token trees $\tau \bowtie$  and $\tau \bowtie$  . We took a further step by implementing $\mathcal{T}$ into HIP/SLEEK [NDQC07], an automatic SL verifier. This requires certain mechanisms on how to embed $\mathcal{T}$ into SL formulas as well as extract the tree share constraints from them.

Contribution 1 (§4). We provide a modular integration of $\langle \mathbb{T}, \oplus \rangle$ into SL formulas. Furthermore, the tree shares constraints can be independently extracted from the SL

formulas to be solved separately.

One important question is how to regulate permissions uniformly at predicate level. In detail, suppose we have a resource described by the predicate R that is shared among threads. Moreover, R can be recursive, *e.g.* list or tree, and thus one cannot simply split/join the permissions address-wise. To simplify, assume all addresses in R have permission τ and we would like to split R into two pieces which are essentially two copies of R but with different permissions, one with $\tau \bowtie$  and the other with $\tau \bowtie$  . What we want can informally be described as "uniformly split all permission τ in R into $\tau \bowtie$  and $\tau \bowtie$  while keeping same addresses and values". Formally, let $\tau \cdot R$ be the *τ-fraction* of R. Then we would like to have the following inference:

$$R \vdash \bullet \cdot R \;\vdash\; (\,\overset{\circ\ \bullet}{\frown} \oplus \overset{\bullet\ \circ}{\frown}\,) \cdot R \;\vdash\; (\,\overset{\circ\ \bullet}{\frown} \cdot R) \;*\; (\,\overset{\bullet\ \circ}{\frown} \cdot R).$$

Contribution 2 (§3). We use $\langle \mathbb{T}, \oplus, \bowtie \rangle$ to develop *scaling permissions* to reason about permissions at large scale, *i.e.*, over arbitrary predicates. Additionally, we explain why the rational model $\mathcal{Q} = \langle [0,1], +, \times \rangle$ is inferior to $\mathcal{T}$ in this aspect. Furthermore, we establish some foundations for the *scaling separation algebra* which is an extension of separation algebra with the scaling operator.

1.2.2 Systems of tree shares

When tree shares were first implemented and used in MSL [ADH09], all tree share formulae were proved manually in Coq. In particular, the proof was derived directly from tree shares properties, or by induction over the structure of tree shares. As Hobor and Gherghina [HG12] attempted to use $\mathcal{T}$ for their barrier structure in automatic verifier HIP/SLEEK, they realized the need for a decision procedure to handle tree share formulas. Formally, we would like to develop a decision procedure that verifies whether a first-order tree share formula Φ is valid. Essentially, this is a model checking problem as the semantics of our formulas is interpreted in the specific tree share domain only. However, this problem is nontrivial because the

domain $\mathbb{T}$ is infinite and therefore a brute-force approach is impossible. A simple solution would be to construct a semi-decision using proof system from Figure 2.3. However, this approach is unreliable as some simple facts about tree shares cannot be derived efficiently (Example 1.2.1).

Example 1.2.1. The formula $\forall a \exists b.\ a \oplus b = \bullet$ is valid while $\exists b \forall a.\ a \oplus b = \bullet$ is invalid. $\triangleleft$

The tree shares constraints from [HG12] are closed formulae expressed using existential form **SAT** and implication form **IMP** that contain positive constraints $a \oplus b = c$:

- **SAT**: $\exists \bar{v}. \bigwedge a \oplus b = c$.
- **IMP**: $\forall \bar{v}. (\exists \bar{v_1}. \bigwedge a \oplus b = c \rightarrow \exists \bar{v_2}. \bigwedge d \oplus e = f)$.

Contribution 3 (§4). We propose sound and complete decision procedures SAT for **SAT** and IMP for **IMP**. Our decision procedures are implemented and benchmarked in HIP/SLEEK.

When modeling fractional permission using tree shares, we exclude the empty tree $\circ$ from the domain $\mathbb{T}$ for two reasons:

1. The redundant predicate $x \overset{\circ}{\mapsto} v$ can be simplified to emp.
2. For a predicate $x \overset{\pi}{\mapsto} v$, it is often the case that we want π to be *positive*, *i.e.* $\pi \neq \circ$, to gain read access as well as to split π into two positive shares π_1, π_2 s.t.:

$$x \overset{\pi}{\mapsto} v \ \wedge \ \pi \neq \circ \ \vdash \ \exists \pi_1 \exists \pi_2.\ x \overset{\pi_1}{\mapsto} v \ * \ x \overset{\pi_2}{\mapsto} v \ \wedge \ \pi = \pi_1 \oplus \pi_2 \ \wedge \ \pi_1 \neq \circ \ \wedge \ \pi_2 \neq \circ.$$

The previous procedures cannot handle negative constraint $x \neq \circ$ properly. In fact, we found several bugs when using them to verify **SAT** and **IMP** constraints with positive shares. On the other hand, positive shares is a special form of negative constraint $\neg(a \oplus b = c)$, *i.e.*, $a \neq \circ$ is equivalent to $\neg(a \oplus \circ = \circ)$. It is worth highlighting that we avoid using the negative form $a \oplus b \neq c$ which means $a \oplus b$ *is defined* and their sum is different from c. In contrast, $\neg(a \oplus b = c)$ contains another possibility that the sum $a \oplus b$ is not defined, *e.g.*, $\neg(\bullet \oplus \bullet = \bullet)$. Thus the general satisfiability and implication problems that contain negative constraints can be expressed as:

- **GSAT**: $\exists \bar{v}.\ \bigwedge a \oplus b = c \bigwedge \neg(a' \oplus b' = c')$.

- **GIMP**: $\forall \bar{v}.\ (\exists \bar{v_1}.\ \bigwedge a \oplus b = c \bigwedge \neg(a' \oplus b' = c')) \rightarrow (\exists \bar{v_2}.\ \bigwedge d \oplus e = f \bigwedge \neg(d' \oplus e' = f'))$.

Contribution 4 (§5). We propose sound and complete decision procedures GSAT for **GSAT** and GIMP for **GIMP**. Our decision procedures are implemented, optimized and certified in Coq. Furthermore, the extracted version using Coq extraction feature is integrated and benchmarked in HIP/SLEEK.

1.2.3 Theory of tree shares

Although decision procedures GSAT and GIMP are adequate for automatic reasoning (*e.g.* in HIP/SLEEK), it is interesting to find out whether the first-order theory of $\langle \mathbb{T}, \oplus \rangle$ is decidable so that we can develop algorithms to answer sophisticated tree share constraints. As join $\oplus$ is defined in term of union $\sqcup$ and intersection $\sqcap$, the question can be generalized to whether the first-order theory of $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot} \rangle$ is decidable. We obtained an affirmative answer to this question together with the exact complexity class.

Contribution 5 (§6). We prove that the first-order theory of $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot} \rangle$ is decidable. Furthermore, its complexity is $\mathsf{STA}(*, 2^{n^{O(1)}}, n)$-complete where $\mathsf{STA}(*, t(n), a(n))$ is the complexity class of alternating Turing machines that use $t(n)$ time and $a(n)$ alternations between universal and existential states and vice-versa.

The bowtie operator $\bowtie$ is critical in constructing the scaling permission, yet there is little research on how to handle tree share constraints with bowtie automatically. In fact, we found out that $\bowtie$ is significantly more complicated than $\oplus$ due to its close connection to string concatenation. Therefore, we are interested in establishing theoretical foundation for bowtie to construct decision procedures for it. We discover that the structure $\langle \mathbb{T} \backslash \{\circ\}, \bowtie \rangle$ is isomorphic to the string structure $\langle \mathbb{S}, \cdot \rangle$ in which $\mathbb{S}$ is some infinitely countable alphabet and $\cdot$ is the string concatenation. Hence, we are able to derive several decidability results for $\bowtie$.

Contribution 6 (§7). We show that the existential theory of $\langle \mathbb{T}, \bowtie \rangle$ is decidable with lower bound NP-hard and upper bound PSPACE*. Furthermore, the first-order theory of $\langle \mathbb{T}, \bowtie \rangle$ is undecidable.

*Turing machines that use polynomial space.

In practical applications, the tree share $\bowtie$-constraints are not always existential. As mentioned above, it is impossible to develop complete decision procedure to handle general tree share $\bowtie$-constraints and thus we are interested in finding a restriction of bowtie in which its first-order theory is decidable. We discover that if either one of the two arguments of $\bowtie$ is fixed to be constant then the decidability of its first-order theory can be recovered. In particular, let $\bowtie_\tau$ be the *τ-right-bowtie* that maps each tree τ' to $\tau' \bowtie \tau$, *i.e.*:

$$\bowtie_\tau \quad \stackrel{\text{def}}{=} \quad \lambda\tau'.\ \tau' \bowtie \tau.$$

Similarly, ${}_\tau\bowtie$ is the *τ-left-bowtie* that maps each tree τ' to $\tau \bowtie \tau'$:

$${}_\tau\bowtie \quad \stackrel{\text{def}}{=} \quad \lambda\tau'.\ \tau \bowtie \tau'.$$

Contribution 7 (§7). Let $\langle \mathbb{T}, \bowtie_\tau, {}_\tau\bowtie \rangle$ be the structure that contains all (infinitely many) left and right bowties then its first-order theory is decidable. Furthermore, the complexity is $\mathsf{STA}(*, 2^{O(n)}, n)$-complete.

The scaling permission requires the use of both $\oplus$ and $\bowtie$ and thus their combined theory is worth investigating. As the substructure $\langle \mathbb{T}, \bowtie \rangle$ is already undecidable, the combined theory of $\langle \mathbb{T}, \oplus, \bowtie \rangle$ therefore is also undecidable. Fortunately, we discovered a decidable fragment in which $\bowtie$ is restricted to right-bowties $\bowtie_\tau$.

Contribution 8 (§7). Let $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot}, \bowtie_\tau \rangle$ be the combined structure that contains all right-bowties then its first-order theory is decidable but its complexity is non-elementary*.

1.3 Structure of the thesis

This thesis is organized as follows:

1. In chapter 2, we provide the formal definition of the tree share structure together with necessary background in separation logic and program verification.

*it is not bounded by any exponential time class $n\mathsf{EXP}$.

2. In chapter 3, we propose a general modal logic framework using tree shares that is capable of reasoning about sophisticated verification tasks such as doing induction over the finiteness of the heap within the object logic or carrying out bi-abductive inference.

3. In chapter 4, we report our results on two decision procedures SAT and IMP to solve satisfiability and entailment problem over tree shares.

4. In chapter 5, we report our results on two certified procedures GSAT and GIMP that can additionally handle tree share disequations with improved performance.

5. In chapter 6, we establish the precise complexity result for the structure $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot} \rangle$.

6. In chapter 7, we prove the connection between bowtie and string concatenation. One of the consequences is that first-order theory of $\langle \mathbb{T}, \bowtie \rangle$ is undecidable. We recover the first-order decidability of $\bowtie$ by restricting constants on the left (${}_{\tau}\bowtie$) or right ($\bowtie_{\tau}$). Consequently, we are able to prove the first-order complexity of two decidable fragments $\langle \mathbb{T}, {}_{\tau}\bowtie, \bowtie_{\tau} \rangle$ and $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot}, \bowtie_{\tau} \rangle$.

7. In chapter 8, we draw our conclusion about the thesis and discuss several directions for future work.

Chapter 2

Preliminaries and notations

> "*You see there is only one constant. One universal. It is the only real truth. Causality. Action, reaction. Cause and effect.*"
>
> Merovingian, *Matrix Reloaded* (2003).

In this chapter, we will discuss the essential related work of the thesis. In particular, we will provide some common knowledge and constructions about separation logic, its precursor Hoare logic, and its successor concurrent separation logic. From this foundation, we will further explain why and how permissions are used in program verification.

This chapter consists of three following sections:

1. § 2.1 includes basic definitions and notations in logic that will be widely used throughout the thesis.
2. § 2.2 contains formal definitions of the tree share structure together with common notations that will be used throughout the thesis.
3. § 2.3 provides some background over separation logic and its formal constructions.
4. § 2.4 conveys information about permission models in program verification.

2.1 Basic definitions and notations

2.1.1 Language and structure

Language. A *signature* is a triple $\sigma = (F, P, \mathsf{arity})$ in which:

1. $F = \{f_1, \ldots, f_n\}$ is the set of function symbols.
2. $P = \{Q_1, \ldots, Q_m\}$ is the set of predicate symbols.
3. $\mathsf{arity} : F \cup P \mapsto \mathbb{N}$ is the arity function that specifies the number of arguments for functions and predicates. Notice that constants are considered as nullary-function.

We will usually represent a signature as a k-tuple $(g_1^{a_1}, \ldots, g_k^{a_k})$ in which g_i is either a function or a predicate symbol and a_i is its arity. We will make sure that the symbol's type (function or predicate) is made clear to the readers. If $a_i = 0$, *i.e.* f_i is a constant, then we will simply write g_i instead of g_i^0. If the arity of a symbol is implicitly known, we will omit it for convenience.

Example 2.1.1. $(+^2, \times^2, S^1, <^2, 0)$ is the signature of Peano arithmetic in which S is the successor function, namely $S(n) = n + 1$. ◁

Next we show in detail how σ-formulas are constructed from the signature σ. Let $V = \{v_1, v_2, \ldots\}$ be the set of variables. Then a *σ-term* is either a variable v, a constant c or of the form $f_k(t_1, \ldots, t_k)$ in which $\{t_i\}_{i=1}^k$ are σ-terms and f is a k-ary function:

$$\mathsf{term} \quad \stackrel{\text{def}}{=} \quad v \;\mid\; c \;\mid\; f(\mathsf{term}_1, \ldots, \mathsf{term}_k).$$

An *atomic σ-formula* is either the equality between two σ-terms $\mathsf{term}_1 = \mathsf{term}_2$ or a predicate consists of k σ-terms $Q(\mathsf{term}_1, \ldots, \mathsf{term}_n)$ in which Q is a k-ary predicate:

$$\mathsf{Atomic} \quad \stackrel{\text{def}}{=} \quad \mathsf{term}_1 \;=\; \mathsf{term}_2 \;\mid\; Q(\mathsf{term}_1, \ldots, \mathsf{term}_n).$$

A *first-order σ-formula* is an element of the closure of atomic σ-formulas under logical

connectives $\{\wedge, \vee, \rightarrow, \neg\}$ and quantifiers $\{\forall, \exists\}$:

$$\Phi \quad \stackrel{\text{def}}{=} \quad \mathsf{Atomic} \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \rightarrow \Phi_2 \mid \forall v.\ \Phi \mid \exists v.\ \Phi.$$

For convenience, if the signature σ is implicitly known, we will omit σ prefix in all related terms.

Theory. A variable instance v in Φ is *bound* if it is within the scope of some quantifier $\forall v$ or $\exists v$ and *free* otherwise. A σ-formula Φ is a *sentence* if it does not contain any free variables.

Example 2.1.2. Let $\Phi \stackrel{\text{def}}{=} v = 0 \vee \exists v.\ v = S(0)$ be a formula in $(+, \times, S, <, 0)$ then (from left to right) the first v instance is free while the second v instance is bounded. Consequently, Φ is not a sentence. On the other hand, $\Phi' \stackrel{\text{def}}{=} \forall x \exists y. x < y$ is a sentence. ◁

A *σ-theory* is a set of σ-sentences. A σ-theory T is *complete* if for each sentence Φ, either Φ or $\neg\Phi$ is in T. On the other hand, T is *decidable* if membership testing in T is decidable, *i.e.*, there is a halting Turing machine that can check whether an arbitrary sentence Φ is in T. It is worth noting that in the context of a theory, completeness and decidability are not equivalent.

Example 2.1.3. Let T_1 the the set of all valid sentences about natural numbers. Then T_1 is complete but not decidable (by Gödel Incompleteness Theorem [Göd29]). In contrast, if $T_2 = \emptyset$ then T_2 is decidable but not complete for any signature σ. ◁

Formula hierarchy. A formula Φ is quantifier-free if it does not contain any quantifier. Furthermore, we let $\Sigma_0 = \Pi_0$ be the sets of all quantifier-free formulas. Let Σ_1 be the set of existential formulas and Π_1 be the set of universal formulas, *i.e.*:

- $\Sigma_1 \stackrel{\text{def}}{=} \{\exists v_1 \ldots \exists v_n.\ \Phi \mid \Phi \text{ is quantifier-free}\}$.
- $\Pi_1 \stackrel{\text{def}}{=} \{\forall v_1 \ldots \forall v_n.\ \Phi \mid \Phi \text{ is quantifier-free}\}$.

Generally, Σ_{i+1} is the set of formulas $\exists v_1 \ldots \exists v_n.\Phi$ for $\Phi \in \Pi_i$ and Π_{i+1} is the set of formulas $\forall v_1 \ldots \forall v_n.\Phi$ for $\Phi \in \Sigma_i$:

- $\Sigma_{i+1} \stackrel{\text{def}}{=} \{\exists v_1 \ldots \exists v_n.\ \Phi \mid \Phi \in \Pi_i\}$.
- $\Pi_{i+1} \stackrel{\text{def}}{=} \{\forall v_1 \ldots \forall v_n.\ \Phi \mid \Phi \in \Sigma_i\}$.

In short, Σ_n/Π_n contains $n-1$ alternations between $\exists$ and $\forall$ in which the outermost quantifiers are existential/universal. A formula Φ is in *Prenex normal form* if $\Phi \in \Sigma_n \cup \Pi_n$ for some n. Furthermore, every formula is equivalent to a Prenex formula [Sri13].

Structure. A *σ-structure* is an interpretation of the symbols in the signature σ. Formally, a σ-structure is the triple $\mathcal{A} = \langle \mathcal{U}, \mathcal{F}, \mathcal{P} \rangle$ such that:

1. $\mathcal{U}$ is the *universe of discourse*.
2. For each k-ary function symbol $f \in F$, there is a corresponding k-ary function $f^{\mathcal{A}} : \mathcal{U}^k \mapsto \mathcal{U}$ in $\mathcal{F}$.
3. For each k-ary predicate symbol $Q \in P$, there is a corresponding k-ary predicate $Q^{\mathcal{A}} \subseteq \mathcal{U}^k$ in $\mathcal{P}$.

For simplicity, we will usually write a structure as $\langle \mathcal{U}, g_1, \ldots, g_n \rangle$ in which $\mathcal{U}$ is the universe and each g_i is either a function or a predicate.

Semantics. An *interpretation* $\mathcal{I} : \mathcal{V} \mapsto \mathcal{U}$ is a mapping from variables to values in $\mathcal{U}$. In addition, let $\mathcal{I}[v \Leftarrow a]$ be the *overriding interpretation* of $\mathcal{I}$ at $v \in \mathcal{V}$ by $a \in \mathcal{U}$, *i.e.*:

$$\mathcal{I}[v \Leftarrow a] \stackrel{\text{def}}{=} \lambda v'.\ \text{if } v' = v \text{ then } a \text{ else } \mathcal{I}(v').$$

For a term $\mathtt{t}$, we override $\mathcal{I}(\mathtt{t})$ to be the evaluation of $\mathtt{t}$, *i.e.*:

1. $\mathcal{I}(c) \stackrel{\text{def}}{=} c^{\mathcal{A}}$, if c is constant.
2. $\mathcal{I}(f(\mathtt{t}_1, \ldots, \mathtt{t}_n)) \stackrel{\text{def}}{=} f^{\mathcal{A}}(\mathcal{I}(\mathtt{t}_1), \ldots, \mathcal{I}(\mathtt{t}_n))$.

The structure $\mathcal{A}$ satisfies a formula Φ under interpretation $\mathcal{I}$, denoted by $(\mathcal{A}, \mathcal{I}) \models \Phi$, if Φ is true under the evaluation of $\mathcal{A}$ and $\mathcal{I}$:

1. $(\mathcal{A}, \mathcal{I}) \models Q(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ iff $Q^{\mathcal{A}}(\mathcal{I}(\mathtt{t}_1), \ldots, \mathcal{I}(\mathtt{t}_n)) \in \mathcal{P}$.
2. $(\mathcal{A}, \mathcal{I}) \models \mathtt{t}_1 = \mathtt{t}_2$ iff $\mathcal{I}(\mathtt{t}_1) = \mathcal{I}(\mathtt{t}_2)$.
3. $(\mathcal{A}, \mathcal{I}) \models \neg\Phi'$ iff $(\mathcal{A}, \mathcal{I}) \not\models \Phi'$.
4. $(\mathcal{A}, \mathcal{I}) \models \Phi_1 \wedge \Phi_2$ iff $(\mathcal{A}, \mathcal{I}) \models \Phi_1$ and $(\mathcal{A}, \mathcal{I}) \models \Phi_2$

5. $(\mathcal{A}, \mathcal{I}) \models \Phi_1 \vee \Phi_2$ iff $(\mathcal{A}, \mathcal{I}) \models \Phi_1$ or $(\mathcal{A}, \mathcal{I}) \models \Phi_2$.
6. $(\mathcal{A}, \mathcal{I}) \models \Phi_1 \rightarrow \Phi_2$ iff $(\mathcal{A}, \mathcal{I}) \models \neg\Phi_1 \vee \Phi_2$.
7. $(\mathcal{A}, \mathcal{I}) \models \forall v.\Phi'$ iff $(\mathcal{A}, \mathcal{I}[v \Leftarrow a]) \models \Phi'$ for every $a \in \mathcal{U}$.
8. $(\mathcal{A}, \mathcal{I}) \models \exists v.\Phi'$ iff $(\mathcal{A}, \mathcal{I}[v \Leftarrow a]) \models \Phi'$ for some $a \in \mathcal{U}$.

If Φ is a sentence (*i.e.* without free variables) then the evaluation $(\mathcal{A}, \mathcal{I}) \models \Phi$ is independent of $\mathcal{I}$. As a result, we will simply write $\mathcal{A} \models \Phi$.

Model. The *first-order theory of* $\mathcal{A}$, denoted by $\mathsf{Th}(\mathcal{A})$, is the set of sentences that are satisfied by $\mathcal{A}$, *i.e.*:

$$\mathsf{Th}(\mathcal{A}) \stackrel{\text{def}}{=} \{\Phi \mid \Phi \text{ is a sentence and } \mathcal{A} \models \Phi\}.$$

Let $A = \{\Psi_1, \Psi_2, \ldots\}$ be a set of sentences called *axioms*. A structure $\mathcal{A}$ is a *model* of A, denoted by $\mathcal{A} \models A$, if it satisfies all sentences in A. The *first-order theory of* A is the set of sentences that are satisfied by all models of A:

$$\mathsf{Th}(A) \stackrel{\text{def}}{=} \{\Phi \mid \text{if } \mathcal{A} \models A \text{ then } \mathcal{A} \models \Phi\}.$$

An alternative definition for $\mathsf{Th}(A)$ is by provability. We say Φ is provable from A (or A proves Φ), denoted by $A \vdash \Phi$, if there exists a natural deduction proof for Φ from axioms of A. Hence $\mathsf{Th}(A)$ is the set of all provable sentences from A. By Gödel's Completeness Theorem [Sri13] which states $A \vdash \Phi$ iff $A \models \Phi$, we know two definitions are equivalent.

Two σ-structures $\mathcal{A}_1$ and $\mathcal{A}_2$ are *elementarily equivalent* if they satisfy the same set of first-order σ-sentences, *i.e.*, $\mathsf{Th}(\mathcal{A}_1) = \mathsf{Th}(\mathcal{A}_2)$.

Conventions. For convenience, we will usually overload a function (predicate) with its symbol, *i.e.*, f represents both the function symbol in σ and the function $f^{\mathcal{A}}$ in $\mathcal{A}$. As a result, we will mention structures without introducing their signatures as such signatures can be derived from the structures themselves. For the purpose of this thesis, **the universe of the structure is a part of the signature**, *i.e.*, it is also the set of constant symbols

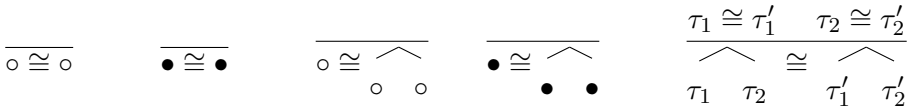


Figure 2.1: Canonical representation of tree shares

in the signature unless we say otherwise. Also, we may reuse some notations in different domains as long as there is no ambiguity.

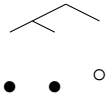
2.2 Tree share structure

We first provide the formal definition of the tree share structure in §2.2.1. Then in §2.2.2 we proceed to introduce some common notations associated with the tree share structure that we will use throughout the thesis.

2.2.1 Tree share domain and basic operators

Here we summarize some formal details of tree shares together with their associated properties as proposed by Dockins *et al.* [DHA09].

Canonical forms. A tree share is either $\bullet$, $\circ$ or $\mathsf{Node}(\tau_1, \tau_2)$ in which τ_1, τ_2 are tree shares and Node is a binary function. To make the representation visual, we will refer $\mathsf{Node}(\tau_1, \tau_2)$ as  . Additionally, we require tree shares are in *canonical form*, *i.e.*, it is in its most compact representation under the inductively-defined equivalence relation $\cong$ (Figure 2.1).

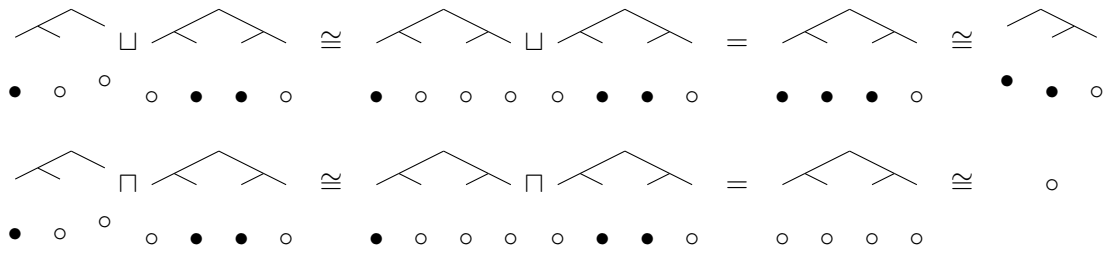
Example 2.2.1.  is not canonical whereas  is canonical. ◁

As we will see, operations on tree shares sometimes need to fold/unfold trees to/from canonical form, a practice we will indicate using the symbol $\cong$. Canonicality is needed to guarantee some of the algebraic properties of tree shares; managing it requires a little care in the proofs but does not pose any fundamental difficulties.

Tree boolean operators. The connectives $\sqcup$ and $\sqcap$ first unfold both trees to the same

shape; then calculate leafwise using the rules $\circ \sqcup \tau = \tau \sqcup \circ = \tau$, $\bullet \sqcup \tau = \tau \sqcup \bullet = \bullet$, $\circ \sqcap \tau = \tau \sqcap \circ = \circ$, and $\bullet \sqcap \tau = \tau \sqcap \bullet = \tau$; and finally refold into canonical form.

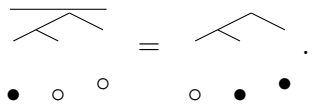
Example 2.2.2.



◁

To complement a tree, we simply flip leaves between $\circ$ and $\bullet$, which does not affect canonical form.

Example 2.2.3.



◁

Using these definitions we get all of the usual properties for Boolean Algebras (BAs), *e.g.* $\overline{\tau_1 \sqcap \tau_2} = \overline{\tau_1} \sqcup \overline{\tau_2}$. The complete set of their properties is in Figure 2.2.

Identity :	$a \sqcup \circ = a$	$a \sqcap \bullet = a$	(2.1)
Null :	$a \sqcup \bullet = \bullet$	$a \sqcap \circ = \circ$	(2.2)
Idempotency :	$a \sqcup a = a$	$a \sqcap a = a$	(2.3)
Involution :	$\bar{\bar{a}} = a$		(2.4)
Complementary :	$a \sqcup \bar{a} = \bullet$	$a \sqcap \bar{a} = \circ$	(2.5)
Commutativity :	$a \sqcup b = b \sqcup a$	$a \sqcap b = b \sqcap a$	(2.6)
Associativity :	$(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$	$(a \sqcap b) \sqcap c = a \sqcap (b \sqcap c)$	(2.7)
Distributivity :	$(a \sqcap b) \sqcup c = (a \sqcup c) \sqcap (b \sqcup c)$	$(a \sqcap b) \sqcup c = (a \sqcup c) \sqcap (b \sqcup c)$	(2.8)

Figure 2.2: BA axioms

The partial function $\oplus$ is defined in term of $\sqcup$ and $\sqcap$. In short, two tree shares are *joinable*

if their intersection is $\circ$ and the resulting share is their union:

$$a \oplus b = c \quad \stackrel{\text{def}}{=} \quad a \sqcap b = \circ \wedge a \sqcup b = c. \tag{2.9}$$

In other words, the join relation is a kind of disjoint union; it is partial because *e.g.* $\bullet \oplus \bullet$ is undefined. One critical property of $\oplus$ that we would like to highlight is the disjointness axiom that distinguishes the tree shares from rationals:

$$\forall x, y.\ x \oplus x = y \ \rightarrow \ x = y.$$

Using other properties of tree shares in Figure 2.3, we can prove a stronger version in which x, y must be identity $\circ$:

$$\forall x, y.\ x \oplus x = y \ \rightarrow \ x = y = \circ.$$

Proof. Let $x \oplus x = y$ then $x = y$ and thus $x \oplus x = x$. On the other hand, $\circ \oplus x = x$ and by cancellation rule in Fig. 2.3, we conclude that $x = \circ$. □

Hence the axiom says that the only element that can be joined with itself is the identity $\circ$. In contrast, rational model $\mathcal{Q} = \langle [0, 1], + \rangle$ does not admit this axiom, *e.g.*, $0.3 + 0.3 = 0.6$ but $0.3 \neq 0.6$. Using $\oplus$ we require the following relationship between the spatial conjunction $*$ and fractional mapping, namely one can split the permission $\pi_1 \oplus \pi_2$ of a fractional mapping into two sub-permissions π_1 and π_2:

$$x \overset{\pi_1}{\mapsto} y \ * \ x \overset{\pi_2}{\mapsto} z \ \dashv\vdash \ y = z \wedge x \overset{\pi_1 \oplus \pi_2}{\longmapsto} y. \tag{2.10}$$

Example 2.2.4. As $\overset{\frown}{\circ\ \bullet} \oplus \overset{\frown}{\bullet\ \circ} = \overset{\frown}{\bullet}$, the following bi-entailment holds:

$$x \overset{\bullet}{\mapsto} v \quad \dashv\vdash \quad x \overset{\frown}{\longmapsto} v \ * \ x \overset{\frown}{\longmapsto} v.$$

◁

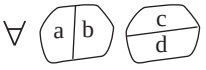
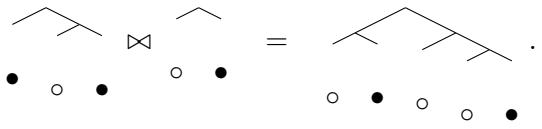
Functional :	$x \oplus y = z_1 \to x \oplus y = z_2 \to z_1 = z_2$	(2.11)
Commutativity :	$x \oplus y = y \oplus x$	(2.12)
Cancellation :	$x_1 \oplus y = z \to x_2 \oplus y = z \to x_1 = x_2$	(2.13)
Unit :	$\exists u.\ \forall x.\ x \oplus u = x$	(2.14)
Disjointness :	$x \oplus x = y \to x = y$	(2.15)
Cross split :	$a \oplus b = z \wedge c \oplus d = z \to \exists ac, ad, bc, bd.$	
	$ac \oplus ad = a \wedge bc \oplus bd = b \wedge ac \oplus bc = c \wedge ad \oplus bd = d$	(2.16)
	$\forall$  $\exists$ 	
Infinite Splitability :	$x \neq \circ \to \exists x_1, x_2.\ x_1 \neq \circ \wedge x_2 \neq \circ \wedge x_1 \oplus x_2 = x$	(2.17)

Figure 2.3: Properties of $\oplus$ which follow from BA axioms in Figure 2.2

Properties of tree multiplication $\bowtie$. In addition to $\oplus$, Dockins *et al.* also invented another operator $\bowtie$ called "bowtie" which is analogous to rational multiplication. Given two tree shares τ_1, τ_2, we compute $\tau_1 \bowtie \tau_2$ by replacing each black leaf $\bullet$ in τ_1 with an instance of τ_2, which bears a resemblance to the string replacement operator.

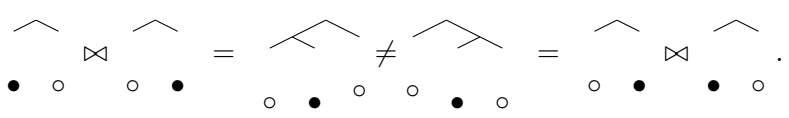
Example 2.2.5.



◁

To summarize, bowtie is an injective cancellative monoid with addition properties (Figure 2.4) in which $\bullet$ is the identity element (for comparison, $\circ$ is the identity of $\oplus$). Not like multiplication, bowtie is not commutative (Example 2.2.6) although it is left-distributive over $\sqcup$ and $\sqcap$.

Example 2.2.6.



◁

Associativity :	$\tau_1 \bowtie (\tau_2 \bowtie \tau_3) = (\tau_1 \bowtie \tau_2) \bowtie \tau_3$	(2.18)
Identity element :	$\tau \bowtie \bullet = \bullet \bowtie \tau = \tau$	(2.19)
Zero element :	$\tau \bowtie \circ = \circ \bowtie \tau = \circ$	(2.20)
Left cancellation :	$\tau \neq \circ \ \rightarrow \ \tau \bowtie \tau_1 = \tau \bowtie \tau_2 \ \rightarrow \ \tau_1 = \tau_2$	(2.21)
Right cancellation :	$\tau \neq \circ \ \rightarrow \ \tau_1 \bowtie \tau = \tau_2 \bowtie \tau \ \rightarrow \ \tau_1 = \tau_2$	(2.22)
Left distributivity over $\sqcap$:	$\tau_1 \bowtie (\tau_2 \sqcap \tau_3) = (\tau_1 \bowtie \tau_2) \sqcap (\tau_1 \bowtie \tau_3)$	(2.23)
Left distributivity over $\sqcup$:	$\tau_1 \bowtie (\tau_2 \sqcup \tau_3) = (\tau_1 \bowtie \tau_2) \sqcup (\tau_1 \bowtie \tau_3)$	(2.24)

Figure 2.4: Properties of $\bowtie$

From the two left distributive properties over $\sqcup, \sqcap$, we can directly derive the left distributive property for $\oplus$:

$$\forall \tau, \tau_1, \tau_2.\ \tau \ \bowtie \ (\tau_1 \oplus \tau_2) \ = \ (\tau \ \bowtie \ \tau_1) \ \oplus \ (\tau \ \bowtie \ \tau_2) \tag{2.25}$$

Proof. Suppose $\tau_1 \oplus \tau_2 = \tau_3$ then $\tau_1 \sqcup \tau_2 = \tau_3$ and $\tau_1 \sqcap \tau_2 = \circ$. By the left distributivity rules, we have:

1. $(\tau \bowtie \tau_1) \sqcup (\tau \bowtie \tau_2) = \tau \bowtie (\tau_1 \sqcup \tau_2) = \tau \bowtie \tau_3$, and
2. $(\tau \bowtie \tau_1) \sqcap (\tau \bowtie \tau_2) = \tau \bowtie (\tau_1 \sqcap \tau_2) = \tau \bowtie \circ = \circ$.

Hence $(\tau \bowtie \tau_1) \oplus (\tau \bowtie \tau_2) = \tau \bowtie \tau_3$. □

In the original paper [DHA09], this operator is mainly used to split a tree τ into two trees $\tau = \tau_l \oplus \tau_r$ s.t. $\tau_l = \tau \bowtie \overset{\frown}{\circ \ \bullet}$ and $\tau_r = \tau \bowtie \overset{\frown}{\bullet \ \circ}$ (by Prop. 2.25). Although it was used in metatheory [ADH+14], no decision procedure over bowtie has been developed due to its complexity and the absence of theoretical foundation.

2.2.2 Tree share notations

Here we introduce some standard definitions and notations for tree shares besides those definitions provided in Chapter 1. Let $\mathbb{T}$ be the tree share domain. Then the height of a tree

$\tau \in \mathbb{T}$, denoted by $|\tau|$, is the length of the longest path from its root to leaves:

$$|\bullet| \;=\; |\circ| \stackrel{\text{def}}{=} 0 \qquad |\overset{\tau_1}{\bullet}\quad\overset{\tau_2}{\circ}| \stackrel{\text{def}}{=} \max(|\tau_1|, |\tau_2|) + 1.$$

Generally, $|\Phi|$ is the height of the formula Φ, *i.e.*, the height of the highest tree in Φ:

$$|\Phi| \stackrel{\text{def}}{=} \max\{|\tau| \mid \tau \in \Phi\}.$$

Example 2.2.7. Here we have several examples about tree height. If a formula does not contain any tree then its height is zero.

1. $|\bullet\ \circ| = 1$ and $|\bullet\ \circ\ \bullet| = 2$.
2. $|\exists v.\ v = \bullet| = |\forall x \forall y \exists z.\ x \oplus y = z| = 0$ and $|\exists x.\ x = \bullet \vee x = \circ\ \bullet\ \circ| = 2$.

◁

Next, we define the *split function* that splits a tree τ into its left and right subtree:

$$\mathsf{Split}(\bullet) \stackrel{\text{def}}{=} (\bullet, \bullet) \qquad \mathsf{Split}(\circ) \stackrel{\text{def}}{=} (\circ, \circ) \qquad \mathsf{Split}(\overset{\tau_l}{\bullet}\quad\overset{\tau_r}{\circ}) \stackrel{\text{def}}{=} (\tau_l, \tau_r).$$

Example 2.2.8.

$$\mathsf{Split}(\bullet\ \circ\ \bullet) \;=\; (\bullet\ \circ, \bullet).$$

◁

The function Split is useful when reasoning about properties of recursive functions defined over tree shares, *e.g.*, properties about height, $\sqcup$ and $\sqcap$. More precisely, many properties over tree shares can be defined in term of their left and right subtrees, and so we can make use of Split for a systematic reasoning by induction.

Lemma 2.2.1 (Properties of Split)**.** The function Split satisfies the following properties:

1. If $|\tau| > 0$ and $\mathsf{Split}(\tau) = (\tau_1, \tau_2)$ then $|\tau_1| < |\tau|$ and $|\tau_2| < |\tau|$.

2. If $|\tau| = 0$ then $\mathsf{Split}(\tau) = (\tau, \tau)$.

3. Split is a bijection from $\mathbb{T}$ to $\mathbb{T}^2$.

4. Let $\mathsf{Split}(\tau_i) = (\tau_i^l, \tau_i^r)$ for $i = 1, 2, 3$ and $\star \in \{\sqcup, \sqcap, \oplus\}$ then:

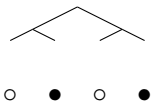
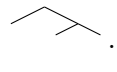
$$\tau_1 \star \tau_2 = \tau_3 \qquad \text{iff} \qquad \tau_1^l \star \tau_2^l = \tau_3^l \ \wedge \ \tau_1^r \star \tau_2^r = \tau_3^r.$$

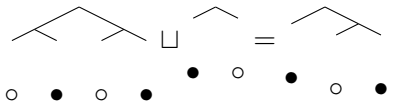
◁

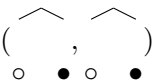
Proof. 1 and 2. Follow directly from the definition of Split and height.

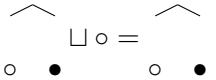
3. We need to show Split is both injective and surjective which is done by strong induction on $n = \max(|\tau_1|, |\tau_2|)$.

4. It suffices to prove the case $\star = \sqcup$ as the other two are similar. Again, this can be done by strong induction on $n = \max(|\tau_1|, |\tau_2|, |\tau_3|)$. □

Example 2.2.9. Let $\tau_1 =$  , $\tau_2 =$  and $\tau_3 =$  . Then:

1. $\tau_1 \sqcup \tau_2 =$  $= \tau_3$.

2. $\mathsf{Split}(\tau_1) = ($  $, \mathsf{Split}(\tau_2) = (\bullet, \circ)$, $\mathsf{Split}(\tau_3) = (\bullet,$  $)$.

3. $\tau_1^l \sqcup \tau_2^l =$  $= \tau_3^l$ and $\tau_1^r \sqcup \tau_2^r =$  $= \tau_3^r$.

◁

Lemma 2.2.2 (Proof framework)**.** We propose a general and systematic framework to prove properties over tree shares using Split. Suppose we need to prove

$$\forall \tau_1 \ldots \forall \tau_n.\ P(\tau_1, \ldots, \tau_n)$$

over n tree share variables $\{\tau_i\}_{i=1}^n$. It suffices to prove two cases:

C_1. $P(\tau_1, \ldots, \tau_n)$ holds for all $(\tau_1, \ldots, \tau_n) \in \{\bullet, \circ\}^n$.

C_2. $P(\tau_1^l, \ldots, \tau_m^l) \wedge P(\tau_1^r, \ldots, \tau_m^r) \Rightarrow P(\tau_1, \ldots, \tau_m)$, where $\mathsf{Split}(\tau_i) = (\tau_i^l, \tau_i^r)$ for $i = 1 \ldots m$.

$\triangleleft$

Proof. We show that if the two properties hold then one can prove $\forall \tau_1 \ldots \forall \tau_m.\ P(\tau_1, \ldots, \tau_m)$ by strong induction over $n = \max(|\tau_1|, \ldots, |\tau_m|)$. The base case $n = 0$ is clear from C_1. Suppose P holds for all $n \leq k$ and we want to prove P also holds for $n = k + 1 > 0$. Using C_2, it suffices to prove

$$P(\tau_1^l, \ldots, \tau_m^l) \quad \text{and} \quad P(\tau_1^r, \ldots, \tau_m^r).$$

Let $n_1 = \max(|\tau_1^l|, \ldots, |\tau_m^l|)$ and $n_2 = \max(|\tau_1^r|, \ldots, |\tau_m^r|)$. By Lemma 2.2.1, we deduct that $n_1 \leq k$ and $n_2 \leq k$. Hence the result follows from the induction hypothesis. $\square$

2.3 Separation logic

In §2.3.1, we will discuss about Hoare logic which is the precursor of Separation logic. We then make a transition to Separation logic in §2.3.2 and finally discuss its extension, Concurrent Separation logic, in §2.3.3.

2.3.1 Hoare logic

Hoare rules. Hoare logic is a formal logic framework developed by Floyd [Flo67] and Hoare [Hoa69] for program verification. To make it simple, we limit our discussion to the toy language $\mathcal{L}_1$ in Figure 2.5.

$$
\begin{array}{lcl}
e & \stackrel{\text{def}}{=} & \ldots, -1, 0, 1, \ldots \mid v \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 \ \mathsf{div}\ e_2 \mid e_1 \ \mathsf{mod}\ e_2 \\
b & \stackrel{\text{def}}{=} & e_1 = e_2 \mid e_1 < e_2 \mid \texttt{NOT}\ b \mid b_1\ \texttt{OR}\ b_2 \mid b_1\ \texttt{AND}\ b_2 \\
\mathsf{c} & \stackrel{\text{def}}{=} & \textbf{skip} \mid x := e \mid \textbf{if}\ b\ \textbf{then}\ \mathsf{c}_1\ \textbf{else}\ \mathsf{c}_2 \mid \textbf{while}\ b\ \textbf{do}\ \mathsf{c} \mid \mathsf{c}_1\ ;\ \mathsf{c}_2
\end{array}
$$

Figure 2.5: A simple language $\mathcal{L}_1$

Here e is the arithmetic expression, b is Boolean expression and c is the command. The **skip** command does nothing, $x := e$ assigns the value e to variable x, **if**...**else** and **while**...**do**

are for control flow. The following *assertion language* helps capture the semantics of these commands, which consists of standard comparisons between two expressions together with Boolean connectives (Figure 2.6).

$$P \stackrel{\text{def}}{=} \top \mid \bot \mid e = e \mid e < e \mid \neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid \forall x.\ P \mid \exists x.\ P.$$

Figure 2.6: Assertion language for Hoare logic

The key concept in Hoare logic is the *Hoare triple* $\{P\}\ \mathsf{c}\ \{Q\}$ in which P is the precondition, c is the executed code and P is the postcondition. Simply put, $\{P\}\ \mathsf{c}\ \{Q\}$ is interpreted as 'given the precondition P then executing the code c will result in the postcondition Q'. In practice, P and Q are *assertion predicates* capture the *program states* which carry information about variable assignments. Thus c can be viewed as the *transition action* that changes the state P into the state Q and the Hoare triple can be viewed as a compact and elegant way to describe such behaviors. The simplest Hoare rule is Skip which says the precondition and postcondition are the same for command **skip**:

$$\frac{}{\{P\}\ \mathbf{skip}\ \{P\}}\ \textsf{Skip}$$

A more complicated one is the rule for assignment:

$$\frac{}{\{P[e \Leftarrow x]\}\ x := e\ \{P\}}\ \textsf{Assign}$$

Here $P[e \Leftarrow x]$ represents the predicate P whose free variable x is replaced by expression e. Informally, the rule Assign says that given the precondition $P[e \Leftarrow x]$ in which x is replaced by e (possibly contains x) then the postcondition is simply P, *e.g.*:

$$\frac{}{\{x + 1 = 2\}\ x := x + 1\ \{x = 2\}}\ \textsf{Assign}$$

On the other hand, the rule Consequence gives us the flexibility to strengthen the precondition or weaken the postcondition:

$$\frac{P \Rightarrow P' \qquad \{P'\}\ \mathsf{c}\ \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\ \mathsf{c}\ \{Q\}}\ \textsf{Consequence}$$

$$\frac{\{P_1\}\ \mathsf{c}\ \{Q_1\} \qquad \{P_2\}\ \mathsf{c}\ \{Q_2\}}{\{P_1 \wedge P_2\}\ \mathsf{c}\ \{Q_1 \wedge Q_2\}}\ \textsf{Conjunction}$$

$$\frac{P \Rightarrow P' \qquad \{P'\}\ \mathsf{c}\ \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\ \mathsf{c}\ \{Q\}}\ \textsf{Consequence}$$

$$\frac{\{P\}\ \mathsf{c}_1\ \{Q\} \qquad \{Q\}\ \mathsf{c}_2\ \{R\}}{\{P\}\ \mathsf{c}_1; \mathsf{c}_2\ \{R\}}\ \textsf{Composition}$$

$$\frac{}{\{P\}\ \textbf{skip}\ \{P\}}\ \textsf{Skip}$$

$$\frac{}{\{P[x \Leftarrow e]\}\ x := e\ \{P\}}\ \textsf{Assign}$$

$$\frac{\{P \wedge b\}\ \mathsf{c}_1\ \{Q\} \qquad \{P \wedge \neg b\}\ \mathsf{c}_2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ \mathsf{c}_1\ \textbf{else}\ \mathsf{c}_2\ \{Q\}}\ \textsf{If}$$

$$\frac{\{I \wedge b\}\ \mathsf{c}\ \{I\} \qquad I \text{ is the loop invariant}}{\{I\}\ \textbf{while}\ b\ \textbf{do}\ \mathsf{c}\ \{I \wedge \neg b\}}\ \textsf{While}$$

Figure 2.7: Hoare rules

In addition, we have rules for control flow commands (**if** ... **else** and **while** ... **do**). The full rule set is listed in Figure 2.7. One highlight in the While rule is the introduction of the *loop invariant* I. The triple $\{I \wedge b\}\ \mathsf{c}\ \{I\}$ says that the predicate I remains unchanged under the effect of command c as long as Boolean expression b is still true. An important feature in Hoare logic is the Composition rule that makes the reasoning compositional:

$$\frac{\{P\}\ \mathsf{c}_1\ \{Q\} \qquad \{Q\}\ \mathsf{c}_2\ \{R\}}{\{P\}\ \mathsf{c}_1; \mathsf{c}_2\ \{R\}}\ \textsf{Composition}$$

Using Composition, the reasoning of the whole program is achieved in a command-by-command manner in which the postcondition of the previous command becomes the precondition of the next command. The following example demonstrates how several rules are used compositionally:

$$\dfrac{\dfrac{\dfrac{}{\{x=1\}\ \textbf{skip}\ \{x=1\}}\ \textsf{Skip} \qquad x=1 \Rightarrow x+1=2}{\{x=1\}\ \textbf{skip}\ \{x+1=2\}}\ \textsf{Consequence} \qquad \dfrac{}{\{x+1=2\}\ x := x+1\ \{x=2\}}\ \textsf{Assign}}{\{x=1\}\ \textbf{skip}; x := x+1\ \{x=2\}}\ \textsf{Composition}$$

Semantics. While it is nice to have Hoare rules for reasoning, it is even more important to justify their correctness. Consider the case when someone proposes a different Skip rule:

$$\frac{}{\{P\}\ \mathbf{skip}\ \{\bot\}}\ \mathsf{Better\_skip}$$

If we ignore its correctness, Better_skip is actually a very powerful rule. From the fact that anything can be proved from $\bot$, we can show that $\{P\}\ \mathbf{skip}; \mathsf{c}\ \{Q\}$ holds for any c, P, Q. Intuitively, we do not use this rule because it seems unreasonable. In other words, the rules do not make sense on their own but rather on the model that they reflect.

The standard model for Hoare rule makes use of a program state $\rho : \mathcal{V} \rightharpoonup \mathcal{D}$ which is a partial function from variables to (integer or boolean) values. Also it is conventional to call ρ alone a *stack*. A *configuration* is a pair (ρ, c) of program state ρ and command c, and a *step relation* $(\rho_1, \mathsf{c}_1) \rightsquigarrow (\rho_2, \mathsf{c}_2)$ is a binary relation of configurations. In short, $(\rho_1, \mathsf{c}_1) \rightsquigarrow (\rho_2, \mathsf{c}_2)$ indicates the change of state from ρ_1 to ρ_2 when the command is changed from c_1 to c_2. The main purpose of step relation is to formally describe the behavior of the command based on the underlying model. Similar to Hoare rules, step relations are described in the form of rule base with side conditions. For instance, the step relation for Assign is called SAssign:

$$\frac{[|e|]_\rho = v \qquad \rho' = \rho[x \Leftarrow v]}{(\rho, x := e; \mathsf{c}) \rightsquigarrow (\rho', \mathsf{c})}\ \mathsf{SAssign}$$

Here we use the syntactic sugar $[|e|]_\rho$ to indicate the evaluation of (arithmetic or boolean) expression e by program state ρ (which is simply an application of term evaluation described in §2.1.1). Furthermore $\rho[x \Leftarrow v]$ is the overriding state of ρ at x by value v. SAssign says that the state ρ after the assignment $x := e$ is updated at x by the evaluation of e. A complete description of the step relation is mentioned in Figure 2.8.

We are now ready to define the Hoare triple in term of small step relation in a *continuation* style proposed by Appel and Blazy [AB07] (Figure 2.9). The predicate $\mathsf{isSafe}(\Gamma)$ means the configuration Γ is *safe*, *i.e.*, either there exists another configuration Γ' that Γ can connect to using small step relation, or the configuration Γ contains an empty program. The condition

$$\frac{}{(\rho, \mathbf{skip}; \mathsf{c}) \rightsquigarrow (\rho, \mathsf{c})}\ \mathsf{SSkip}$$

$$\frac{[|e|]_\rho = v \qquad \rho' = \rho[x \Leftarrow v]}{(\rho, x := e; \mathsf{c}) \rightsquigarrow (\rho', \mathsf{c})}\ \mathsf{SAssign}$$

$$\frac{[|b|]_\rho = \top}{(\rho, \mathbf{if}\ b\ \mathbf{then}\ \mathsf{c}_1\ \mathbf{else}\ \mathsf{c}_2; \mathsf{c}) \rightsquigarrow (\rho, \mathsf{c}_1; \mathsf{c})}\ \mathsf{SIf}_1$$

$$\frac{[|b|]_\rho = \bot}{(\rho, \mathbf{if}\ b\ \mathbf{then}\ \mathsf{c}_1\ \mathbf{else}\ \mathsf{c}_2; \mathsf{c}) \rightsquigarrow (\rho, \mathsf{c}_2; \mathsf{c})}\ \mathsf{SIf}_2$$

$$\frac{[|b|]_\rho = \top}{(\rho, \mathbf{while}\ b\ \mathbf{do}\ \mathsf{c}_1; \mathsf{c}) \rightsquigarrow (\rho, \mathsf{c}_1; \mathbf{while}\ b\ \mathbf{do}\ \mathsf{c}_1; \mathsf{c})}\ \mathsf{SWhile}_1$$

$$\frac{[|b|]_\rho = \bot}{(\rho, \mathbf{while}\ b\ \mathbf{do}\ \mathsf{c}_1; \mathsf{c}) \rightsquigarrow (\rho, \mathsf{c})}\ \mathsf{SWhile}_2$$

Figure 2.8: Step relation for Hoare logic

$\mathsf{guarded}(P, \mathsf{c})$ asserts that command c is *guarded* by predicate P, *i.e.*, all state ρ satisfying P must be safe with respect to command c. Finally, $\{P\}\ \mathsf{c}\ \{Q\}$ means for all commands c' guarded by postcondition Q, its composition with c, *i.e.* $\mathsf{c}; \mathsf{c}'$, is therefore guarded by precondition P.

$$\begin{aligned}
\mathsf{isSafe}(\Gamma) &\stackrel{\text{def}}{=} \exists \Gamma'.\ \Gamma \rightsquigarrow \Gamma' \vee \exists \rho.\ \Gamma = (\rho, \emptyset) \\
\mathsf{guarded}(P, \mathsf{c}) &\stackrel{\text{def}}{=} \forall \sigma.\ \sigma \models P \rightarrow \mathsf{isSafe}(\sigma, \mathsf{c}) \\
\{P\}\ \mathsf{c}\ \{Q\} &\stackrel{\text{def}}{=} \forall \mathsf{c}'.\ \mathsf{guarded}(Q, \mathsf{c}') \rightarrow \mathsf{guarded}(P, \mathsf{c}; \mathsf{c}')
\end{aligned}$$

Figure 2.9: Semantics of Hoare triple

One may ask why we would prefer to reason axiomatically over the Hoare rules rather than directly over the step relations. The answer is quite simple: Hoare rules are usually significantly more compact and meaningful than the step relations in term of *capturing the high-level properties of the verified program.* In step relation, we need to keep track of the program state which may contain hundreds of variables and yet it does not reflect any significance about the program semantic. In contrast, the Hoare rules allow us to abstract away the program state while they are still capable of expressing the desirable properties of the program. Thus the proof rule approach yields a more efficient computation framework that can be deployed into automatic tools. On the other hand, the step relation serves as an

important ingredient for the soundness proof of the Hoare rules:

Proposition 2.3.1 ([Hoa69])**.** The set of Hoare rules in Figure 2.7 are sound with respect to the step relation in Figure 2.8, *i.e.*, any derivable Hoare triple is valid. ◁

2.3.2 Separation logic

The main disadvantage of Hoare logic is the lack of reasoning support over memory manipulation commands, *e.g.*, the function `malloc`() in C. To tackle this problem, Reynolds [Rey02] and O'Hearn [IO01] introduced *Separation logic* (SL) which is extended from Hoare logic with *shape analysis* to reason about memory. Its inception is inspired by the *pointer aliasing* problem:

$$\overline{\{x \mapsto 42 \wedge y \mapsto 42\}\ [x] := 1\ \{?\}}$$

Here we leave the postcondition with the question mark ? because the correct answer is not unique. Ideally, if x and y refer to two different addresses then the postcondition is easy to spot: $x \mapsto 1 \wedge y \mapsto 42$. However, x and y could be aliasing, *i.e.*, they refer to the same address. If that is the case then we have the postcondition $x \mapsto 1 \wedge y \mapsto 1$. One ugly solution is to enumerate all possible scenarios in the postcondition, *i.e.*:

$$(x \mapsto 1 \wedge y \mapsto 42 \wedge x \neq y) \vee (x \mapsto 1 \wedge y \mapsto 1 \wedge x = y).$$

However, this solution will compute an exponential size formula with respect to the number of pointers and thus is not practical for large programs. Even worse, checking pointer aliasing is undecidable as y can be replaced with arbitrary terms consisting of complicated functions and thus the undecidability follows immediately from Rice's theorem [Ric53].

An elegant solution to this problem is the introduction of *separating conjunction* $*$ that gave birth to SL, *e.g.*, $x \mapsto 42 * y \mapsto 42$ asserts x and y are disjoint memory addresses. Consequently, the postcondition can be identified as $x \mapsto 1 * y \mapsto 10$ as there is no confusion between x and y.

Language and rules. To make use of all beautiful features of SL, we extend our toy

$$
\begin{array}{lcl}
e & \stackrel{\text{def}}{=} & \ldots, -1, 0, 1, \ldots \mid v \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 \ \mathsf{div}\ e_2 \mid e_1 \ \mathsf{mod}\ e_2 \\
b & \stackrel{\text{def}}{=} & e_1 = e_2 \mid e_1 < e_2 \mid \texttt{NOT}\ b \mid b_1\ \texttt{OR}\ b_2 \mid b_1\ \texttt{AND}\ b_2 \\
\mathsf{c} & \stackrel{\text{def}}{=} & \textbf{skip} \mid x := e \mid \textbf{if}\ b\ \textbf{then}\ \mathsf{c}_1\ \textbf{else}\ \mathsf{c}_2 \mid \textbf{while}\ b\ \textbf{do}\ \mathsf{c} \mid \mathsf{c}_1\ ;\ \mathsf{c}_2 \mid \\
 & & [e_1] := e_2 \mid x := [e] \mid x := \texttt{new}(e) \mid \texttt{free}(e)
\end{array}
$$

Figure 2.10: A simple language $\mathcal{L}_2$ for SL

language $\mathcal{L}_1$ in Figure 2.5 to $\mathcal{L}_2$ in Figure 2.10 that contains four extra memory-related commands. The *store* command $[e_1] := e_2$ updates the value at address e_1 with e_2 while the *load* command $x := [e]$ assigns whatever value at address e to variable x. The *new* command $x := \texttt{new}(e)$ is used to assign x with a fresh address whose stored value is e and *free* command $\texttt{free}(e)$ is for the deallocation of address e.

On the other hand, the assertion language in Figure 2.6 is extended to contain emp, $\mapsto$ and $*$ for *heap* (memory) reasoning in Figure 2.11. Informally, emp represents the empty heap, $e_1 \mapsto e_2$ is the *maps-to* predicate of single address e_1 with value e_2, and $P_1 * P_2$ asserts that the heap can be split into two disjoint heaps that satisfy P_1 and P_2. For instance, $x \mapsto 1 * y \mapsto 1$ is satisfied by a heap that contains exactly two distinct addresses x and y whose stored values are both 1.

$$
\begin{array}{lcl}
P & \stackrel{\text{def}}{=} & \top \mid \bot \mid e = e \mid e < e \mid \neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid \forall x.\ P \mid \exists x.\ P \mid \\
 & & \mathsf{emp} \mid e_1 \mapsto e_2 \mid P_1 * P_2.
\end{array}
$$

Figure 2.11: Assertion language for separation logic

One important rule is the Frame rule for local reasoning:

$$
\frac{\{P\}\ \mathsf{c}\ \{Q\}}{\{F * P\}\ \mathsf{c}\ \{F * Q\}}\ \mathsf{Frame}
$$

In short, this rule states that the *frame* predicate F can be ignored when proving the postcondition Q if the precondition P is already sufficient. As a result, verification tools are able to concentrate on just the modified state of the machine rather than the complete

$$\frac{}{\{P[x \Leftarrow v] \wedge e \mapsto v\}\ x := [e]\ \{P\ \wedge\ e \mapsto v\}}\ \mathsf{Load}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{\{e_1 \mapsto \_\}\ [e_1] := e_2\ \{v_1 \mapsto v_2\}}\ \mathsf{Store}$$

$$\frac{e \Downarrow v}{\{\mathsf{emp}\}\ x := \mathtt{new}(e)\ \{x \mapsto v\}}\ \mathsf{New}$$

$$\frac{}{\{e \mapsto \_\}\ \mathtt{free}(e)\ \{\mathsf{emp}\}}\ \mathsf{Free}$$

Figure 2.12: Memory-related rules for Separation logic where $e \Downarrow v$ asserts v is the evaluation of the expression e.

state. For comparison, if $*$ is replaced by $\wedge$ then Frame becomes unsound as justified by the example at the start of this subsection:

$$\frac{\{P\}\ \mathsf{c}\ \{Q\}}{\{F \wedge P\}\ \mathsf{c}\ \{F \wedge Q\}}\ \mathsf{Unsound\_frame}$$

One beautiful feature of SL is that all Hoare rules in Figure 2.9 can be reused. Furthermore, we have four more rules for the additional memory commands in Figure 2.12. The Load rule is similar to Assign in Figure 2.7 except x is updated by the stored value v of address e. The Store rule says that the stored value of address e_1 is updated to e_2 after the execution of $[e_1] := e_2$. In the precondition, $e \Downarrow v$ indicates that the expression e is evaluated to value v. This is to handle the case where both e_1 and e_2 contain some common variables, *e.g.*, $[x] := \mathsf{new}(x + 1)$. Also we use the syntactic sugar $e_1 \mapsto \_$ to mean the address e_1 is already allocated, *i.e.*, $e_1 \mapsto \_ \stackrel{\text{def}}{=} \exists v.\ e_1 \mapsto v$. For New, the postcondition $x \mapsto v$ means the stored value at fresh address x is assigned to v. Finally, the Free rule allows us to free the address e.

Semantics. The model for SL is extended from the Hoare model by including two additional components to the program state, namely a heap $h : \mathsf{Addr} \rightharpoonup \mathsf{Val}$ which is a partial function from addresses to values, and a *break* $\mathsf{brk} \in \mathbb{N}$ that stores the maximal address used by the program. The sole purpose of brk is to implement the constructor Con for fresh address required by command `new()`, *i.e.*, $\mathsf{Con}(x)$ will increase the value of brk by one and assign

$$
\begin{array}{rcl}
(\rho, h, \mathsf{brk}) \models \mathsf{emp} & \stackrel{\text{def}}{=} & \mathsf{dom}(h) = \emptyset \\
(\rho, h, \mathsf{brk}) \models e_1 \mapsto e_2 & \stackrel{\text{def}}{=} & [|e_1|]_\rho = v_1 \ \wedge \ [|e_2|]_\rho = v_2 \ \wedge \\
 & & \mathsf{dom}(h) = \{v_1\} \ \wedge \ h(v_1) = v_2 \\
(\rho, h, \mathsf{brk}) \models P * Q & \stackrel{\text{def}}{=} & \exists h_1 \exists h_2.\ h_1 \uplus h_2 = h \ \wedge \\
 & & (\rho, h_1, \mathsf{brk}) \models P \ \wedge \ (\rho, h_2, \mathsf{brk}) \models Q
\end{array}
$$

Figure 2.13: Semantics of assertion language for SL

that value to x. As a result, a configuration Γ is a triple (ρ, h, brk) of stack ρ, heap h and break brk.

For a heap h, we denote its domain as $\mathsf{dom}(h)$ which is a subset of Addr. Two heaps h_1 and h_2 are *disjoint*, denoted by $h_1 \perp h_2$, if their domains are disjoint, *i.e.*, $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset$. The *joint heap* h of two disjoint heap h_1 and h_2, denoted by $h = h_1 \uplus h_2$, is the combined heap of h_1 and h_2, *i.e.*:

$$
h = h_1 \uplus h_2 \quad \stackrel{\text{def}}{=} \quad \begin{cases} \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset, \\ \mathsf{dom}(h) = \mathsf{dom}(h_1) \cup \mathsf{dom}(h_2), \\ h(a) = h_1(a) \text{ if } a \in \mathsf{dom}(h_1), \\ h(a) = h_2(a) \text{ if } a \in \mathsf{dom}(h_2). \end{cases}
$$

The semantics for heap-related predicates is formally defined in Figure 2.13. The emp predicate is satisfied by the empty heap. The map predicate $e_1 \mapsto e_2$ is satisfied by a single-cell heap at address e_1. The star predicate $P * Q$ says that the heap can be partitioned into two disjoint heaps that satisfy P and Q respectively.

The next ingredient for SL semantics is the formation of step relation to capture the behavior of each individual command. The step relation for basic commands in Figure 2.8 is reused with a small change of state representation: we replace the old state ρ with the new state $\kappa = (\rho, h, \mathsf{brk})$. As a result, the new configuration is a pair (κ, c) of program state κ and command c. In Figure 2.14, we describe the step relation for heap-related commands. The

$$\frac{\kappa = (\rho, h, \mathsf{brk}) \qquad \kappa' = (\rho', h, \mathsf{brk}) \qquad [\![e]\!]_\rho = v \qquad v \in \mathsf{dom}(h) \qquad \rho' = \rho[x \Leftarrow h(v)]}{(\kappa, x := [e]; \mathsf{c}) \rightsquigarrow (\kappa', \mathsf{c})}\ \mathsf{SLoad}$$

$$\frac{\kappa = (\rho, h, \mathsf{brk}) \qquad \kappa' = (\rho, h', \mathsf{brk}) \qquad [\![e_1]\!]_\rho = v_1 \qquad [\![e_2]\!]_\rho = v_2 \qquad v_1 \in \mathsf{dom}(h) \qquad h' = h[v_1 \Leftarrow v_2]}{(\kappa, [e_1] := e_2; \mathsf{c}) \rightsquigarrow (\kappa', \mathsf{c})}\ \mathsf{SStore}$$

$$\frac{\kappa = (\rho, h, \mathsf{brk}) \qquad \kappa' = (\rho', h', \mathsf{brk} + 1) \qquad [\![e]\!]_\rho = v \qquad \rho' = \rho[x \Leftarrow \mathsf{brk}] \qquad h' = h[\mathsf{brk} \Leftarrow v]}{(\kappa, x := \mathtt{new}(e); \mathsf{c}) \rightsquigarrow (\kappa', \mathsf{c})}\ \mathsf{SNew}$$

$$\frac{\kappa = (\rho, h, \mathsf{brk}) \qquad \kappa' = (\rho, h', \mathsf{brk}) \qquad [\![e]\!]_\rho = v \qquad h' = h[v \Leftarrow \mathsf{None}]}{(\kappa, \mathtt{free}(e); \mathsf{c}) \rightsquigarrow (\kappa', \mathsf{c})}\ \mathsf{SFree}$$

Figure 2.14: Semantics of small step relation for heap-related commands

step SLoad updates the stack variable x with the stored value at address e. The step SStore updates the stored value at the heap address e_1 with e_2. The step SNew updates the stack variable x with brk and heap address brk with stored value e. Furthermore, it also increases the value of the break brk by one. Lastly, the step SFree frees the address e in the heap. Here we use the syntactic sugar $h[v \Leftarrow \mathsf{None}]$ to indicate the address v in h is deallocated.

We are now ready to define the semantics of the Hoare SL triple $\{P\}\ \mathsf{c}\ \{Q\}$ (Figure 2.15) which is a modified version of the triple in Figure 2.9. The predicate $\mathsf{isSafe}(\Gamma)$ and $\mathsf{guarded}(P, \mathsf{c})$ remain unchanged. They say that the configuration Γ is safe and the command c is guarded by P respectively. The condition $\mathsf{closed}(F, \mathsf{c})$ basically asserts the independence between command c and predicate F, *i.e.*, c does not affect the validity of F. Finally, the SL triple $\{P\}\ \mathsf{c}\ \{Q\}$ is defined similarly to the Hoare triple in Figure 2.9 except it also reflects the presence of the frame predicate F. One may ask why we do not simply reuse the definition in Figure 2.15 but rather choose to make all these complicated changes. The answer is simple: these changes are essential to guarantee the correctness of the Frame rule.

Remark. Other than the separating conjunction $*$, we also have other separating connectives

$$
\begin{array}{rcl}
\mathsf{isSafe}(\Gamma) & \stackrel{\text{def}}{=} & \exists \Gamma'.\ \Gamma \rightsquigarrow \Gamma' \vee \exists \kappa.\ \Gamma = (\kappa, \emptyset) \\
\mathsf{guarded}(P, \mathsf{c}) & \stackrel{\text{def}}{=} & \forall \kappa\ \kappa \models P \rightarrow \mathsf{isSafe}(\kappa, \mathsf{c}) \\
\mathsf{closed}(F, \mathsf{c}) & \stackrel{\text{def}}{=} & \forall v.\ \mathsf{modified}(\mathsf{c}, v) \rightarrow \forall h \forall \rho \forall \mathsf{brk}.\ [(\rho, h, \mathsf{brk}) \models P \rightarrow \\
 & & \forall n.\ (\rho[v \Leftarrow n], h, \mathsf{brk}) \models F] \\
\{P\}\ \mathsf{c}\ \{Q\} & \stackrel{\text{def}}{=} & \forall \mathsf{c}' \forall F.\ \mathsf{closed}(F, \mathsf{c}) \rightarrow \mathsf{guarded}(F * Q, \mathsf{c}') \rightarrow \mathsf{guarded}(F * P, \mathsf{c}; \mathsf{c}')
\end{array}
$$

Figure 2.15: Semantics of Hoare SL triple

$$
\begin{array}{rcl}
(\rho, h, \mathsf{brk}) \models P \uplus Q & \stackrel{\text{def}}{=} & \exists h_1 \exists h_2 \exists h_3.\ h_1 \uplus h_2 \uplus h_3 = h\ \wedge \\
 & & (\rho, h_1 \uplus h_2, \mathsf{brk}) \models P\ \wedge\ (\rho, h_2 \uplus h_3, \mathsf{brk}) \models Q \\
(\rho, h, \mathsf{brk}) \models P \mathrel{-\!\!*} Q & \stackrel{\text{def}}{=} & \forall h_1.\ (h \perp h_1\ \wedge\ (\rho, h_1, \mathsf{brk}) \models P)\ \rightarrow\ (\rho, h \uplus h_1, \mathsf{brk}) \models Q \\
(\rho, h, \mathsf{brk}) \models P \mathrel{-\!\!\oplus} Q & \stackrel{\text{def}}{=} & \exists h_1.\ h_1 \perp h\ \wedge\ (\rho, h_1, \mathsf{brk}) \models P\ \wedge\ (\rho, h \uplus h_1, \mathsf{brk}) \models Q
\end{array}
$$

Figure 2.16: Semantics of separating connectives

such as the *overlapping* $\uplus$ and two *magic wands* $-\!\!*, -\!\!\oplus$:

$$P \quad \stackrel{\text{def}}{=} \quad \ldots \mid P_1 \uplus P_2 \mid P_1 \mathrel{-\!\!*} P_2 \mid P_1 \mathrel{-\!\!\oplus} P_2.$$

The semantics of these operators are formally defined in Figure 2.16. In particular, $P \uplus Q$ says that we can split the heap into two overlapping heaps, $h_1 \uplus h_2$ and $h_2 \uplus h_3$, that satisfy P and Q respectively. The *universal magic wand* $P \mathrel{-\!\!*} Q$ means that for any heap h_1 that satisfies P and can be joined with the current heap then their combined heap will satisfy Q. On the other hand, the *existential magic wand* $P \mathrel{-\!\!\oplus} Q$ asserts there exists a heap h_1 that satisfies P and can be combined with the current heap to satisfy Q.

The magic wand $-\!\!*$ was first proposed by Reynolds [Rey02] to reason about imperative programs with shared mutual data structure like lists and trees. One interesting property of this operator is its adjoint relationship with $*$:

$$P * Q \vdash R \quad \text{iff} \quad P \vdash Q \mathrel{-\!\!*} R.$$

Here the entailment $\vdash$ is defined via semantics, *i.e.*, $P_1 \vdash P_2 \stackrel{\text{def}}{=} \forall \kappa.\ \kappa \models P_1 \rightarrow \kappa \models P_2$. Furthermore, if the entailment happens to be bi-directional, we will refer to it as $P_1 \dashv\vdash P_2$.

Proof. $\Rightarrow$ direction: For simplicity, we assume program state only contains the heap component. Let $h \models P$. Then we need to show $h \models Q \mathrel{-\!\!*} R$. It is equivalent to show that for some h_1 s.t. $h_1 \perp h$ and $h_1 \models Q$ then $h_1 \uplus h \models R$. By definition of $*$, we derive $h \uplus h_1 \models P * Q$. Hence by the premise $P * Q \vdash R$, we conclude $h \uplus h_1 \models R$.

$\Leftarrow$ direction: Let $h \models P * Q$. Then there exist h_1, h_2 s.t. $h = h_1 \uplus h_2$, $h_1 \models P$ and $h_2 \models Q$. From the premise $P \vdash Q \mathrel{-\!\!*} R$, we conclude $h_1 \models Q \mathrel{-\!\!*} R$. As $h_2 \models Q$, we have $h_1 \uplus h_2 \models R$ by definition of $\mathrel{-\!\!*}$. Hence the result follows. □

The other two operators overlapping $\uplus\!\!\!\uplus$ and existential magic wand $-\!\!\oplus$ are used by Hobor and Villard [HV13] to reason about programs with graph-like structures. In their setting, the spatial graph can be specified recursively using $\uplus\!\!\!\uplus$ in which the overlapped parts among sub-graphs are not specified:

$$\mathsf{graph}(x) \ \Leftrightarrow\ (x = 0 \ \wedge\ \mathsf{emp}) \ \vee\ \exists d \exists l \exists r.\ x \mapsto (d, l, r) \ \uplus\!\!\!\uplus\ \mathsf{graph}(l) \ \uplus\!\!\!\uplus\ \mathsf{graph}(r).$$

Their main contribution is the invention of the *ramify* rule together with a proof system to help reason about overlaid data structures. This rule allows an effective transition from reasoning over SL formulas to reasoning over mathematical graph formulas:

$$\frac{\{P\}\ \mathsf{c}\ \{Q\} \qquad R \vdash (P * (Q \mathrel{-\!\!*} R'))}{\{R\}\ \mathsf{c}\ \{R'\}}\ \mathsf{Ramify}$$

2.3.3 Concurrent separation logic

Concurrent separation logic (CSL), proposed by O'Hean [OHe07], is an extension of separation logic to reason about correctness of concurrent programs. One key feature of CSL is the *parallel composition* $\mathsf{c}_1 \mid\mid \mathsf{c}_2$ which says two commands c_1 and c_2 are executed concurrently. For convenience, we usually call c_1 and c_2 as *concurrent threads*, *i.e.*, the entities that the commands refer to. To reason about the semantics of $\mid\mid$, CSL has the Parallel rule which says

that if two threads are independent of each other then the reasoning is modular:

$$\frac{\begin{array}{ll}\{P_1\}\ \mathsf{c}_1\ \{Q_1\} & \mathsf{fv}(\mathsf{c}_1, P_1, Q_1) \cap \mathsf{modified}(\mathsf{c}_2) = \emptyset \\ \{P_2\}\ \mathsf{c}_2\ \{Q_2\} & \mathsf{fv}(\mathsf{c}_2, P_2, Q_2) \cap \mathsf{modified}(\mathsf{c}_1) = \emptyset\end{array}}{\{P_1 * P_2\}\ \mathsf{c}_1\ ||\ \mathsf{c}_2\ \{Q_1 * Q_2\}}\ \mathsf{Parallel}$$

Here $\mathsf{fv}(\mathsf{c}, P, Q)$ is the set of free variables in predicates P, Q and command c whereas $\mathsf{modified}(\mathsf{c})$ is the set of variables whose values are changed by the command c. For example, the following code segment describes the behavior of two concurrent threads that update the two addresses 1 and 2 with stored value 42:

$$\frac{\dfrac{}{\{1 \mapsto \_\}\ [1] := 42\ \{1 \mapsto 42\}}\ \mathsf{Store} \qquad \dfrac{}{\{2 \mapsto \_\}\ [2] := 42\ \{2 \mapsto 42\}}\ \mathsf{Store}}{\{1 \mapsto \_\ *\ 2 \mapsto \_\}\ [1] := 42 \quad \| \quad [2] := 42\ \{1 \mapsto 42\ *\ 2 \mapsto 42\}}\ \mathsf{Parallel}$$

Unfortunately, it is often the case that we require certain communication amongst concurrent threads so that they can cooperate to achieve a common goal. The means of such communication is called *resource*, which is a fragment of memory shared among threads. In SL, resources are usually represented by a SL predicate, *i.e.*, $x \mapsto 1$ is the resource of a single address while $\mathsf{list}(x)$ is a list resource whose head is at address x:

$$\mathsf{list}(x) \stackrel{\text{def}}{=} (x = 0\ \wedge\ \mathsf{emp})\ \vee\ \exists d \exists n.\ x \mapsto (d, n)\ *\ \mathsf{list}(n).$$

A *race condition* happens when several threads attempt to access the same resource. For instance, the following code segment has two concurrent threads that want to access the stored value in address x:

$$[x] := 1; y := [x] + 1 \quad \| \quad [x] := 2; \mathbf{skip}$$

In the example above, we are interested in deriving the value of y of the first thread. Without the presence of the second thread, the answer is straightforward: y should be 2 because the stored value in x is set to 1 by previous command. Unfortunately, the second thread can *interfere* by updating the store value of x to 2 before the assignment of y is executed. As

a result, it is possible that the value of y is 3. One standard solution is to 'guard' x using semaphore or lock mechanism so that only one thread can access to x at a time, giving us the guarantee that y has value 2.

A formal semantics for CSL is out of the scope of this thesis. Instead we would like to briefly discuss the related literature review. The semantics of CSL was first defined by Brookes [Bro06, Bro07a] using trace semantics on a simplified language together with the soundness proof. Subsequently an alternative semantics of CSL was proposed by Hayman [HW06] using Petri nets or Calcagno *et al.* [COY07] on abstract separation algebras. Hobor *et al.* [Hob08, HAZ08] extended the operational semantics to reason about first-class locks, *i.e.*, locks that can be constructed and destroyed at runtime. Vafeiadis [Vaf11] presented a soundness proof for CSL in terms of standard operation semantics in which permissions are included inside the heap to reason about resource sharing.

A major disadvantage of CLS is the problem of synchronization, *i.e.*, its lack of expressiveness to reason about synchronous values. Assuming the interleaving semantics (*i.e.* threads are scheduled to run for certain time interval), we consider the following example:

$$\{x = 0\}$$

$$x := x + 1 \quad \| \quad x := x + 2$$

$$\{??\}$$

$$y := x$$

We are interested in finding the postcondition after $\|$. After two concurrent threads execute the commands $x := x + 1$ and $x := x + 2$, the value of x is assigned to y. The predicate in the precondition specifies the value of x to be 0 initially. Despite the interleaving effect, after $\|$ there is only one choice for the value of x which is 3. However, the strong assumptions in Parallel that two threads are independent prevent us from deriving that result. Furthermore, the use of lock or semaphore can significantly increase the overhead cost and thus hurts performance. To overcome these problems, Vafeiadis and Parkinson [VP07] proposed a flexible

$$\frac{(\kappa, \mathsf{c}_1) \rightsquigarrow (\kappa', \mathsf{c}_1')}{(\kappa, (\mathsf{c}_1 \mid\mid \mathsf{c}_2); \mathsf{c}) \rightsquigarrow (\kappa', (\mathsf{c}_1' \mid\mid \mathsf{c}_2); \mathsf{c})}\ \mathsf{LPar}$$

$$\frac{(\kappa, \mathsf{c}_2) \rightsquigarrow (\kappa', \mathsf{c}_2')}{(\kappa, (\mathsf{c}_1 \mid\mid \mathsf{c}_2); \mathsf{c}) \rightsquigarrow (\kappa', (\mathsf{c}_1 \mid\mid \mathsf{c}_2'); \mathsf{c})}\ \mathsf{RPar}$$

$$\frac{}{(\kappa, (\mathbf{skip} \mid\mid \mathbf{skip}); \mathsf{c}) \rightsquigarrow (\kappa, \mathbf{skip}; \mathsf{c})}\ \mathsf{SkipPar}$$

$$\frac{(\kappa, \mathsf{c}_1) \rightsquigarrow \mathbf{abort}}{(\kappa, (\mathsf{c}_1 \mid\mid \mathsf{c}_2); \mathsf{c}) \rightsquigarrow \mathbf{abort}}\ \mathsf{LAPar}$$

$$\frac{(\kappa, \mathsf{c}_2) \rightsquigarrow \mathbf{abort}}{(\kappa, (\mathsf{c}_1 \mid\mid \mathsf{c}_2); \mathsf{c}) \rightsquigarrow \mathbf{abort}}\ \mathsf{RAPar}$$

Figure 2.17: Small step relation for parallel composition in [Vaf11]

logic called RGSep which is the combination between Rely/Guarantee approach [Jon83] and CSL. He demonstrated the usefulness of RGSep in verifying fine-grained concurrent program among cooperative threads [Vaf07], *i.e.*, threads that agree on the order of certain executions. RGSep is the underlying logic in automatic reasoning tools such as SmallFootRG [CPV07] and CAVE [Vaf09].

We seal this subsection with a discussion about the small step relation for the parallel composition operator in [Vaf11]. Let **abort** be a special dead-end configuration that indicates the execution went wrong (*e.g.* a thread tries to read an invalid memory cell) then Fig. 2.17 provides details of the small step relation of ||. It is worth noting that two rules LAPar and RAPar are defined in a cautious manner: we can reach the dead-end configuration as soon as one of the component commands gets stuck.

2.4 Permission models

In concurrent programming, it is often the case that a given resource (*e.g.* a list or a binary tree) is shared among several threads for mutual computation. For instance, a list is shared

between two threads where one finds the minimal element while the other computes the maximal element. As a result, threads need either 'read' or 'write' access to the shared resource. However, the core of CSL is too restrictive to serve for this purpose. Consider the following Hoare triple that reasons about race condition:

$$\{a \mapsto 42 \ * \ x \mapsto \_ \ * \ y \mapsto \_\}$$

$$[x] := [a] \ \ || \ \ [y] := [a]$$

$$\{a \mapsto 42 \ * \ x \mapsto 42 \ * \ y \mapsto 42\}$$

We have the situation when both threads want to access to address a and both address x and y are updated with the stored value in a. To deduce the postcondition, we apply the Load rule to each thread individually*:

$$\frac{}{\begin{array}{c}\{a \mapsto 42 \ * \ x \mapsto \_\} \\ [x] := [a] \\ \{a \mapsto 42 \ * \ x \mapsto 42\}\end{array}}\ \textsf{Load} \qquad \frac{}{\begin{array}{c}\{a \mapsto 42 \ * \ y \mapsto \_\} \\ [y] := [a] \\ \{a \mapsto 42 \ * \ y \mapsto 42\}\end{array}}\ \textsf{Load}$$

However, when we try to apply the Parallel rule, both the precondition and postcondition contain $a \mapsto 42 * a \mapsto 42$, which is invalid. To overcome this problem, we allow *permissions* to be embedded into heaps so that single-cell heap $x \mapsto v$ can be split further. Particularly, we let $x \overset{\pi}{\mapsto} v$ denote the *fractional maps-to* which assigns address x with stored value v and permission π. One critical requirement is to find a suitable model for π. Boyland [Boy03] proposed the rational permission model $\mathcal{Q} = \langle [0,1], + \rangle$ in which permissions are rationals in the range $[0,1]$. In this model, 0 means *empty permission*, 1 is the *full permission* and the remaining rationals are *fractional permissions*. Nevertheless, a permission π can be split into two smaller permissions π_1 and π_2 using addition, *i.e.*, $\pi = \pi_1 + \pi_2$. Then the corresponding effect over the fractional mapping is the following equivalence:

$$x \xmapsto{\pi_1+\pi_2} v \quad \dashv\vdash \quad x \xmapsto{\pi_1} v \ * \ x \xmapsto{\pi_2} v.$$

*Here we cheat a little bit as the rule we apply is actually a simple combination of Load and Store rules.

Let $x \mapsto v$ be the syntactic sugar for $x \overset{1}{\mapsto} v$. Also, we assume that any cell $x \overset{\pi}{\mapsto} v$ for $\pi > 0$ is granted the read permission while the write permission is granted to memory cells with full permission, *i.e.*, $x \mapsto v$. Then the above Hoare triple can be proved using Store, Parallel and Consequence:

$$\dfrac{\dfrac{\dfrac{}{\{a \overset{0.5}{\longmapsto} 42 \ * \ x \mapsto \_\}\ [x] := [a]\ \{a \overset{0.5}{\longmapsto} 42 \ * \ x \mapsto 42\}}\ \text{Load} \qquad \dfrac{}{\{a \overset{0.5}{\longmapsto} 42 \ * \ y \mapsto \_\}\ [y] := [a]\ \{a \overset{0.5}{\longmapsto} 42 \ * \ y \mapsto 42\}}\ \text{Load}}{\{a \overset{0.5}{\longmapsto} 42 \ * \ x \mapsto \_ \ * \ a \overset{0.5}{\longmapsto} 42 \ * \ y \mapsto \_\}\ [x] := [a] \ ||\ [y] := [a]\ \{a \overset{0.5}{\longmapsto} 42 \ * \ x \mapsto 42 \ * \ a \overset{0.5}{\longmapsto} 42 \ * \ y \mapsto 42\}}\ \text{Parallel}}{\{a \mapsto 42 \ * \ x \mapsto \_ \ * \ y \mapsto \_\}\ [x] := [a] \ ||\ [y] := [a]\ \{a \mapsto 42 \ * \ x \mapsto 42 \ * \ y \mapsto 42\}}\ \text{Consequence}$$

To construct the semantics to SL with rational permissions, we need to modify the definition of heap. In this setting, we use *fractional heaps* which are a partial mapping from addresses to pairs of values and permissions:

$$h : \mathsf{Addr} \rightharpoonup \mathsf{Val} \times \mathsf{Perm}.$$

It is worth noticing that the normal heap is actually a special fractional heap in which all permissions are full. For simplicity, we omit the break brk from program state κ and thus κ is now a pair of stack ρ and fractional heap h. Then the semantics of $x \overset{\pi}{\mapsto} e$ is defined as address x with stored value e and permission π:

$$(\rho, h) \models x \overset{\pi}{\mapsto} e \ \overset{\text{def}}{=}\ \mathsf{dom}(h) = \{x\} \ \wedge \ [|e|]_\rho = v \ \wedge \ h(x) = (v, \pi).$$

The notion of disjoint heaps is relaxed in the sense that their domains can still overlap but each overlapped address must agree on values and their permissions are joinable. In detail, two fractional heaps h_1 and h_2 are disjoint, denoted by $h_1 \perp h_2$, if for any shared address a

s.t. $h_1(a) = (v_1, \pi_1)$ and $h_2(a) = (v_2, \pi_2)$ then $v_1 = v_2$ and $\pi_1 + \pi_2 \leq 1$:

$$h_1 \perp h_2 \stackrel{\text{def}}{=} \forall a, v_1, v_2, \pi_1, \pi_2.\ a \in \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) \rightarrow h_1(a) = (v_1, \pi_1) \rightarrow h_2(a) = (v_2, \pi_2) \rightarrow v_1 = v_2 \wedge \pi_1 + \pi_2 \leq 1.$$

Consequently, the combined heap of h_1 and h_2, denoted as $h_1 \uplus h_2$, has domain $\mathsf{dom}(h) = \mathsf{dom}(h_1) \cup \mathsf{dom}(h_2)$ and its permission of each address is the permission sum from h_1 and h_2 in the respective address:

$$h_1 \uplus h_2 \stackrel{\text{def}}{=} \begin{cases} h_1 \perp h_2, \\ \mathsf{dom}(h) = \mathsf{dom}(h_1) \cup \mathsf{dom}(h_2), \\ h(a) = h_1(a) \text{ if } a \in \mathsf{dom}(h_1) \backslash \mathsf{dom}(h_2), \\ h(a) = h_2(a) \text{ if } a \in \mathsf{dom}(h_2) \backslash \mathsf{dom}(h_1), \\ h(a) = (v, \pi_1 + \pi_2) \text{ if } h_1(a) = (v, \pi_1) \text{ and } h_2(a) = (v, \pi_2). \end{cases}$$

Furthermore, let π_1, π_2 be permissions such that $\pi_1 + \pi_2 \leq 1$ then we can verify that the permission $\pi_1 + \pi_2$ can be split via $*$ in the fractional mapping:

$$(h, \rho) \models x \overset{\pi_1+\pi_2}{\longmapsto} v \quad \text{iff} \quad (h, \rho) \models x \overset{\pi_1}{\longmapsto} v \ * \ x \overset{\pi_2}{\longmapsto} v.$$

Proof. We only prove the $\Leftarrow$ direction here as the other direction is similar. By definition of $*$, there exist h_1, h_2 s.t. $h_1 \uplus h_2 = h$, $(\rho, h_1) \models x \overset{\pi_1}{\longmapsto} v$ and $(\rho, h_2) \models x \overset{\pi_2}{\longmapsto} v$. Each h_i is a single-cell heap of the same address x and stored value v while their permissions are π_1 and π_2. As $h_1 \uplus h_2 = h$, we deduce that h is also a single-cell heap of address x, stored value v and permission $\pi_1 + \pi_2$. Thus the result follows. □

Although the rational model $\mathcal{Q} = \langle \mathbb{Q}, + \rangle$ is simple and easy to use, it suffers from the disjointness problem described in Figure 1.1 where the logic fails to distinguish a tree from a DAG.

There are other flavors of permission besides rationals. Bornat *et al.* [BCOP05] introduced

integer counting permissions $\langle \mathbb{Z}, +, 0 \rangle$ to reason about semaphores and combined rationals and integers into a hybrid permission model. Heule *et al.* [HLMS11] flexibly allowed permissions to be either concretely rational or abstractly read-only to lower the nuisance of detailed accounting. A more general read-only permissions was proposed by Charguéraud and Pottier [CP17] that transforms a predicate P into read-only mode $\mathsf{RO}(P)$ which can duplicated/merged with the bi-entailment $\mathsf{RO}(P) \dashv\vdash \mathsf{RO}(P) \star \mathsf{RO}(P)$. Their permissions distribute pleasantly over disjunction and existential quantifier but only work one way for $\star$, *i.e.*, $\mathsf{RO}(H_1 \star H_2) \vdash \mathsf{RO}(H_1) \star \mathsf{RO}(H_2)$. Parkinson [Par05] proposed subsets of the natural numbers for shares $\langle \mathcal{P}(\mathbb{N}), \uplus \rangle$ to fix the disjointness problem. Compared to tree shares, Parkinson's model is less practical computationally and does not have an obvious multiplicative structure.

Verification tools often implement rational permissions because of its simplicity. For example, VeriFast [JSP10] uses rationals to verify programs with locks and semaphores. It also allows simple and restrictive forms of scaling permissions which can be applied uniformly over standard predicates. On the other hand, HIP/SLEEK [LCT15] uses rationals to model "thread as resource" so that the ownership of a thread and its resources can be transferred. Chalice [LM09] has rational permissions to verify properties of multi-threaded, objected-based programs such as data races and dead-locks. Viper [MSS16] has an expressive intermediate language that supports both rational and abstract permissions. However, a number of verification tools have chosen tree shares due to their better metatheorical properties. VST [App11b] is equipped with tree share permissions and an extensive tree share library. HIP/SLEEK uses tree shares to verify the barrier structure [HG11] although their permission procedure at that time is highly incomplete. Lastly, tree share permissions are featured in Heap-Hop [Vil11] to reason over asynchronous communications.

Chapter 3

Reasoning over disjoint fractional permissions

> "*You cannot ever reach perfection, but you can believe in an asymptote toward which you are ceaselessly striving.*"
>
> Paul Kalanithi, *When Breath Becomes Air*.

Resource sharing is a fundamental phenomenon in concurrent programming where several threads have permissions to access a common resource. As a result, the logic for verification needs to capture the notion of permission ownership and transferring. One typical practice is the use of rational numbers in $(0, 1]$ as permissions in which 1 is the full permission and the rest are fractional permissions. Unfortunately, rational permissions are not a good fit for separation logic because they remove the essential "disjointness" feature of the logic itself which was discussed in §1.1. In this chapter, we propose a general logic framework with predicate multiplication that supports permission reasoning in separation logic while desirably preserving the disjointness property. We show that our framework is applicable to sophisticated verification tasks such as doing induction over the finiteness of the heap within the object logic or carrying out bi-abductive inference. We can also prove precision of recursive predicates within the object logic. We introduce "scaling separation algebras" (SSA), a compositional extension of separation algebras, to model our logic, and use them to construct a concrete model. We discuss several applications of shares to model other permission types such as token-counting and string-like permissions. Last but not least, the application of tree shares is demonstrated by the fact they are one of the structures that satisfy SSA.

This chapter is organized as follows*:

1. In §3.1, we introduce our desired scaling rules for predicate multiplication together with an illustrated example.
2. In §3.2, we explain the bi-abduction inference in the presence of fractional permissions and predicate multiplication.
3. In §3.3, we display and discuss our core proof system for predicate multiplication together with concrete examples.
4. In §3.4, we introduce "scaling separation algebras" (SSA), a compositional extension of separation algebras, to model our logic, and use them to construct a concrete model.
5. In §3.5, we provide justification for our side conditions by showing the consequence of the disjointness axiom with respect to other axioms in SSA.
6. In §3.6, we discuss the shortcoming of rational permissions and propose applications of our SSA in modeling other permission types such as token-counting and string-like permissions.
7. In §3.7, we discuss the tool ShareInfer that employs our proof theories.
8. In §3.8, we mention the related work and draw our conclusion.

3.1 Predicate multiplication

Consider the toy program in Figure 3.1. Starting from the tree rooted at `x`, the program itself is dead simple. First (line 3) we check if the `x` is null, *i.e.* if we have reached a leaf; if so, we `return`. If not, we split into parallel threads (lines 4–6 and 7–9) that do some processing on the root data in both branches. In the toy example, the processing just `print`s out the root data (lines 4 and 7); the `print` command is unimportant: what is important that we somehow access some of the data in the tree. After processing the root, both parallel branches call the `processTree` function recursively on the left `x->l` (lines 5 and 8) and right

*The materials in this chapter are taken from the paper "Logical Reasoning over Disjoint Fractional Permissions" [LH17], a joint work with my supervisor Aquinas Hobor.

```
1 struct tree {int d; struct tree* l; struct tree* r;};
2 void processTree(struct tree* x) {
3   if (x == 0) { return; }

4   print(x -> d);          ||  7   print(x -> d);
5   processTree(x -> l);    ||  8   processTree(x -> l);
6   processTree(x -> r);    ||  9   processTree(x -> r);

10 }
```

Figure 3.1: The `processTree` function in a C-like language with a parallel operator $\mathtt{c}_1||\mathtt{c}_2$

`x->r` (lines 6 and 9) branches, respectively. After both parallel processes have terminated, the function returns (line 10). The initial caller (unshown) may then *e.g.* deallocate the tree. The program is simple, so we would like its verification to be equally simple.

Predicate multiplication is the tool that leads to a simple proof. Specifically, we would like to verify that `processTree` has the specification:

$$\forall \pi, x.\ \ (\ \{\pi \cdot \mathsf{tree}(x)\}\ \ \mathtt{processTree}(x)\ \ \{\pi \cdot \mathsf{tree}(x)\}\ ).$$

where the tree predicate is defined as:

$$\mathsf{tree}(x) \stackrel{\text{def}}{=} (x = \mathsf{null}) \vee (\exists d, l, r.\ x \mapsto (d, l, r) * \mathsf{tree}(l) * \mathsf{tree}(r)). \tag{3.1}$$

The formal definition of predicate multiplication $\pi \cdot P$ is postponed until §3.3.2 as we would like to maintain a high level of abstraction. Intuitively, one can understand the notion $\pi \cdot P$ as "the current heap has a fractional permission π of the predicate P". Our precondition and postcondition both say that x is a pointer to a heap-represented π-owned tree. Critically, we want to ensure that our π-share at the end of the program is equal to the π-share at the beginning. This way if our initial caller had full ownership (denoted by $\mathcal{F}$) before calling `processTree`, he will have full ownership afterwards (allowing him to *e.g.* deallocate the tree).

The intuition behind the proof can be explained informally as follow. First in line 3, we check if `x` is null; if so we are in the base case of the tree definition and can simply return.

If not we can eliminate the left disjunct and can proceed to split the $*$-separated bits into disjoint subtrees `l` and `r`, and then dividing the ownership of those bits into two "halves". Let $\mathcal{L}$ be any share other than $\mathcal{E}$ (empty permission) or $\mathcal{F}$ (full permission) and let $\mathcal{R} \stackrel{\text{def}}{=} \overline{\mathcal{L}}$ be its compliment; intuitively $\mathcal{L}$ is the "left half" and $\mathcal{R}$ is the "right half". When we start start the parallel computation on lines 4 and 7 we want to pass the left branch of the multiplication computation the $\mathcal{L} \otimes \pi$-share of the spatial resources, and the right branch of the multiplication computation the $\mathcal{R} \otimes \pi$. In both branches we then need to show that we can read from the data cell, which in the simple policy we use for this thesis boils down to making sure that the product of two non-$\mathcal{E}$ shares cannot be $\mathcal{E}$. This is a basic property for reasonable share models with multiplication. In the remainder of the parallel code (lines 5–6 and 8–9) we need to make recursive calls, which is done by simply instantiating π with $\mathcal{L} \otimes \pi$ and $\mathcal{R} \otimes \pi$ in the recursive specification (as well as `l` and `r` for x). The later half proof after the parallel call is pleasantly symmetric to the first half in which we fold back the original tree predicate by merging the two halves $\mathcal{L} \otimes \pi$ and $\mathcal{R} \otimes \pi$ back into π. Consequently, we arrive at the postcondition $\pi \cdot \mathsf{tree}(x)$, which is identical to the precondition.

3.1.1 Proof rules for predicate multiplication

In Figure 3.3 we put the formal verification for `processTree`, which follows the informal argument very closely. However, before we go through it, let us consider the reason for this alignment: because the key rules for reasoning about predicate multiplication are bidirectional. These rules are given in Figure 3.2. The non-spatial rules are all straightforward and follow the basic pattern that predicate multiplication both pushes into and pulls out of the operators of our logic without meaningful side conditions*. The DotPure rule means that predicate multiplication ignores pure facts, too. Complicating the picture slightly, predicate multiplication pushes into implication $\Rightarrow$ but does not pull out of it. Combining DotImpl with DotPure we get a one-way rule for negation: $\pi \cdot (\neg P) \vdash \neg\pi\cdot$. We will explain why we cannot get both directions in §3.3.2 and §3.5.

*As a minor side condition we require that the universe of quantification be nonempty in the DotUniv rule.

$$\frac{P \vdash Q}{\pi \cdot P \vdash \pi \cdot Q}\ \textsc{DotPos} \qquad \frac{}{\pi \cdot \langle P \rangle \dashv\vdash \langle P \rangle}\ \textsc{DotPure} \qquad \frac{}{\pi \cdot (P \Rightarrow Q) \vdash (\pi \cdot P) \Rightarrow (\pi \cdot Q)}\ \textsc{DotImpl}$$

$$\frac{}{\pi \cdot (P \wedge Q) \dashv\vdash (\pi \cdot P) \wedge (\pi \cdot Q)}\ \begin{matrix}\textsc{Dot}\\ \textsc{Conj}\end{matrix} \qquad \frac{}{\pi \cdot (P \vee Q) \dashv\vdash (\pi \cdot P) \vee (\pi \cdot Q)}\ \begin{matrix}\textsc{Dot}\\ \textsc{Disj}\end{matrix} \qquad \frac{}{\pi \cdot (\neg P) \vdash \neg\pi \cdot P}\ \begin{matrix}\textsc{Dot}\\ \textsc{Neg}\end{matrix}$$

$$\frac{\tau \neq \emptyset}{\pi \cdot (\forall x : \tau.\ P(x)) \dashv\vdash \forall x : \tau.\ \pi \cdot P(x)}\ \textsc{DotUniv} \qquad \frac{}{\pi \cdot (\exists x : \tau.\ P(x)) \dashv\vdash \exists x : \tau.\ \pi \cdot P(x)}\ \textsc{DotExis}$$

$$\frac{}{\mathcal{F} \cdot P \dashv\vdash P}\ \begin{matrix}\textsc{Dot}\\ \textsc{Full}\end{matrix} \qquad \frac{}{\pi_1 \cdot (\pi_2 \cdot P) \dashv\vdash (\pi_1 \otimes \pi_2) \cdot P}\ \begin{matrix}\textsc{Dot}\\ \textsc{Dot}\end{matrix} \qquad \frac{}{\pi \cdot x \mapsto y \dashv\vdash x \overset{\pi}{\mapsto} y}\ \begin{matrix}\textsc{Dot}\\ \textsc{MapsTo}\end{matrix}$$

$$\frac{\mathsf{precise}(P)}{(\pi_1 \oplus \pi_2) \cdot P \dashv\vdash (\pi_1 \cdot P) * (\pi_2 \cdot P)}\ \textsc{DotPlus} \qquad \frac{P \vdash \mathsf{uniform}(\pi') \qquad Q \vdash \mathsf{uniform}(\pi')}{\pi \cdot (P * Q) \dashv\vdash (\pi \cdot P) * (\pi \cdot Q)}\ \textsc{DotStar}$$

Figure 3.2: Distributivity of the scaling operator over pure and spatial connectives

Most of the spatial rules are also simple in which $\langle P \rangle \overset{\text{def}}{=} |P| \wedge \mathsf{emp}$. Accordingly, since $\langle \top \rangle \dashv\vdash \mathsf{emp}$, DotPure yields $\pi \cdot \mathsf{emp} \dashv\vdash \mathsf{emp}$. The DotFull rule says that $\mathcal{F}$ is the scalar identity on predicates, just as it is the multiplicative identity on the share model itself. The DotDot rule allows us to "collapse" repeated predicate multiplication using share multiplication; we will shortly see how we use it to verify the recursive calls to `processTree` in lines 5–6 and 8–9. Similarly, the DotMapsTo rule shows how predicate multiplication combines with basic maps-to by multiplying the associated shares together. All three rules are bidirectional and require no side conditions. We also want to mention that these rules are proved correct with respect to the assumed model in Coq [Dev].

While the last two rules are both bidirectional, they both have side conditions. The DotPlus rule shows how predicate multiplication distributes over $\oplus$. The $\vdash$ direction does not require a side condition, but the $\dashv$ direction we require that P be *precise* in the usual separation logic sense. Informally, precision is some kind of uniqueness condition that helps pinpoint the exact shape of the given predicate. One common place where a predicate is not precise is due to disjunction, *e.g.*, $x \mapsto 1 \vee y \mapsto 2$. Precision will be discussed in §3.3.3; for now a

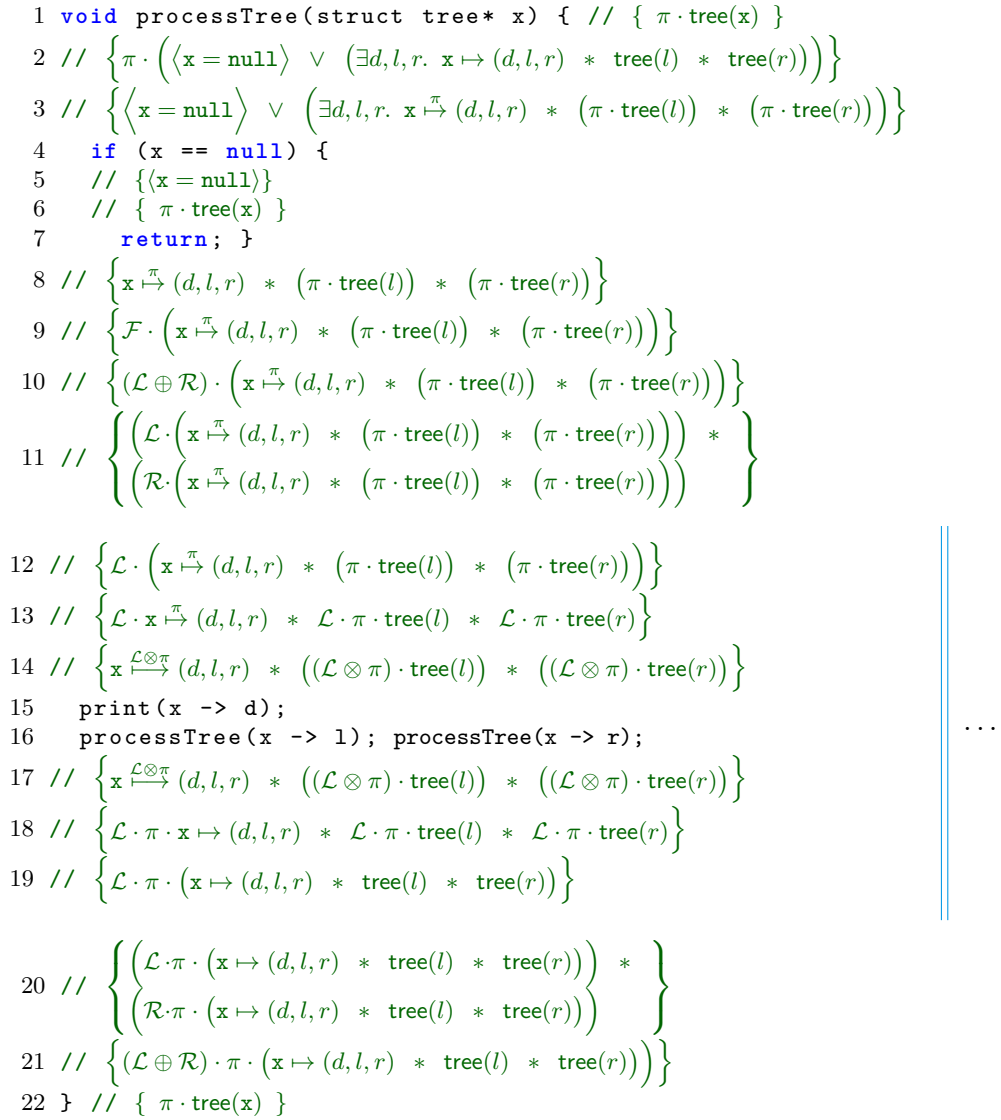


Figure 3.3: Reasoning with the scaling operator $\pi \cdot P$.

simple counterexample shows why it is necessary:

$$\mathcal{L} \cdot (x \mapsto a \vee (x+1) \mapsto b) * \mathcal{R} \cdot (x \mapsto a \vee (x+1) \mapsto b) \quad \not\vdash \quad \mathcal{F} \cdot (x \mapsto a \vee (x+1) \mapsto b).$$

The premise is also consistent with $x \overset{\mathcal{L}}{\mapsto} a * (x+1) \overset{\mathcal{R}}{\mapsto} b$, which is inconsistent with the conclusion.

The DotStar rule shows how predicate multiplication distributes into and out of the separating conjunction $*$. It is also bidirectional, but again requires a side condition of

uniformity. Informally, $P \vdash \mathsf{uniform}(\pi)$ asserts that any heap satisfies P has the permission π uniformly at each of its defined addresses. Interestingly, this condition is trivial in the standard SL without fractional permissions because any predicate is automatically full-uniform. In §3.5 we explain why we cannot admit this rule without a side condition.

In the meantime, let us argue that most predicates used in practice in separation logic are uniform. First, every SL predicate defined in non-fractional settings, such as $\mathsf{tree}(x)$, is $\mathcal{F}$-uniform. Second, P is a π-uniform predicate if and only if $\pi' \cdot P$ is $(\pi' \otimes \pi)$-uniform. Third, the $*$-conjunction of two π-uniform predicates is also π-uniform. Since a significant motivation for predicate multiplication is to allow standard SL predicates to be used in fractional settings, these already cover many common cases in practice. It is useful to consider examples of non-uniform predicates for contrast. Here are three (where we have removed the base case to cut down on clutter):

$$\mathsf{slist}(x) \quad \dashv\vdash \quad \exists d, n.\ ((\langle d = 17\rangle * x \overset{\mathcal{L}}{\mapsto} (d, n)) \vee (\langle d \neq 17\rangle * x \overset{\mathcal{R}}{\mapsto} (d, n))) * \mathsf{slist}(n).$$

$$\mathsf{dlist}(x) \quad \dashv\vdash \quad \exists d, n.\ x \mapsto d, n * \mathcal{L} \cdot \mathsf{dlist}(n).$$

$$\mathsf{dtree}(x) \quad \dashv\vdash \quad \exists d, l, r.\ x \mapsto d, l, r * \mathcal{L} \cdot \mathsf{dtree}(l) * \mathcal{R} \cdot \mathsf{dtree}(r).$$

The $\mathsf{slist}(x)$ predicate owns different amounts of permissions at different memory cells depending on the value of those cells. The $\mathsf{dlist}(x)$ predicate owns decreasing amounts of the list, *e.g.* the first cell is owned more than the second, which is owned more than the third. The $\mathsf{dtree}(x)$ predicate is even stranger, owning different amounts of different branches of the tree, essentially depending on the path to the root. None of these predicates mix well with DotStar, but perhaps they are not useful to verify many programs in practice, either.

In §3.3.2 and §3.3.3 we will discuss how to prove predicates are precise and uniform. In §3.3.5 will demonstrate our techniques to do so by applying them to two examples.

3.1.2 Verification of `processTree` using predicate multiplication

We will explain in detail how the proof of `processTree` (Fig. 3.3) is carried out using scaling rules (Fig. 3.2). In line 2, we unfold the definition of predicate $\mathsf{tree}(x)$ which consists of one base case and one inductive case. We reach line 3 by pushing π inward using various rules DotPure, DotDisj, DotExis, DotMapsto and DotStar. To use DotStar we must prove that $\mathsf{tree}(x)$ is $\mathcal{F}$-uniform, which we show how to do in §3.3.5. Typically we prove this lemma once and use it many times.

The base base $\mathtt{x} = \mathtt{null}$ is handled in lines 5-6 by applying rule DotPure, *i.e.*, $\langle \mathtt{x} = \mathtt{null} \rangle \vdash \pi \cdot \langle \mathtt{x} = \mathtt{null} \rangle$ and then DotPos, $\pi \cdot \langle \mathtt{x} = \mathtt{null} \rangle \vdash \pi \cdot \mathsf{tree}(x)$. For the inductive case, we first apply DotFull in line 9 and then replace $\mathcal{F}$ with $\mathcal{L} \oplus \mathcal{R}$ (recall that $\mathcal{R}$ is $\mathcal{L}$'s compliment). On line 11 we use DotPlus to translate the split on shares with $\oplus$ into a split on heaps with $*$.

We show only one parallel process; the other is a mirror image. Line 12 gives the precondition from the Parallel rule, and then in lines 13 and 14 we continue to "push in" the predicate multiplication. To verify the code in lines 15–16 just requires Frame. Notice that we need the DotDot rule to "collapse" the two uses of predicate multiplication into one so that we can apply the recursive specification (with the new π' in the recursive precondition equal to $\mathcal{L} \otimes \pi$).

Having taken the predicate completely apart, it is now necessary to put Humpty Dumpty back together again. Here is why it is vital that all of our proof rules are bidirectional, without which we would not be able to reach the final postcondition $\pi \cdot \mathsf{tree}(x)$. The final wrinkle is that for line 21 we must prove the precision of the $\mathsf{tree}(x)$ predicate. We show how to do so (on a slightly simpler example) in §3.3.5, but typically in a verification this is proved once per predicate as a lemma.

3.2 Bi-abduction inference

In this section, we will tackle the bi-abduction problem in the context of fractional permissions. Simply put, the task of bi-abduction is to fulfill missing information in an incomplete SL entailment. To be precise, given partial entailment $A * [??] \vdash B * [??]$, we would like to identify the two different missing pieces $[??]$ in the antecedent and consequent to complete the entailment. The first piece is called the *anti-frame* while the second is called the *frame*. This problem is motivated by the scalability problem of verification tools when dealing with sizable programs [GVA07, YLB+08]. In particular, bi-abductive inference can be paired up with the Frame rule [IO01] to produce a smooth compositional reasoning framework:

$$\dfrac{\dfrac{\overline{\{A_1\}c_1\{A_2\}}}{\{A_1 * F_1\}c_1\{A_2 * F_1\}}\ \textsc{Frame} \qquad \dfrac{\dfrac{\overline{\{B_1\}c_2\{B_2\}}}{\{B_1 * F_2\}c_1\{B_2 * F_2\}}\ \textsc{Frame} \quad \dfrac{\overline{A_2 * [F_1] \vdash B_1 * [F_2]}}{\{A_2 * F_1\}c_2\{B_2 * F_2\}}\ \textsc{BiAb Con}}{\{A_1 * F_1\}c_1; c_2\{B_2 * F_2\}}\ \textsc{Composition}$$

In short, the antecedent and consequent of $c_1; c_2$ are derived from the ones in c_1 and c_2 with the aid of bi-abduction and frame rule. Consequently, this approach promises a scalable verification framework and thus bi-abduction received considerable research attention recently [CDOY09, BGK17, CDV09, LGQC14]. A general procedure to solve this problem consists of two subroutines: the *abduction* and *frame inference* to induce the anti-frame and frame respectively. Thus our task is to explain the necessary steps to generalize the two subroutines to handle fractional entailments.

3.2.1 Fractional residue computation

Consider the fractional point-to bi-abduction problem with rational permissions:

$$a \overset{\pi_1}{\longmapsto} b * [??] \vdash a \overset{\pi_2}{\longmapsto} b * [??].$$

There are three cases to consider, *i.e.*, $\pi_1 = \pi_2$, $\pi_1 < \pi_2$ or $\pi_1 > \pi_2$. In the first case, both the (minimal) anti-frame F_a and frame F_f are emp; for the second case we have $F_\mathrm{a} = \mathsf{emp}$,

$F_\mathrm{f} = a \overset{\pi_2 - \pi_1}{\longmapsto} b$ and the last case gives us $F_\mathrm{a} = a \overset{\pi_1 - \pi_2}{\longmapsto} b, F_\mathrm{f} = \mathsf{emp}$. Here we compute the residue permission using rational subtraction, which is essentially the inverse of addition. In general, we can attempt to define subtraction $\ominus$ from a fractional model $\langle \mathcal{U}, \oplus \rangle$ as $a \ominus b = c \overset{\text{def}}{=} b \oplus c = a$. However, this definition is too coarse as we require subtraction to be a total function so that the residue is always computable. To overcome this problem, we relax the requirements for $\ominus$, asking only that it satisfies the following two properties:

$$C_1 : a \oplus (b \ominus a) = b \oplus (a \ominus b) \qquad C_2 : a \ll b \oplus c \Rightarrow a \ominus b \ll c.$$

where $a \ll b \overset{\text{def}}{=} \exists c.\ a \oplus c = b$. The condition C_1 provides a convenient way to compute the fractional residue in both the frame and anti-frame while C_2 asserts that $a \ominus b$ is efficiently the minimal element that when joined with b becomes greater than a. For example, subtraction in rational permissions can be defined as:

$$a \ominus b \overset{\text{def}}{=} \text{if } a \geq b \text{ then } a - b \text{ else } 0.$$

Using subtraction, the fractional residue can be derived in a unique and pleasant way:

$$\frac{}{a \overset{\pi_1}{\longmapsto} b * a \overset{\pi_2 \ominus \pi_1}{\longmapsto} b \vdash a \overset{\pi_2}{\longmapsto} b * a \overset{\pi_1 \ominus \pi_2}{\longmapsto} b} \textsc{Msub}$$

Generally, if P is precise then we can use the following rule to compute the residue of P:

$$\frac{\text{precise}(P)}{\pi_1 \cdot P * (\pi_2 \ominus \pi_1) \cdot P \vdash \pi_2 \cdot P * (\pi_1 \ominus \pi_2) \cdot P} \textsc{Psub}$$

Furthermore, we can show that the above solution is minimal with respect to $\ll$, *i.e.*:

$$\pi_1 \oplus a = \pi_2 \oplus b \Rightarrow \pi_2 \ominus \pi_1 \ll a \wedge \pi_1 \ominus \pi_2 \ll b.$$

Proof. From $\pi_1 \oplus a = \pi_2 \oplus b$, we have $\pi_1 \ll \pi_2 \oplus b$. Thus by C_2, we arrive $\pi_1 \ominus \pi_2 \ll b$. The other inequality is similar. □

3.2.2 Extension of predicate axioms

Programs commonly contain recursive structures such as list, tree as these structures allows efficient mechanisms to store and retrieve data. Correspondingly, the assertion language is enriched with inductive predicates to reason about recursive structures. In general, the inference problem over recursive structures is hard, *e.g.* the first-order theory of Peano arithmetic $\langle 0, S, +, \times \rangle$ is undecidable [Chu36]. As a result, verification tools [CDD+15, LGQC14] often contain a list of predicate axioms/facts and use them to derive entailments of the related predicates. These axioms are represented as entailment $A \vdash B$ and are classified into folding and unfolding rules. As their names suggest, folding rules are predicate constructors whereas unfolding rules help unroll a predicate into smaller components. Basically, one can derive useful lemmas from the definition of inductive predicates *e.g.* [Bro07b]. For example, some axioms for the tree predicate (Eqn. 3.1) are:

$$F_1 : x = 0 \wedge \mathsf{emp} \vdash \mathsf{tree}(x) \qquad F_2 : x \mapsto (v, x_1, x_2) * \mathsf{tree}(x_1) * \mathsf{tree}(x_2) \vdash \mathsf{tree}(x).$$

$$U : \mathsf{tree}(x) \wedge x \neq 0 \vdash \exists v, x_1, x_2.\ x \mapsto (v, x_1, x_2) * \mathsf{tree}(x_1) * \mathsf{tree}(x_2).$$

We show how to transform these axioms into fractional forms. The key ingredient is the DotPos rule from Fig, 3.2 that extracts a fractional portion of entailment, *i.e.*, $(P \vdash Q) \Rightarrow (\pi \cdot P \vdash \pi \cdot Q)$. Using other scaling rules, we can transform the rule U above into the general fractional form U':

$$\dfrac{\dfrac{U : \mathsf{tree}(x) \wedge x \neq 0 \ \vdash\ \exists v, x_1, x_2.\ x \mapsto (v, x_1, x_2) * \mathsf{tree}(x_1) * \mathsf{tree}(x_2)}{\pi \cdot (\mathsf{tree}(x) \wedge x \neq 0) \ \vdash\ \pi \cdot (\exists v, x_1, x_2.\ x \mapsto (v, x_1, x_2) * \mathsf{tree}(x_1) * \mathsf{tree}(x_2))}\ \textsc{DotPos}}{U' : \pi \cdot \mathsf{tree}(x) \wedge x \neq 0 \ \vdash\ \exists v, x_1, x_2.\ x \overset{\pi}{\mapsto} (v, x_1, x_2) * \pi \cdot \mathsf{tree}(x_1) * \pi \cdot \mathsf{tree}(x_2)}\ \text{Scaling rules}$$

The fractional forms for the two folding rules F_1 and F_2 can be derived in a similar fashion:

$$F_1' : x = 0 \wedge \mathsf{emp} \vdash \pi \cdot \mathsf{tree}(x) \qquad F_2' : x \overset{\pi}{\mapsto} (v, x_1, x_2) * \pi \cdot \mathsf{tree}(x_1) * \pi \cdot \mathsf{tree}(x_2) \vdash \pi \cdot \mathsf{tree}(x).$$

As our scaling rules are all bi-directional, they can be applied both in the antecedent and consequent to produce a smooth transformation to fractional axioms. Also, recall that our

DotStar rule $\pi \cdot (P * Q) \dashv\vdash \pi \cdot P * \pi \cdot Q$ has a side condition that both P and Q are π'-uniform. Happily, this condition is trivial in the transformation as standard predicates (*i.e.* those without permissions) are automatically $\mathcal{F}$-uniform. Alternatively, the precision and uniformity properties can be transfered to fractional predicates by the following rules:

$$\text{precise}(\pi \cdot P) \Leftrightarrow \text{precise(P)} \qquad P \vdash \mathsf{uniform}(\pi) \Leftrightarrow \pi' \cdot P \vdash \mathsf{uniform}(\pi' \otimes \pi).$$

For example, $\mathsf{tree}(x)$ is precise and full-uniform. Thus $\pi \cdot \mathsf{tree}(x)$ is also precise and π-uniform.

3.2.3 Abductive inference

Given predicates A and B, we find the anti-frame F_{a} that satisfies the entailment:

$$A * F_{\text{a}} \vdash B.$$

In addition, we want the predicate $A * F_{\text{a}}$ to be satisfiable and F_{a} to be reasonably minimal (measured by the size of heap satisfies it). Between the two subroutines in bi-abduction, the abduction problem is technically more challenging as tools need to generate hypotheses to fill in the anti-frame. With inductive definitions, the hypothesis space is infinitely large and thus it is impractical, if not impossible, to develop a complete anti-frame solver. Calcagno *et al.* [CDOY09] tackled this problem by presenting a general framework for anti-frame inference which contains rules of the form:

$$\frac{\Delta' * [M'] \rhd H' \qquad \text{Cond}}{\Delta * [M] \rhd H}$$

in which Cond is the side condition together with consequents (H, H'), heap formulas (Δ, Δ') and anti-frames (M, M'). During the abductive inference, the tool starts with some base axiom and gradually applies proof rules to propagate the anti-frame. Simultaneously, it also ensures the current constructed antecedent Δ and consequent H are comparable with the original ones while Δ does not contradict with the anti-frame. On the other hand, these abduction rules are directly constructed from predicate axioms. As we previously discussed

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(x_2 = 0 \wedge \mathsf{emp}) * [\mathsf{emp}] \rhd \mathsf{emp}}\ \textsc{Base}}{(x_2 = 0 \wedge \mathsf{emp}) * [\mathsf{emp}] \rhd \pi_3 \cdot \mathsf{tree}(x_2)}\ \mathsf{F_1}'}{\pi_2 \cdot \mathsf{tree}(x_1) * (x_2 = 0 \wedge \mathsf{emp}) * [(\pi_3 \ominus \pi_2) \cdot \mathsf{tree}(x_1)] \rhd \pi_3 \cdot \mathsf{tree}(x_1) * \pi_3 \cdot \mathsf{tree}(x_2)}\ \textsc{Psub}}{x \overset{\pi_1}{\longmapsto} (v, a, x_2) * \pi_2 \cdot \mathsf{tree}(x_1) * (x_2 = 0 \wedge \mathsf{emp}) * [(\pi_3 \ominus \pi_2) \cdot \mathsf{tree}(x_1) * x \overset{\pi_3 \ominus \pi_1}{\longmapsto} (v, a, x_2)] \rhd x \overset{\pi_3}{\longmapsto} (v, a, x_2) * \pi_3 \cdot \mathsf{tree}(x_1) * \pi_3 \cdot \mathsf{tree}(x_2)}\ \textsc{Msub}
$$

$$
\dfrac{}{x \overset{\pi_1}{\longmapsto} (v, a, x_2) * \pi_2 \cdot \mathsf{tree}(x_1) * (x_2 = 0 \wedge \mathsf{emp}) * [a = x_1 \wedge (\pi_3 \ominus \pi_2) \cdot \mathsf{tree}(x_1) * x \overset{\pi_3 \ominus \pi_1}{\longmapsto} (v, a, x_2)] \rhd \pi_3 \cdot \mathsf{tree}(x)}\ \textsc{match}\ +\mathrm{F}_2'
$$

Figure 3.4: Abductive inference

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{emp} \rhd \mathsf{emp} * [\mathsf{emp}]}\ \textsc{Base}}{x \overset{\pi_1 \oplus (\pi_3 \ominus \pi_1)}{\longmapsto} (v, x_1, x_2) \rhd x \overset{\pi_3}{\longmapsto} (v, x_1, x_2) * [x \overset{(\pi_1 \ominus \pi_3)}{\longmapsto} (v, x_1, x_2)]}\ \textsc{Msub}}{x \overset{\pi_1 \oplus (\pi_3 \ominus \pi_1)}{\longmapsto} (v, x_1, x_2) * (\pi_2 \oplus (\pi_3 \ominus \pi_2)) \cdot \mathsf{tree}(x_1) \rhd x \overset{\pi_3}{\longmapsto} (v, x_1, x_2) * \pi_3 \cdot \mathsf{tree}(x_1) * [x \overset{(\pi_1 \ominus \pi_3)}{\longmapsto} (v, x_1, x_2) * (\pi_2 \ominus \pi_3) \cdot \mathsf{tree}(x_1)]}\ \textsc{Psub}
$$

$$
\dfrac{}{x \overset{\pi_1 \oplus (\pi_3 \ominus \pi_1)}{\longmapsto} (v, x_1, x_2) * (\pi_2 \oplus (\pi_3 \ominus \pi_2)) \cdot \mathsf{tree}(x_1) * (x_2 = 0 \wedge \mathsf{emp}) \rhd x \overset{\pi_3}{\longmapsto} (v, x_1, x_2) * \pi_3 \cdot \mathsf{tree}(x_1) * \pi_3 \cdot \mathsf{tree}(x_2) * [x \overset{(\pi_1 \ominus \pi_3)}{\longmapsto} (v, x_1, x_2) * (\pi_2 \ominus \pi_3) \cdot \mathsf{tree}(x_1)]}\ \mathrm{F}_1'
$$

Figure 3.5: Frame inference

the transformation of predicate axioms into their fractional versions, the abduction problem for fractional heaps is inherently straightforward. For demonstration, consider the abduction together with its inference given in Figure 3.4. Here we explain the steps to construct the anti-frame $a = x_1 \wedge (\pi_3 \ominus \pi_2) \cdot \mathsf{tree}(x_1) * x \overset{\pi_3 \ominus \pi_1}{\longmapsto} (v, a, x_2)$. First, we start with the base axiom that asserts the empty heap on both sides and let $x_2 = 0$. Next, we apply the folding rule $\mathsf{F_1}'$ to infer $\pi_3 \cdot \mathsf{tree}(x_2)$ on the right hand side. For the subsequent two steps, we invoke two rules Psub and Msub to compute the fractional residues inside the anti-frame when filling the antecedent and consequent. As the frame will be constructed in the next subroutine, all the current fractional residues in the consequent are dropped off. Finally, we unify the variable a with x_1 and apply the folding rule $\mathsf{F_2}'$ to complete the consequent.

3.2.4 Frame inference

The task of frame inference is to compute the residue in the consequent. In particular, given the entailment $A \vdash B$, we would like to find the residue frame F_f such that:

$$A \dashv\vdash B * F_f.$$

This task is more pleasant than abduction because the frame F_f can be sufficiently deduced from A. As the frame F_f vitally prevents resource loss, verification tools [CDNQ12, DPJ08, BCO06] essentially have different techniques for their own frame inference mechanism. In principle, tools progressively try to match resources from consequent B with the ones in the antecedent A. Once B is completely matched, the remaining proportion in A is returned as the residue frame. Similar to abduction, a set of predicate axioms/facts is used to deal with predicate inference. Smoothly, such tools can upgrade to fractional reasoning by simply replacing the axiom set with the its fractional version and using two subtraction rules MSUB and PSUB to induce the fractional residue frame. For instance, we will show how to compute the frame from the previous abduction example (see Figure 3.5 for more details):

$$x \overset{\pi_1}{\longmapsto} (v, a, x_2) * \pi_2 \cdot \mathsf{tree}(x_1) * (x_2 = 0 \wedge \mathsf{emp}) * (a = x_1 \wedge (\pi_3 \ominus \pi_2) \cdot \mathsf{tree}(x_1)$$

$$* \ x \overset{\pi_3 \ominus \pi_1}{\longmapsto} (v, a, x_2)) \rhd \pi_3 \cdot \mathsf{tree}(x) * [??].$$

First, we simplify the antecedent by grouping same-address permissions and substituting $a = x_1$:

$$x \overset{\pi_1 \oplus (\pi_3 \ominus \pi_1)}{\longmapsto} (v, x_1, x_2) * (\pi_2 \oplus (\pi_3 \ominus \pi_2)) \cdot \mathsf{tree}(x_1) * (x_2 = 0 \wedge \mathsf{emp}) \rhd \pi_3 \cdot \mathsf{tree}(x) * [??].$$

Also, we apply the unfolding rule U' on the consequent to expose the matching pattern:

$$x \overset{\pi_1 \oplus (\pi_3 \ominus \pi_1)}{\longmapsto} (v, x_1, x_2) * (\pi_2 \oplus (\pi_3 \ominus \pi_2)) \cdot \mathsf{tree}(x_1) * (x_2 = 0 \wedge \mathsf{emp}) \rhd$$

$$x \overset{\pi_3}{\longmapsto} (v, x_1, x_2) * \pi_3 \cdot \mathsf{tree}(x_1) * \pi_3 \cdot \mathsf{tree}(x_2) * [??].$$

After the pre-processing, we start the frame inference by repeatedly matching consequent with antecedent; the left-over resources are accumulated inside the frame [??]; see Figure 3.5. Hence, the residue frame from is computed as $x \xmapsto{(\pi_1 \ominus \pi_3)} (v, x_1, x_2) * (\pi_2 \ominus \pi_3) \cdot \mathsf{tree}(x_1)$. Similar to abduction, we initiate with a base axiom that asserts both hand sides are emp. We then start the matching between two sides while using two rules MSUB and PSUB to compute the residues in the frame. In the last step, we apply folding rule $\mathsf{F_1}'$ to match $\pi_3 \cdot \mathsf{tree}(x_2)$ with $x_2 = 0 \wedge \mathsf{emp}$.

3.3 A proof theory for fractional permissions

In this section we examine predicate multiplication, and fractional separation logic more generally, from a proof theory perspective. In §3.4 we will develop a model (in the metalogic) to show that our logic is sound, but for now we do not assume a concrete model for our object logic. That is, in §3.3 all of our proofs are carried out in the object logic using various inference rules such as:

$$\frac{}{P \vdash P}\ \vdash\textsc{Refl} \qquad \frac{P \vdash Q \quad P \vdash R}{P \vdash Q \wedge R}\ \wedge\textsc{Right} \qquad \frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R}\ *\!\!-\!\!*\textsc{Adj} \qquad \frac{\forall x.(P(x) \vdash Q)}{\exists x.P(x) \vdash Q}\ \exists\textsc{Left}$$

Our goal in §3.3 is thus to discuss the proof rules we provide and demonstrate their usefulness. Some of the theorems have somewhat delicate proofs, so all of them have been verified in Coq. Instead of showing complete proofs we will simply give sketches showing the key intermediate points.

First we will give the ingredients of our base logic in §3.3.1, including a brief review of modal logic. Then we will discuss our new proof rules for predicate multiplication and fractional maps-tos (§3.3.2), precision (§3.3.3), and induction over fractional heaps (§3.3.4). The new rules are contained in a series of figures on page 62. We conclude (§3.3.5) with two examples that show that our proof theory is strong enough to prove real properties: that $\mathsf{tree}(x)$ is $\mathcal{F}$-uniform and that $\mathsf{list}(x)$ is precise.

3.3.1 Base logic

Our logic starts with the following base operators as follows:

$$P \stackrel{\text{def}}{=} |P| \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \mathsf{emp} \mid * \mid -\!\!* \mid \forall x.P(x) \mid \exists x.P(x) \mid \mu X.P. \quad (3.2)$$

The intended meaning of most of these operators is entirely standard: conjunction $\wedge$, disjunction $\vee$, implication $\Rightarrow$, separating identity emp, separating conjunction $*$, separating implication $-\!\!*$, universal $\forall$ and existential $\exists$ quantification. We allow covariant equirecursive predicates μ via the standard Knaster-Tarski fixpoint μ [Tar55], which allows us to build recursive definitions such as $\mathsf{tree}(x)$ as in equation (3.1) or the somewhat simpler $\mathsf{list}(x)$ predicate defined as

$$\mathsf{list}(x) \quad \dashv\vdash \quad \langle x = \mathsf{null} \rangle \vee \exists d, n.\ x \mapsto (d, n) * \mathsf{list}(n). \quad (3.3)$$

We can also inject any "pure" fact P in the metalogic into the object logic with $|P|$, and thus define $\top \stackrel{\text{def}}{=} |\top|$ and $\bot \stackrel{\text{def}}{=} |\bot|$. Note that $|P|$ does not restrict the separated resources, *e.g.* $|P| \dashv\vdash |P| \wedge \top$. Our main use of the metalogic in §3.3 is to conveniently manage routine proof details such as substitution of equalities, variable management, and $\bot$ elimination. At the cost of some additional hassle in the mechanized proofs we could add these kinds of routine features to the object logic if we wished to to avoid the metalogic entirely. We use the same symbols for operators at both the meta and object level. Our convention is that when we wish to use a meta-level symbol we will use parentheses to make sure it is out of scope from all $\vdash$ symbols; for example, in the $\exists$Left rule above, the $\forall$ above the line is a meta-level symbol whereas the $\exists$ below the line is an object-level symbol. The standard proof rules for the base symbols shown in equation (3.2) is given in Figure 3.6.

As we will see in §3.3.2, predicate multiplication has a strong connection with modal logic. We also use modal logic in §3.3.4 to structure induction over the finiteness of the heap within the object logic. Accordingly, we add the symbol families $\Box_R P$ and $\Diamond_R P$ to our logic, where R is an index into a family of modal operators. That is, we have a multimodal logic. In the Kripke model we show in §3.3.2, R will be a relation between worlds, and as usual $\Box_R$ will be

$$\frac{}{P \vdash P}\ \textsc{Refl} \qquad \frac{P \vdash Q \quad P \vdash R}{P \vdash Q \wedge R}\ \wedge\textsc{Right} \qquad \frac{P \vdash R}{P \wedge Q \vdash R}\ \wedge\textsc{Left1} \qquad \frac{Q \vdash R}{P \wedge Q \vdash R}\ \wedge\textsc{Left2}$$

$$\frac{P \vdash Q \quad Q \vdash R}{P \vdash R}\ \textsc{Trans} \qquad \frac{P \vdash R \quad Q \vdash R}{P \vee Q \vdash R}\ \vee\textsc{Left} \qquad \frac{P \vdash Q}{P \vdash Q \vee R}\ \vee\textsc{Right1} \qquad \frac{P \vdash R}{P \vdash Q \vee R}\ \vee\textsc{Right2}$$

$$\frac{P \vdash R \quad Q \vdash S}{P * Q \vdash R * S}\ \vdash* \qquad \frac{P \vdash Q \Rightarrow R}{P \wedge Q \vdash R}\ \wedge\Rightarrow\textsc{Adj} \qquad \frac{}{\top \vdash P \vee \neg P}\ \text{LEM} \qquad \frac{P \Rightarrow (\top \vdash Q)}{|P| \vdash Q}\ \textsc{Pure Left} \qquad \frac{Q}{P \vdash |Q|}\ \textsc{Pure Right}$$

$$\frac{}{P * \mathsf{emp} \dashv\vdash P}\ *\mathsf{emp} \qquad \frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R}\ *\!-\!\!*\textsc{Adj} \qquad \frac{}{P * Q \dashv\vdash Q * P}\ *\textsc{Comm} \qquad \frac{}{P * (Q * R) \dashv\vdash (P * Q) * R}\ *\textsc{Assoc}$$

$$\frac{P(c) \vdash Q}{\forall x.P(x) \vdash Q}\ \forall\textsc{Left} \qquad \frac{\forall x.(P \vdash Q(x))}{P \vdash \forall x.Q(x)}\ \forall\textsc{Right} \qquad \frac{\forall x.(P(x) \vdash Q)}{\exists x.P(x) \vdash Q}\ \exists\textsc{Left} \qquad \frac{P \vdash Q(c)}{P \vdash \exists x.Q(x)}\ \exists\textsc{Right}$$

$$\frac{\forall P, Q, x.(P(x) \vdash Q(x)) \Rightarrow (F(P)(x) \vdash F(Q)(x))}{\forall x.((\mu F)(x) \dashv\vdash F(\mu F)(x))}\ \mu\textsc{FoldUnfold} \qquad \frac{\forall x.(P(x) \dashv\vdash F(P)(x))}{\forall x.((\mu F)(x) \vdash Px)}\ \mu\textsc{LeastFixpoint}$$

Figure 3.6: Proof theory for separation logic with covariant recursion

$$\frac{\top \vdash P}{\top \vdash \Box_R P}\ \text{N} \qquad \frac{}{\Box_R(P \Rightarrow Q) \vdash (\Box_R P) \Rightarrow (\Box_R Q)}\ \text{K}$$

$$\frac{}{\forall x.\ \Box_R P(x) \vdash \Box_R\big(\forall x.P(x)\big)}\ \text{BF} \qquad \frac{}{\Diamond_R P \dashv\vdash \neg\Box_R\neg P}\ \Diamond\Box$$

Figure 3.7: Standard axioms for modal logic

a necessary/univeral modality whereas its dual $\Diamond_R$ will be a possibility/existential modality. Our proof theory for modal logic is divided into two parts. The general part consists of four rules in Fig. 3.7. These axioms are standard (including their names such as K). We assume a classical setting with axiom $\Diamond\Box$. In addition, various modal operators R satisfy additional rules such as the following:

$$\frac{}{\Box_R P \vdash P}\ \text{T} \qquad \frac{}{\Box_R P \vdash \Box_R\Box_R P}\ 4 \qquad \frac{}{\Box_R\Box_R P \vdash \Box_R P}\ \text{C4}$$

$$\frac{}{\Box_R P \vdash \Diamond_R P}\ \text{D} \qquad \frac{}{\Diamond_R P \vdash \Box_R P}\ \text{CD} \qquad \frac{\Box_R P \vdash P}{\top \vdash P}\ \text{W}$$

By the correspondence theory for modal logic, given a Kripke model for the logic, these axioms hold if and only if R is reflexive (T), transitive (4), dense (C4), total (D), functional (CD), and/or finitely chained (W). No single modal logic R satisfies all of the above axioms

(*e.g.*, D and W lead to a contradiction). Once one proves that a model satisfies characteristic modal axioms many lemmas follow "for free." For example, any CD modal logic satisfies $\Diamond_R(P \wedge Q) \dashv\vdash (\Diamond_R P) \wedge (\Diamond_R Q)$. As we are about to see, this is exactly the DotConj rule from Figure 3.2.

3.3.2 Proof theory for $\pi \cdot P$ and $x \overset{p}{\mapsto} y$

Predicate multiplication. In §3.1 we presented the "user view" of predicate multiplication in Figure 3.2. That is, we presented the rules that someone who wants to verify programs may find convenient. As we explained, a key selling point for such users is that essentially all of the rules are bidirectional. Counting each direction of the various rules separately, there are 23 such rules. For reasons of modularity it is convenient to consider DotMapsTo separately, leaving 21 rules. By finding a connection to modal logic, we can prove these 21 rules from the core set of 10 rules given in Figure 3.8 (again counting each direction as a separate goal). When we provide an abstract model in §3.4 for our logic, needing to prove a smaller rule set will simplify our task. Better still, the connection will give us insight into why certain properties are true—and why others are not.

To see why there is a modal connection, we will preview the underlying semantic model before putting it back on the shelf until §3.4.3. Recall that the informal meaning of $\pi \cdot P$ is that we have a π-fraction of predicate P. The formal semantics of this notion relies on a little trick:

$$h \models \pi \cdot P \quad \overset{\text{def}}{=} \quad \exists h'.\ mul(\pi, h') = \pi \wedge h' \models P. \tag{3.4}$$

A heap h contains a π-fraction of P if there exists a **bigger** heap h' that satisfies P, and when you "multiply" that bigger heap h' by the scalar π you get to the smaller heap h. By "*mul*" we mean the scalar multiplication of a heap by a share—intuitively, for each memory location x, multiplying π with the share π_x associated with x to reach $\pi \otimes \pi_x$. We will discuss *mul* further in §3.4.3.

Recall that the standard Kripke model for $\Box_R$ is $w \models \Box_R \overset{\text{def}}{=} \forall w'.wRw' \Rightarrow w' \models Q$ and for $\Diamond_R$ is $w \models \Diamond_R \overset{\text{def}}{=} \exists w'.wRw' \wedge w' \models Q$. Accordingly, predicate multiplication is exactly the

$$\frac{}{\pi \cdot P \vdash \pi \boxdot P}\ \text{CD} \qquad \frac{}{\mathcal{F} \boxdot P \vdash P}\ \text{T} \qquad \frac{}{\pi \boxdot (\pi' \boxdot P) \dashv\vdash (\pi \otimes \pi') \boxdot P}\ \text{MSeq}$$

$$\frac{}{\pi \cdot (|P| \wedge \mathsf{emp}) \dashv\vdash |P| \wedge \mathsf{emp}}\ \text{DotPure} \qquad \frac{}{(\pi_1 \oplus \pi_2) \cdot P \vdash (\pi_1 \cdot P) * (\pi_2 \cdot P)}\ \text{Dot Plus1}$$

$$\frac{P \vdash \mathsf{uniform}(\pi') \quad Q \vdash \mathsf{uniform}(\pi')}{\pi \cdot (P * Q) \dashv\vdash (\pi \cdot P) * (\pi \cdot Q)}\ \text{DotStar} \qquad \frac{\mathrm{precise}(P)}{(\pi_1 \cdot P) * (\pi_2 \cdot P) \vdash (\pi_1 \oplus \pi_2) \cdot P}\ \text{Dot Plus2}$$

Figure 3.8: Core proof theory for predicate multiplication

$$\frac{}{\mathsf{emp} \vdash \mathsf{uniform}(\pi)}\ \mathsf{uniform/emp} \qquad \frac{}{\mathsf{uniform}(\pi) * \mathsf{uniform}(\pi) \dashv\vdash \mathsf{uniform}(\pi)}\ \mathsf{uniform}*$$

$$\frac{P \vdash \mathsf{uniform}(\pi)}{\pi' \cdot P \vdash \mathsf{uniform}(\pi' \otimes \pi)}\ \mathsf{uniform Dot} \qquad \frac{\mathrm{precise}(P)}{\mathrm{precise}(\pi \cdot P)}\ \text{Dot Precise}$$

Figure 3.9: Uniformity and precision for predicate multiplication

$$\frac{}{\pi \cdot x \mapsto y \dashv\vdash x \overset{\pi}{\mapsto} y}\ \text{Dot MapsTo} \qquad \frac{}{x \overset{\pi}{\mapsto} y \vdash \mathsf{uniform}(\pi)}\ \overset{\mapsto}{\mathsf{uniform}} \qquad \frac{}{(x \overset{\pi}{\mapsto} y_1 * \top) \wedge (x \overset{\pi'}{\mapsto} y_2 * \top) \vdash |y_1 = y_2|}\ \overset{\mapsto}{\text{inversion}}$$

$$\frac{}{\mathrm{precise}(x \overset{\pi}{\mapsto} y)}\ \overset{\mapsto}{\text{precise}} \qquad \frac{}{x \overset{\pi}{\mapsto} y \vdash \neg\mathsf{emp}}\ \overset{\mapsto}{\mathsf{emp}} \qquad \frac{}{x \overset{\pi}{\mapsto} y \vdash |x \neq \mathtt{null}|}\ \overset{\mapsto}{\mathtt{null}}$$

Figure 3.10: Proof theory for fractional maps-to

$$\frac{\top \vdash \mathsf{precisely}(P)}{\mathrm{precise}(P)}\ \mathsf{precisely}\ \text{Precise} \qquad \frac{G \vdash \mathsf{precisely}(P) \quad G \vdash \mathsf{precisely}(Q)}{G \vdash \mathsf{precisely}(P * Q)}\ \mathsf{precisely}*$$

$$\frac{G \vdash \mathsf{precisely}(P)}{G \vdash \mathsf{precisely}(P \wedge Q)}\ \mathsf{precisely}\wedge \qquad \frac{\exists x.\big(G \vdash \mathsf{precisely}(P(x))\big)}{G \vdash \mathsf{precisely}(\forall x.P(x))}\ \mathsf{precisely}\forall$$

$$\frac{\begin{array}{c}G \vdash \mathsf{precisely}(P)\\ G \vdash \mathsf{precisely}(Q)\end{array}}{G \wedge (P * \top) \wedge (Q * \top) \vdash \bot \over G \vdash \mathsf{precisely}(P \vee Q)}\ \mathsf{precisely}\vee \qquad \frac{\forall x.\big(G \vdash \mathsf{precisely}(P(x))\big) \quad \forall x, y.\big(G \wedge (P(x) * \top) \wedge (P(y) * \top) \vdash |x = y|\big)}{G \vdash \mathsf{precisely}(\exists x.P(x))}\ \mathsf{precisely}\exists$$

$$\frac{}{\mathsf{precisely}(P) \vdash \big((P{*}Q) \wedge (P{*}R)\big) \Rightarrow (P{*}(Q \wedge R))}\ \mathsf{precisely}\ \text{Left} \qquad \frac{\forall Q, R.\big(G \vdash \big((P{*}Q) \wedge (P{*}R)\big) \Rightarrow (P{*}(Q \wedge R))\big)}{G \vdash \mathsf{precisely}(P)}\ \mathsf{precisely}\ \text{Right}$$

Figure 3.11: Proof theory for precision

$$\frac{}{\circledcirc P \vdash P}\ \text{T} \qquad \frac{}{\circledcirc P \vdash \circledcirc\circledcirc P}\ 4 \qquad \frac{}{\rhd_\pi P \vdash \rhd_\pi \rhd_\pi P}\ 4 \qquad \frac{\rhd_\pi P \vdash P}{\top \vdash P}\ \text{W} \qquad \frac{}{\rhd_\pi P \dashv\vdash \rhd_\pi \circledcirc P}\ \rhd_\pi\circledcirc \qquad \frac{}{\rhd_\pi P \dashv\vdash \circledcirc \rhd_\pi P}\ \circledcirc\rhd_\pi$$

$$\frac{}{(P * Q) \wedge \circledcirc R \vdash (P \wedge \circledcirc R) * (Q \wedge \circledcirc R)}\ \circledcirc* \qquad \frac{P \vdash U(\pi) \wedge \neg\mathsf{emp}}{(P * Q) \wedge \rhd_\pi R \vdash (P \wedge \rhd_\pi R) * (Q \wedge R)}\ \rhd_\pi*$$

Figure 3.12: Proof theory for substructural induction

modal $\Diamond$ using the relation "factor-π" F_π, which is defined as $hF_\pi h' \stackrel{\text{def}}{=} mul(\pi, h') = h$. That is, $\pi \cdot P \stackrel{\text{def}}{=} \Diamond_{F_\pi} P$. It is convenient to state some of the core predicate multiplication rules using its modal dual, written $\pi \boxdot P$ and defined as $\Box_{F_\pi} P$. Just as with any modal $\Box/\Diamond$ pair, $\boxdot$ and $\cdot$ satisfy N, K, BF, and $\Diamond\Box$.

We call the F_π relation "factor-π" since given an "input" h, the relation is defined with an output h' when h can have π "factored out of it" to reach h'. To make an analogy, the natural 15 is in the "factor-3" relation with 5. This analogy provides good intuition because the relation F_π is functional ($wF_\pi w' \Rightarrow wF_\pi w'' \Rightarrow w' = w''$) but not total (given an arbitrary π, not every w has a F_π-successor w', just as 15 does not have a successor in $\mathbb{N}$ for the "factor-7" relation). In other words, $\boxdot$ satisfies axiom CD but not axiom D. We will discuss axiom D further shortly. Once we prove axiom CD on the underlying semantic model, we then get 11 of the rules for predicate multiplication for free: DotPos, DotDisj, DotConj, DotImpl, DotNeg, DotUniv, and DotExis. Consequently, instead of considering the proof rules in Fig. 3.2, we now can focus our attention to the modal proof rules which are more general:

Theorem 3.3.1 ([Dev])**.** The proof rules in Fig. 3.2 are sound with respect to any fractional heap model that satisfies modal axioms in Fig. 3.7 and 3.8. ◁

In §3.5 we will see what goes wrong if we assume the underlying semantic model satisfies D for arbitrary π. For now we will just observe that the lack of axiom D has a few negative consequences for the logic. In particular, the rules DotImpl and DotNeg are one directional only ($\vdash$), and we need a minor side condition ($\tau \neq \emptyset$) for the $\dashv$ direction of DotUniv. In addition, we can satisfy the rule $\pi \cdot |P| \vdash |P|$, but not the rule $|P| \vdash \pi \cdot |P|$. To get bidirectionality our pure facts must force the empty heap, which is why we use $\langle P \rangle \stackrel{\text{def}}{=} |P| \wedge \mathsf{emp}$ in the DotPure rule.

There is, however, a special case: when $\pi = \mathcal{F}$, the multiplicative identity, then not only does $\boxdot$ satisfy D, it satisfies the stronger axiom T (reflexivity). In the analogy, all $n \in \mathbb{N}$ have a successor in the "factor-1" relation (*i.e.* n itself). Axiom T for $\mathcal{F}$ (plus CD for all π) is why DotFull holds. One final advantage of our modal setup is that we can prove the DotDot rule from a more general modal rule for combining boxy operators MSeq.

Our modal setup does not help us with the remaining 6 rules DotPure, DotPlus, and DotStar, which must be proved individually on the model. In Figure 3.8 we have shown the two directions of DotPlus separately so that it is clear that we only need precise(P) in the $\dashv$ direction.

Proving the side conditions for DotPlus and DotStar. To use predicate multiplication in practice we will need to prove two kinds of side conditions: that P is π-uniform (*i.e.* $P \vdash \mathsf{uniform}(\pi)$) and that P is precise. Figure 3.9 contains the three basic axioms (again deferring maps-to briefly) that allow us to prove uniformity for predicates such as list and tree during program verifications. They are all simple to state: uniform/emp tells us that emp is π-uniform for all π; the conclusion (all defined heap locations are held with share π) is vacuously true. The uniformDot rule tells us that if P is π-uniform then when we multiply P by a fraction π' the result is $(\pi' \otimes \pi)$-uniform.

The uniform$*$ rule is more interesting. The $\dashv$ direction follows from uniform/emp and the $*$emp rule ($P * \mathsf{emp} \dashv\vdash P$). The $\vdash$ direction is not automatic but very useful. One consequence is that from $P \vdash \mathsf{uniform}(\pi)$ and $Q \vdash \mathsf{uniform}(\pi)$ we can prove $P * Q \vdash \mathsf{uniform}(\pi)$. The $\vdash$ direction follows from disjointness. On non-disjoint share models such as $\mathbb{Q}$ it fails since *e.g.* $x \overset{0.3}{\mapsto} y * x \overset{0.3}{\mapsto} y \vdash x \overset{0.6}{\mapsto} y$; the two left maps-tos are 0.3-uniform whereas the right one is 0.6-uniform.

The DotPrecise rule is a partial solution to proving that a predicate is precise. It states that $\pi \cdot P$ is precise if and only if P is precise. In §3.3.3 we will show how to prove that P itself is precise.

Fractional maps-to. Figure 3.10 gives the proof theory we need to for our fractional maps-to $x \overset{\pi}{\mapsto} y$. The new rules in our context are the first three, the first of which (DotMapsTo) was already given in Figure 3.2 and the second of which ($\mapsto$ Uniform) is straightforward. The third ($\mapsto$ Inversion)is a little less obvious; it states that it's impossible to have two fractional maps-tos to the same address that point to different values. We need this fact to *e.g.* prove that predicates such as tree that contain existentials are precise. The remaining lemmas are all well-known facts: maps-to is precise (DotPrecise), non-empty (Dotemp),

and non-null (DotNull).

3.3.3 A proof theory for proving that predicates are precise

Proving that a predicate is π-uniform is relatively straightforward using the proof rules presented so far. However, proving that a predicate is precise is not as pleasant. Traditionally precision is defined (and checked for concrete predicates) in the metalogic [OHe07] using the following definition:

$$\text{precise}(P) \stackrel{\text{def}}{=} \forall h, h_1, h_2.\ h_1 \subseteq h \Rightarrow h_2 \subseteq h \Rightarrow (h_1 \models P) \Rightarrow (h_2 \models P) \Rightarrow h_1 = h_2. \quad (3.5)$$

Here we write $h_1 \subseteq h_2$ to mean that h_1 is a subheap of h_2, *i.e.* $\exists h'.h_1 \oplus h' = h_2$, where $\oplus$ is the joining operation on the underlying separation algebra [DHA09]. Essentially precision is a kind of uniqueness property: if a predicate P is precise then it can only be true on a single subheap.

Rather than checking precision in the metalogic, we wish to do so in the object logic. We give a proof theory that lets us do so in Figure 3.11. Among other advantages, proving precision in the object logic lets us to leverage the machinery for object-logic induction that we present in §3.3.4 to prove the precision of recursive predicates. The core idea is simple: we define a new object logic operator "$\mathsf{precisely}(P)$" that captures the notion of precision relativized to the current heap; essentially it is a partially applied version of the standard definition of precise(P) in equation (3.5):

$$h \models \mathsf{precisely}(P) \stackrel{\text{def}}{=} \forall h_1, h_2.h_1 \subseteq h \Rightarrow h_2 \subseteq h \Rightarrow (h_1 \models P) \Rightarrow (h_2 \models P) \Rightarrow h_1 = h_2. \quad (3.6)$$

Although we have given precisely's model to aid intuition, we emphasize that in §3.3 all of our proofs take place in the object logic; we never unfold precisely's definition. Although on first glace it may look modal, precisely is not in either its model (since it quantifies over two worlds, not one) or in its proof theory (*e.g.* it does not satisfy axiom N). It is also generally weaker than the typical notion of precision. For example, the predicate $x \mapsto 7 \vee y \mapsto 7$ is not precise; however the entailment $z \mapsto 8 \vdash \mathsf{precisely}(x \mapsto 7 \vee y \mapsto 7)$ is provable from

Figure 3.11 (and it is vacuously true in the model).

That said, the two notions are closely connected as given in the preciselyPRECISE rule. We also give introduction preciselyRIGHT and elimination rules preciselyLEFT that make a connection between precision and an "antidistribution" of $*$ over $\wedge$.

We also give a number of rules for showing how precisely combines with the connectives of our logic. The rules for propositional $\wedge$ and separating $*$ conjunction follow well-understood patterns, with the addition of an arbitrary premise context G being the key feature. The rule for disjunction $\vee$ is a little trickier, with an additional premise that forces the disjunction to be exclusive rather than inclusive. An example of such an exclusive disjunction is in the standard definition of the tree predicate, where the first disjunct $\langle x = \texttt{null} \rangle$ is fundamentally incompatible with the second disjunct $\exists d, l, r.x \mapsto d, l, r * \ldots$ since $\mapsto$ does not allow the address to be `null` (by axiom $\mapsto$ `null` from Figure 3.10). The rules for universal quantification $\forall$ existential quantification $\exists$ are essentially generalizations of the rules for the traditional conjunction $\wedge$ and disjunction $\vee$.

Using these lemmas, it is straightforward to prove the precision of simple predicates such as $\langle x = \texttt{null} \rangle \vee (\exists y.x \mapsto y * y \mapsto 0)$. However, as we will see, finding and proving the key lemmas that enable the proof of the precision of recursive predicates is a little subtle.

3.3.4 Proof theory for induction over the finiteness of the heap

Recursive predicates such as $\mathsf{list}(x)$ and $\mathsf{tree}(x)$ are extremely common in separation logic. However, proving properties of such predicates, such as proving that $\mathsf{list}(x)$ is precise, is a little tricky since the μFOLDUNFOLD rule provided by the Tarski fixed point does not automatically provide an induction principle. Generally speaking such properties follow by some kind of induction argument, either over auxiliary parameters* or over the finiteness of the heap itself. Both arguments usually occur in the metalogic rather than the object logic.

We have two contributions to make for proving inductive properties. First, we show how to do

e.g. if we augment trees to have the form $\mathsf{tree}(x, \tau)$, where τ is an inductively-defined type in the metalogic.

induction over the heap in a fractional setting. Intuitively this is more complicated than in the non-fractional case because there are infinite sequences of strictly smaller subheaps. That is, for a given initial heap h_0, there are infinite sequences $h_1, h_2, \ldots$ such that $h_0 \supsetneq h_1 \supsetneq h_2 \supsetneq \ldots$. The disjointness property does not fundamentally change this issue, so we illustrate with an example with the shares in $\mathbb{Q}$. The heap h_0 satisfying $x \overset{1}{\mapsto} y$ is strictly larger than the heap h_1 satisfying $x \overset{\frac{1}{2}}{\mapsto} y$, which is strictly larger than the heap h_2 satisfying $x \overset{\frac{1}{4}}{\mapsto} y$; in general h_i satisfies $x \overset{\frac{1}{2^i}}{\mapsto} y$. Since our sequence never terminates, we cannot use it as the basis for an induction argument. The solution is that we require that the heaps decrease by at least some constant size c. If each subsequent heap must shrink by at least *e.g.* $c = 0.25$ of a memory cell then eventually the sequence will terminate just as in the nonfractional case (which is actually just the case when $c = \mathcal{F}$).

Our second contribution is the development of a proof theory in the object logic that can carry out these kinds of induction proofs in a relatively straightforward way. The proof rules that let us do so are given in Figure 3.12. Once good lemmas are identified, we find doing induction proofs over the finite heap formally in the object logic simpler than doing the same proofs in the metalogic.

The key to our induction technique is two new modal operators: "within" $\circledcirc$ and "shrinking" $\rhd_\pi$. Essentially $\rhd_\pi P$ is used as an induction guard, preventing us from applying our induction hypothesis P until we are on a π-smaller subheap. When $\pi = \mathcal{F}$ we sometimes omit it, writing just $\rhd P$. In the Kripke model for the modal operator, the definition of the π-shrinking relation S_π is a little subtle*:

$$h_1 S_\pi h_4 \overset{\text{def}}{=} \exists h_2, h_3.\ h_1 \supseteq h_2 \wedge h_3 \oplus h_4 = h_2 \wedge (h_3 \models \mathsf{uniform}(\pi) \wedge \neg\mathsf{emp}). \tag{3.7}$$

The shrinking modality is just a modal box over the above relation $\rhd_\pi P \overset{\text{def}}{=} \Box_{S_\pi} P$. That is, if $h \models \rhd_\pi P$ then P is true **on all strict subheaps of h that are smaller by at least a π-piece**. Given this understanding, the key elimination rule for $\rhd_\pi$ (axiom $\rhd_\pi *$) may seem natural: essentially it is verifying that the induction hypothesis guarded by $\rhd_\pi$ is satisfied.

*As everywhere in §3.3, the model is for intuition only. All lemmas follow from the proof theory, not by unfolding the model.

To start an induction proof to prove an arbitrary goal $\top \vdash P$, we use the modal rule W to introduce an induction hypothesis, resulting in the new entailment goal of $\triangleright_\pi P \vdash P$.

Some definitions, such as $\mathsf{list}(x)$, have only one "recursive call"; others, such as $\mathsf{tree}(x)$ have more than one. Moreover, sometimes we wish to apply our inductive hypothesis immediately after satisfying the guard, whereas other times it is convenient to satisfy the guard somewhat before we need the inductive hypothesis. To handle both of these issues we use the "within" modality, defined as the boxy modality with relation $h_1 W h_2 \stackrel{\text{def}}{=} h_1 \supseteq h_2$. In other words, $h \models \circledcirc P$ means that P is true on all subheaps of h, which is the intuition behind the axiom $\circledcirc *$. Since the subheap relation is reflexive, "within" satisfies axiom T. Generally speaking if we wish to apply our induction hypothesis somewhat after meeting its guard (or if we wish to apply it more than once) we use the $\triangleright_\pi \circledcirc$ rule to add the $\circledcirc$ modality before eliminating the guard. We will see an example of this shortly.

3.3.5 Using our proof theory

We now turn to two examples of using our proof theory from page 62 to demonstrate that the axiom set is strong and flexible enough to prove real properties.

Proving that $\mathsf{tree}(x)$ is $\mathcal{F}$-uniform. Recall that a predicate is π-uniform if any fractional heap satisfies it must has permission π at every defined address. Our logical axioms for induction and uniformity are able to establish the uniformity of predicates in a fairly simple way. Here we focus on the $\mathsf{tree}(x)$ predicate because it is a little harder due to the two recursive "calls" in its unfolding.

Our initial proof goal is $\mathsf{tree}(x) \vdash \mathsf{uniform}(\mathcal{F})$. Standard natural deduction arguments then reach the goal $\top \vdash \forall x.\mathsf{tree}(x) \Rightarrow \mathsf{uniform}(\mathcal{F})$, after which we apply the W axiom ($\pi = \mathcal{F}$ is convenient) to start the induction, adding the hypothesis $\triangleright \forall x.\mathsf{tree}(x) \Rightarrow \mathsf{uniform}(\mathcal{F})$, which we strengthen with the $\triangleright_\pi \circledcirc$ rule to reach $\triangleright \circledcirc \forall x.\mathsf{tree}(x) \Rightarrow \mathsf{uniform}(\mathcal{F})$. Natural deduction from there reaches:

$$(\langle x = \mathtt{null} \rangle \vee \exists d, l, r. x \mapsto (d, l, r) * \mathsf{tree}(l) * \mathsf{tree}(r)) \wedge (\triangleright \circledcirc \forall x.\mathsf{tree}(x) \Rightarrow \mathsf{uniform}(\mathcal{F})) \vdash \mathsf{uniform}(\mathcal{F}).$$

The proof breaks into two cases. The first reduces to $\langle x = \texttt{null}\rangle \wedge (\triangleright \cdots) \vdash \mathsf{uniform}(\mathcal{F})$, which follows from the $\mathsf{uniform/emp}$ rule. The second case reduces to $\big(x \mapsto (d,l,r) * \mathsf{tree}(l) * \mathsf{tree}(r)\big) \wedge \big(\triangleright \circledcirc \forall x.\mathsf{tree}(x) \Rightarrow \mathsf{uniform}(\mathcal{F})\big) \vdash \mathsf{uniform}(\mathcal{F})$. Using the $\mathsf{uniform}*$ rule we can then reach:

$$\big(x \mapsto (d,l,r) * (\mathsf{tree}(l) * \mathsf{tree}(r))\big) \wedge \big(\triangleright \circledcirc \forall x.\mathsf{tree}(x) \Rightarrow \mathsf{uniform}(\mathcal{F})\big) \vdash \mathsf{uniform}(\mathcal{F}) * \mathsf{uniform}(\mathcal{F}).$$

We are now able to cut with the $\triangleright_\pi *$ rule to meet the inductive guard since $x \mapsto (d,l,r) \vdash \mathsf{uniform}(\mathcal{F}) \wedge \neg\mathsf{emp}$ due to the rules $\mapsto\mathsf{uniform}$ and $\mapsto\mathsf{emp}$. Our remaining goal is thus:

$$(x \mapsto (d,l,r) \wedge \triangleright \cdots) * \big((\mathsf{tree}(l) * \mathsf{tree}(r)) \wedge \circledcirc \forall x.\mathsf{tree}(x) \Rightarrow \mathsf{uniform}(\mathcal{F})\big) \vdash \mathsf{uniform}(\mathcal{F}) * \mathsf{uniform}(\mathcal{F}).$$

We split over $*$. The first goal is $x \mapsto (d,l,r) \wedge \triangleright \cdots \vdash \mathsf{uniform}(\mathcal{F})$, which follows from $\mapsto\mathsf{uniform}$. The second goal is $(\mathsf{tree}(l) * \mathsf{tree}(r)) \wedge \circledcirc \forall x.\mathsf{tree}(x) \Rightarrow \mathsf{uniform}(\mathcal{F})\big) \vdash \mathsf{uniform}(\mathcal{F})$. We apply $\circledcirc *$ to distribute the inductive hypothesis into the $*$, and $\mathsf{uniform}*$ to split the right hand side, yielding:

$$\big(\mathsf{tree}(l) \wedge \circledcirc \forall x.\mathsf{tree}(x) \Rightarrow \mathsf{uniform}(\mathcal{F})\big) * \big(\mathsf{tree}(r) \wedge \circledcirc \forall x.\mathsf{tree}(x) \Rightarrow \mathsf{uniform}(\mathcal{F})\big) \vdash \mathsf{uniform}(\mathcal{F}) * \mathsf{uniform}(\mathcal{F}).$$

We again split over $*$ to reach two essentially identical cases. We apply axiom T to remove the $\circledcirc$ and after standard manipulations reach *e.g.* $\forall x.\mathsf{tree}(x) \Rightarrow \mathsf{uniform}(\mathcal{F}) \vdash \mathsf{tree}(l) \Rightarrow \mathsf{uniform}(\mathcal{F})$, which is immediate. Further details on this proof can be found in Figure 3.13 in which $H \stackrel{\text{def}}{=} \forall t.\ \mathsf{tree}(t) \Rightarrow U(\mathcal{F})$. Here we also use the following shortcuts for convenience: US $\stackrel{\text{def}}{=} \mathsf{uniform}*$, UE $\stackrel{\text{def}}{=} \mathsf{uniform/emp}$, UM $\stackrel{\text{def}}{=}\mapsto \mathsf{uniform}$, ME $\stackrel{\text{def}}{=}\mapsto \mathsf{emp}$, $U(\pi) \stackrel{\text{def}}{=} \mathsf{uniform}(\pi)$.

Comment on completeness. Although our axioms for induction and uniformity are powerful enough to be useful, they are not complete. It is hard to prove the following predicate is $\mathcal{F}$-uniform*

$$\mathsf{tricky}(x) \quad \dashv\vdash \quad x \overset{0.5}{\mapsto} 0 * 0.5 \cdot \mathsf{tricky}(x).$$

*To help intuition this example uses shares in $\mathbb{Q}$; similar games can occur in disjoint share models.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{emp} \vdash U(\mathcal{F})}\ \text{UE}}{\langle x = \mathtt{null}\rangle \wedge \rhd_{\mathcal{F}} \circledcirc H \vdash U(\mathcal{F})} \cdots \quad \dfrac{\dfrac{}{x \mapsto (d, t_1, t_2) \vdash U(\mathcal{F}) \wedge \neg\mathsf{emp}}\ \begin{matrix}\text{UM+}\\ \text{ME}\end{matrix} \quad \dfrac{}{(\mathsf{tree}(t_1) * \mathsf{tree}(t_2)) \wedge \circledcirc H \vdash U(\mathcal{F})}\ \begin{matrix}\text{L}_3\\ \text{L}_2\end{matrix}}{x \mapsto (d, t_1, t_2) * (\mathsf{tree}(t_1) * \mathsf{tree}(t_2)) \wedge \rhd_{\mathcal{F}} \circledcirc H \vdash U(\mathcal{F})} \cdots}{\dfrac{\mathsf{tree}(x) \wedge \rhd_{\mathcal{F}} \circledcirc H \vdash U(\mathcal{F})}{\mathsf{tree}(x) \vdash U(\mathcal{F})}\ \text{L}_1}$$

Main proof

$$\dfrac{\dfrac{\mathsf{tree}(x) \wedge \rhd_{\mathcal{F}} \circledcirc H \vdash U(\mathcal{F})}{\rhd_{\mathcal{F}} \circledcirc H \vdash H}\ \rhd_\pi\circledcirc}{\rhd_{\mathcal{F}} H \vdash H}\ \text{W}}{\vdash H} \cdots}{\mathsf{tree}(x) \vdash U(\mathcal{F})}$$

Proof of L_1

$$\dfrac{\dfrac{\dfrac{\dfrac{}{H \vdash \mathsf{tree}(t_i) \Rightarrow U(\mathcal{F})} \cdots}{\circledcirc H \vdash \mathsf{tree}(t_i) \Rightarrow U(\mathcal{F})}\ \text{T}}{\mathsf{tree}(t_i) \wedge \circledcirc H \vdash U(\mathcal{F}), i = 1, 2} \cdots}{(\mathsf{tree}(t_1) \wedge \circledcirc H) * (\mathsf{tree}(t_2) \wedge \circledcirc H) \vdash U(\mathcal{F}) * U(\mathcal{F})}\ \circledcirc*}{\dfrac{\mathsf{tree}(t_1) * \mathsf{tree}(t_2) \wedge \circledcirc H \vdash U(\mathcal{F}) * U(\mathcal{F})}{\mathsf{tree}(t_1) * \mathsf{tree}(t_2) \wedge \circledcirc H \vdash U(\mathcal{F})}\ \text{US}} \cdots$$

Proof of L_3

$$\dfrac{\dfrac{P \vdash U(\pi) \wedge \neg\mathsf{emp}}{(P * Q) \wedge \rhd_\pi R \vdash (P \wedge \rhd_\pi R) * (Q \wedge R)}\ \rhd_\pi* \quad \dfrac{\dfrac{\dfrac{P \vdash U(\pi) \wedge \neg\mathsf{emp}}{P \wedge \rhd_\pi R \vdash U(\pi)} \cdots \quad Q \wedge R \vdash U(\pi)}{(P \wedge \rhd_\pi R) * (Q \wedge R) \vdash U(\pi) * U(\pi)} \cdots}{(P \wedge \rhd_\pi R) * (Q \wedge R) \vdash U(\pi)}\ \text{US} \cdots}{(P * Q) \wedge \rhd_\pi R \vdash U(\pi)}$$

Proof of L_2

Figure 3.13: Proof that $\mathsf{tree}(x)$ is full-uniform

To do so we would need to add a notion of limits. Predicates like $\mathsf{tricky}(x)$ do not seem practical.

Proving that $\mathsf{list}(x)$ is precise. Precision is a more complex property than π-uniformness, so it is not surprising that it is harder to prove. Accordingly we will use the simpler predicate $\mathsf{list}(x)$ as an example; the additional trick we need to prove that $\mathsf{tree}(x)$ is precise are applications of the $\rhd_\pi\circledcirc$ and $\circledcirc*$ rules in the same manner as the proof that $\mathsf{tree}(x)$ is $\mathcal{F}$-uniform. We have proved that both $\mathsf{list}(x)$ and $\mathsf{tree}(x)$ are precise using our proof rules in Coq [Dev].

In Figure 3.14 we give four key lemmas used in our proof*. All four are derived (sometimes

*We abuse notation by reusing the inference rule format used to present axioms to present derived lemmas as well.

$$\frac{}{\mathsf{precisely}(P) \dashv\vdash (P * \top) \Rightarrow \mathsf{precisely}(P)}\ (\text{A}) \qquad \frac{\mathrm{precise}(P)}{P * \mathsf{precisely}(Q) \vdash \mathsf{precisely}(P * Q)}\ (\text{D})$$

$$\frac{\begin{array}{c} Q \wedge (R * \top) \vdash \mathsf{precisely}(R) \\ Q \wedge (S * \top) \vdash \mathsf{precisely}(S) \\ (R * \top) \wedge (S * \top) \vdash \bot \end{array}}{Q \wedge ((R \vee S) * \top) \vdash \mathsf{precisely}(R \vee S)}\ (\text{B}) \qquad \frac{\begin{array}{c} \forall x. \Big(Q \wedge (P(x) * \top) \vdash \mathsf{precisely}\big(P(x)\big)\Big) \\ \forall x, y. \Big((P(x) * \top) \wedge (P(y) * \top) \vdash |x = y|\Big) \end{array}}{Q \wedge \Big(\big(\exists x. P(x)\big) * \top\Big) \vdash \mathsf{precisely}\Big(\exists x. P(x)\Big)}\ (\text{C})$$

Figure 3.14: Key lemmas we use to prove recursive predicates precise

with a little cleverness) from the basic proof rules given in Figure 3.11. The basic structure of the proof is as follows. To prove $\mathrm{precise}(\mathsf{list}(x))$ we first use the $\mathsf{precisely}\textsc{Precise}$ rule to transform the goal into $\top \vdash \mathsf{precisely}(\mathsf{list}(x))$. We cannot immediately apply axiom W, however, since without a concrete $*$-separated conjunct **outside** the $\mathsf{precisely}$, we cannot dismiss the inductive guard with the $\triangleright_\pi *$ axiom. Accordingly, we next use lemma (A) and standard natural deduction to reach the goal $\top \vdash \forall x.(\mathsf{list}(x) * \top) \Rightarrow \mathsf{precisely}(\mathsf{list}(x))$, after which we apply axiom W with $\pi = \mathcal{F}$.

Afterwards we do some standard natural deduction steps yielding the goal

$$\Big(\triangleright\, \forall x.\big(\mathsf{list}(x) * \top\big) \Rightarrow \mathsf{precisely}\big(\mathsf{list}(x)\big)\Big) \wedge \Big(\big(\langle x = \mathtt{null}\rangle \vee \exists d, n. x \mapsto (d, n) * \mathsf{list}(n)\big) * \top\Big) \vdash$$
$$\mathsf{precisely}\big(\langle x = \mathtt{null}\rangle \vee \exists d, n. x \mapsto (d, n) * \mathsf{list}(n)\big).$$

We are now in a position to apply lemma (B) to break up the disjunction. We now have three goals. The first goal is that $\langle x = \mathtt{null}\rangle$ is precise, which follows from the fact that emp is precise, which in turn can be proved using the axiom $\mathsf{precisely}\textsc{Right}$. The third goal is that the two branches of the disjunction are mutually incompatible, which follows from $\langle x = \mathtt{null}\rangle$ being incompatible with maps-to using axiom $\mapsto \mathtt{null}$. The second (and last remaining) goal needs to use lemma (C) twice to break up the existentials. Two of the three new goals are to show that the two existentials are uniquely determined, which follow from

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\top \vdash \mathsf{precisely}(\mathsf{emp})}\ \textsc{Right}^{\mathsf{precisely}}}{\triangleright P \wedge (\langle x = \mathtt{null}\rangle * \top) \vdash \mathsf{precisely}(\langle x = \mathtt{null}\rangle)} \cdots \quad \dfrac{}{\triangleright P \wedge (\exists d, n.x \mapsto (d,n) * \mathsf{list}(n)) \vdash \mathsf{precisely}(\exists d, n.x \mapsto (d,n) * \mathsf{list}(n))}\ \mathrm{L}_1 \quad \dfrac{}{(\langle x = \mathtt{null}\rangle * \top) \wedge (\exists d, n.x \mapsto (d,n) * \mathsf{list}(n) * \top) \vdash \bot}\ \overset{\mapsto}{\mathtt{null}}}{\triangleright P \wedge \Big((\langle x = \mathtt{null}\rangle \vee \exists d, n.x \mapsto (d,n) * \mathsf{list}(n)) * \top\Big) \vdash \mathsf{precisely}(\langle x = \mathtt{null}\rangle \vee \exists d, n.x \mapsto (d,n) * \mathsf{list}(n))}\ (\mathrm{B}) \cdots}{\dfrac{\triangleright \forall x.(\mathsf{list}(x) * \top \Rightarrow \mathsf{precisely}(\mathsf{list}(x))) \vdash \forall x.(\mathsf{list}(x) * \top \Rightarrow \mathsf{precisely}(\mathsf{list}(x)))}{\top \vdash \forall x.(\mathsf{list}(x) * \top \Rightarrow \mathsf{precisely}(\mathsf{list}(x)))}\ \mathrm{W}}\ (\mathrm{A}) \cdots$$

$$\dfrac{\mathsf{precisely}(\mathsf{list}(x))}{\mathrm{precise}(\mathsf{list}(x))}\ \mathsf{precisely}\textsc{Precise}$$

Main proof

$$\dfrac{\dfrac{\dfrac{}{\triangleright P \wedge (x \mapsto (d,n) * \mathsf{list}(n) * \top) \vdash \mathsf{precisely}(x \mapsto (d,n) * \mathsf{list}(n))}\ \mathrm{L}_2 \quad \dfrac{}{(x \mapsto (d,n_1) * \mathsf{list}(n_1) * \top) \wedge (x \mapsto (d,n_2) * \mathsf{list}(n_2) * \top) \vdash |n_1 = n_2|}\ \mapsto \mathrm{I}}{\triangleright P \wedge (\exists n.x \mapsto (d,n) * \mathsf{list}(n) * \top) \vdash \mathsf{precisely}(\exists n.x \mapsto (d,n) * \mathsf{list}(n))}\ (\mathrm{C}) \quad \dfrac{}{(\exists n.x \mapsto (d_1,n) * \mathsf{list}(n) * \top) \wedge (\exists n.x \mapsto (d_2,n) * \mathsf{list}(n) * \top) \vdash |d_1 = d_2|}\ \mapsto \mathrm{I}}{\triangleright P \wedge (\exists d, n.x \mapsto (d,n) * \mathsf{list}(n)) \vdash \mathsf{precisely}(\exists d, n.x \mapsto (d,n) * \mathsf{list}(n))}\ (\mathrm{C})$$

Proof of L_1

$$\dfrac{\dfrac{}{\triangleright P \wedge (x \mapsto (d,n) * \mathsf{list}(n) * \top) \vdash x \mapsto (d,n) * \mathsf{precisely}(\mathsf{list}(n))}\ \triangleright_\pi * \quad \dfrac{}{\mathrm{precise}(x \mapsto (d,n))}\ \mapsto \textsc{Precise}}{\triangleright P \wedge (x \mapsto (d,n) * \mathsf{list}(n) * \top) \vdash \mathsf{precisely}(x \mapsto (d,n) * \mathsf{list}(n))}\ (\mathrm{D})$$

Proof of L_2

Figure 3.15: Proof that $\mathsf{list}(x)$ is precise.

$\mapsto$INVERSION, leaving the goal

$$\Big(\triangleright\ \forall x.\big(\mathsf{list}(x) * \top\big) \Rightarrow \mathsf{precisely}\big(\mathsf{list}(x)\big)\Big) \wedge \Big(x \mapsto (d,n) * \big(\mathsf{list}(n) * \top\big)\Big) \vdash \mathsf{precisely}\Big(x \mapsto (d,n) * \mathsf{list}(n)\Big).$$

We now cut with lemma (D), using axiom $\mapsto$PRECISE to prove its premise, yielding the goal

$$\Big(\triangleright\ \forall x.\big(\mathsf{list}(x) * \top\big) \Rightarrow \mathsf{precisely}\big(\mathsf{list}(x)\big)\Big) \wedge \Big(x \mapsto (d,n) * \big(\mathsf{list}(n) * \top\big)\Big) \vdash x \mapsto (d,n) * \mathsf{precisely}\Big(\mathsf{list}(n)\Big).$$

We can now use the $\triangleright_\pi *$ axiom to defeat the inductive guard. The rest of the proof is straightforward. Details of the above proof can be found in Figure. 3.15 in which $\mapsto$ I is shortcut for the rule $\mapsto$ INVERSION.

3.4 Soundness proof: Building a model for our logic

To justify the correctness of our proof theories on page 62, we will provide a heap model that satisfy them. We have verified in Coq [Dev] that the models we provide in §3.4 enable the proof theory explained and demonstrated in §3.3.

We present our models in several parts. In §3.4.1 we begin with a brief review of Cancellative Separation Algebras (CSA). In §3.4.2 we explain what we need from our fractional share models. In §3.4.3 we develop an extension to CSAs called "Scaling Separation Algebras" (SSA). In §3.4.4 we give some constructors for building more complex SSAs out of simpler ones, and apply this technique to generate a simple concrete model for our logic. In §3.4.5 we develop the machinery necessary to support our rules for object-level induction over heap.

3.4.1 Cancellative separation algebras

A Separation Algebra (SA) is a set H with an associative, commutative partial operation $\oplus$. Separation algebras can have a single unit or multiple units; we use $\mathit{identity}(x)$ to indicate that x is a unit. A Cancellative SA $\langle H, \oplus\rangle$ further requires that $a \oplus b_1 = c \Rightarrow a \oplus b_2 = c \Rightarrow b_1 = b_2$. We can define a partial order on H using $\oplus$ by $h_1 \subseteq h_2 \stackrel{\text{def}}{=} \exists h'.h_1 \oplus h' = h_2$. Calcagno *et al.* [COY07] showed that CSAs can model separation logic via $h \models P * Q \stackrel{\text{def}}{=} \exists h_1, h_2.h_1 \oplus h_2 = h \wedge (h_1 \models P) \wedge (h_2 \models Q)$, $h \models \mathsf{emp} \stackrel{\text{def}}{=} \mathit{identity}(h)$, etc., from which our base proof theory from §3.3.1 follows.

If R is a binary relation over H, then the Kripke model for $\Box_R$ is $h \models \Box_R P \stackrel{\text{def}}{=} \forall h'.hRh' \Rightarrow h' \models P$ and for $\Diamond_R$ is $h \models \Diamond_R P \stackrel{\text{def}}{=} \exists h'.hRh' \wedge h' \models P$. The modal axioms N, K, BF, and $\Diamond\Box$ are immediate.

The standard definition of precise(P) was given as equation (3.5) in §3.3.3, together with the definition for our new $\mathsf{precisely}(P)$ operator in equation (3.6). What is difficult here is finding a set of axioms (Figure 3.11) and derivable lemmas (*e.g.* Figure 3.14) that are strong enough to be useful in the object-level inductive proofs. Once the axioms are found, proving them from the model given is straightforward. Cancellation is not necessary to model basic separation logic [DYBG+13], but we need it to prove the introduction $\mathsf{precisely}$RIGHT and

elimination rules preciselyLEFT for our new operator precisely.

3.4.2 Fractional share algebras

A fractional share algebra $\langle S, \oplus, \otimes, \mathcal{E}, \mathcal{F} \rangle$ is a set S with two operations: partial addition $\oplus$ and total multiplication $\otimes$. The substructure $\langle S, \oplus \rangle$ is a CSA with the single unit $\mathcal{E}$. For the reasons discussed in 3.5 we require that $\oplus$ satisfies the disjointness axiom $a \oplus a = b \Rightarrow a = \mathcal{E}$. Furthermore, we require that the existence of a top element $\mathcal{F}$, representing complete ownership, and assume that each element $s \in S$ has a complement $\overline{s}$ such that $s \oplus \overline{s} = \mathcal{F}$.

Often (*e.g.* in the fractional $\mapsto$ operator) we wish to restrict ourselves to the "positive shares" $S^+ \stackrel{\text{def}}{=} S \setminus \{\mathcal{E}\}$. To emphasize that a share is positive we often use the metavariable π rather than s. $\oplus$ is still associative, commutative, and cancellative; every element other than $\mathcal{F}$still has a complement. To enjoy a partial order on S^+ and other SA- or CSA-like structures that lack identities (sometimes called "permission algebras") we define $\pi_1 \subseteq \pi_2 \stackrel{\text{def}}{=} (\exists \pi'. \pi_1 \oplus \pi' = \pi_2) \vee (\pi_1 = \pi_2)$.

For the multiplicative structure we require that $\langle S, \otimes, \mathcal{F} \rangle$ be a monoid, *i.e.* that $\otimes$ is associative and has identity $\mathcal{F}$. Since we restrict maps-tos and the permission scaling operator to be positive, we want $\langle S^+, \otimes, \mathcal{F} \rangle$ to be a submonoid. Accordingly, when $\{\pi_1, \pi_2\} \subset S^+$, we require that $\pi_1 \otimes \pi_2 \neq \mathcal{E}$. Finally, we require that $\otimes$ distributes over $\oplus$ on the right, that is $(s_1 \oplus s_2) \otimes s_3 = (s_1 \otimes s_3) \oplus (s_2 \otimes s_3)$; and that $\otimes$ is cancellative on the right given a positive left multiplicand, *i.e.* $\pi \otimes s_1 = \pi \otimes s_2 \Rightarrow s_1 = s_2$.

The tree share model we present in §3.6.2 satisfies all of the above axioms, so we have a nontrivial model. As we will see shortly, it would be very convenient if we could assume that $\otimes$ also distributed on the left, or if we had multiplicative inverses on the left rather than merely cancelation on the right. However, we will see in §3.5.2 that both assumptions are untenable.

S_1. $force(\pi, force(\pi', a)) = force(\pi, a)$

S_2. $force(\pi, mul(\pi', a)) = force(\pi, a)$

S_3. $mul(\pi, force(\pi', a)) = force(\pi \otimes_S \pi', a)$

S_4. $mul(\pi, mul(\pi', a)) = mul(\pi \otimes_S \pi', a)$

S_5. $identity(a) \Rightarrow force(\pi, a) = a$

S_6. $a \subseteq_H force(\mathcal{F}, a)$

S_7. $\pi_1 \subseteq_S \pi_2 \Rightarrow force(\pi_1, a) \subseteq_H force(\pi_2, a)$

S_8. $force(\pi, a) \oplus_H force(\pi, b) = c \Rightarrow force(\pi, c) = c$

S_9. $identity(a) \Rightarrow mul(\pi, a) = a$

S_{10}. $mul(\mathcal{F}, a) = a$

S_{11}. $mul(\pi, a_1) = mul(\pi, a_2) \Rightarrow a_1 = a_2$

S_{12}. $mul(\pi, a) \subseteq_H a$

S_{13}. $\pi_1 \oplus_S \pi_2 = \pi_3 \Rightarrow \forall b, c. ((mul(\pi_1, b) \oplus_H mul(\pi_2, b) = c) \Leftrightarrow (c = mul(\pi_3, b)))$

S_{14}. $force(\pi', a) \oplus_H force(\pi', b) = force(\pi', c) \quad \Leftrightarrow$

$$mul(\pi, force(\pi', a)) \oplus_H mul(\pi, force(\pi', b)) = mul(\pi, force(\pi', c))$$

Figure 3.16: The 14 additional axioms for scaling separation algebras beyond those inherited from cancellative separation algebras

3.4.3 Scaling separation algebras

A scaling separation algebra is $\langle H, S, \oplus_H, \oplus_S, \otimes_S, \mathcal{E}, \mathcal{F}, mul, force \rangle$, where $\langle H, \oplus_H \rangle$ is a CSA and $\langle S, \oplus_S, \otimes_S, \mathcal{E}, \mathcal{F} \rangle$ is a fractional share algebra. Intuitively, $mul(\pi, h_1)$ multiplies every share inside h_1 by π and returns the resulting heap h_2. The multiplication occurs on the left, so for each original share π' in h_1, the resulting share in h_2 is becomes $\pi \otimes_S \pi'$. The even simpler $force(\pi, h_1)$ simply overwrites all shares in h_1 with the constant share π to reach the resulting heap h_2.

As explained in §3.3.2 we can define the "factor-π" relation F_π by $h_1 F_\pi h_2 \stackrel{\text{def}}{=} mul(\pi, h_2) = h_1$, and predicate multiplication $\pi \cdot P$ as $\Diamond_{F_\pi} P$. The dual operator $\pi \boxdot P$ is just $\Box_{F_\pi} P$. We use *force* to define the $\mathsf{uniform}$ predicate as $h \models \mathsf{uniform}(\pi) \stackrel{\text{def}}{=} force(\pi, h) = h$. A heap h is π-uniform when replacing all the shares in h with π gets you back to h—*i.e.*, they must have been π to begin with.

We need to understand how all of the ingredients in an SSA relate to each other to prove the core logical axioms in Figures 3.8 and 3.9. We distill the various relationships we need to model our logic in Figure 3.16. Although there are a goodly number of them, most are reasonably intuitive.

Axioms S_1 through S_4 straightforwardly describe how *force* and *mul* compose with each other. Axioms S_5, S_9, and S_{10} give conditions when *force* and *mul* are identity functions: when either is applied to identity/empty heaps, and when *mul* is applied to the multiplicative identity on shares $\mathcal{F}$. Axioms S_6 and S_{12} relate heap order with forcing the full share $\mathcal{F}$ and multiplication by an arbitrary share π. Axiom S_7 says that *force* is order-preserving. Axiom S_8 is how the disjointness axiom on shares is expressed on heaps: when two π-uniform heaps are joined, the result is π-uniform. Axiom S_{11} says that *mul* is injective over heaps. Axiom S_{13} is delicately stated. In the right-to-left direction of $\Leftrightarrow$, it states that *mul* preserves the share model's join structure on heaps. In the left-to-right direction, S_{13} is similar to axiom S_8, saying that the share model's join structure **must** be preserved. Taking both directions together, S_{13} translates the **right** distribution property of $\oplus_S$ over $\otimes_S$ into heaps. The final axiom S_{14} is a bit of a compromise. We wish we could satisfy:

$$S'_{14}. \qquad a \oplus_H b = c \quad \Leftrightarrow \quad mul(\pi, a) \oplus_H mul(\pi, b) = mul(\pi, c).$$

S'_{14} is a kind of dual for S_{13}, *i.e.* it would correspond to a **left** distributivity property of $\oplus_S$ over $\otimes_S$ in the share model into heaps. We also wish we could satisfy:

$$S'_{15} : \forall \pi, a. \exists b.\ mul(\pi, b) = a.$$

which would correspond to the existence of left multiplicative inverses on shares. Unfortunately, as we will see in §3.5.2, the disjointness of $\oplus_S$ is incompatible with simultaneously supporting both left and right distributivity; as well as with multiplicative inverses. Accordingly, S_{14} weakens S'_{14} so that it only holds when a and b are π'-uniform (which by S_8 forces c to be π'-uniform as well).

The axioms from Figure 3.16 are enough to prove all of the logical rules in Figures 3.8 and 3.9. uniform/emp follows from S_5 and uniform$*$ from S_8 and S_5 in the $\vdash$ and $\dashv$ directions, respectively. uniformDot follows from S_2, S_3, and S_{11}. Axiom CD also follows from S_{11}, T from S_{10}, MSeq from S_4, DotPure S_9, DotPlus1 from S_{13}, and the $\Uparrow$ direction of DotPrecise from S_{11} and S_{12}.

The remaining rules, DotPlus2, the $\Downarrow$ direction of DotPrecise, and both directions of DotStar, are harder to prove. The first two of those use S_2, S_6, and S_8 to find a heap large enough to contain two copies of a precise predicate; DotPlus2 also uses S_{13}. Both directions of DotStar follow from S_3, S_8, and S_{14}; recall that S_8 is how disjointness on the share model is translated into heaps.

If we could satisfy S'_{14}, we could remove the π'-uniform premises of DotStar; the proof is straightforward. S'_{15} would enable modal axiom D, *i.e.* $\pi \boxdot P \vdash \pi \cdot P$. In §3.3.2 we mentioned that D would yield some pleasant consequences such as a bidirectional DotImpl rule.

We establish the formal connection between SSAs and our proof theories by showing that the proposed axioms are sufficient to guarantee the soundness of the proof rules for predicate multiplication, precision and uniformity:

Theorem 3.4.1 ([Dev])**.** The proof theories in Fig. 3.8, 3.9, 3.10, 3.11 together with standard modal axioms N, K, BF, $\Diamond\Box$ in Fig. 3.7 are sound with respect to any fractional heap model that satisfies the SSA axioms, *i.e.* CSA axioms [COY07] and 14 axioms in Fig. 3.16. $\triangleleft$

3.4.4 Compositionality of scaling separation algebras

Despite their complex axiomatization, we gain two advantages from developing SSAs rather than directly proving our logical axioms on a concrete model. First, they give us a precise understanding of exactly which operations and properties (S_1–S_{14}) are used to prove the logical axioms. Second, following Dockins *et al.* [DHA09] we can build up large SSAs compositionally from smaller SSAs.

To do so cleanly it will be convenient to consider a slight variant of SSAs, "Weak SSAs" that allow, but do not require, the existence of identity elements in the underlying CSA model. A WSSA satisfies exactly the same axioms as an SSA, except that we use the weaker $\subseteq_H$ definition we defined for permission algebras, *i.e.* $a_1 \subseteq_H a_2 \stackrel{\text{def}}{=} (\exists a'. a_1 \oplus_H a' = a_2) \vee (a_1 = a_2)$. Note that S_5 and S_9 are vacuously true when the CSA does not have identity elements. We need identity elements to prove the logical axioms from the model; we only use WSSAs to gain compositionality as we construct a suitable final SSA. Keeping the share components $\langle S, \oplus_S, \otimes_S, \mathcal{E}, \mathcal{F} \rangle$ constant, we give three SSA constructors to get a flavor for what we can

do with the remaining components $\langle H, \oplus_H, \mathit{force}, \mathit{mul}\rangle$. For **convenience**, we will omit unchanged components/operators during the construction.

Example 3.4.1 (Shares)**.** The share model itself $\langle S, \oplus_S, \otimes_S\rangle$ is an SSA, and the positive (non-$\mathcal{E}$) shares $\langle S^+, \oplus_S, \otimes_S\rangle$ are a WSSA, with $\mathit{force}_S(\pi, \pi') \stackrel{\text{def}}{=} \pi$ and $\mathit{mul}_S(\pi, \pi') \stackrel{\text{def}}{=} \pi \otimes \pi'$. ◁

Example 3.4.2 (Semiproduct)**.** Let $\langle A, \oplus_A, \mathit{force}_A, \mathit{mul}_A\rangle$ be an SSA/WSSA, and B be a set. Define:

$$(a_1, b_1) \oplus_{A\times B} (a_2, b_2) = (a_3, b_3) \stackrel{\text{def}}{=} a_1 \oplus_A a_2 = a_3 \wedge b_1 = b_2 = b_3$$

$$\mathit{force}_{A\times B}(\pi, (a, b)) \stackrel{\text{def}}{=} (\mathit{force}_A(\pi, a), b) \qquad \mathit{mul}_{A\times B}(\pi, (a, b)) \stackrel{\text{def}}{=} (\mathit{mul}_A(\pi, a), b)$$

Then $\langle A \times B, \oplus_{A\times B}, \mathit{force}_{A\times B}, \mathit{mul}_{A\times B}\rangle$ is an SSA/WSSA. ◁

Example 3.4.3 (Finite partial map)**.** Let A be a set and $\langle B, \oplus_B, \mathit{force}_B, \mathit{mul}_B\rangle$ be an SSA/WSSA. Define $f \oplus_{A \stackrel{\text{fin}}{\rightharpoonup} B} g = h$ pointwise [DHA09]. Define:

$$\mathit{force}_{A \stackrel{\text{fin}}{\rightharpoonup} B}(\pi, f) \stackrel{\text{def}}{=} \lambda x.\mathit{force}_B(\pi, f(x)) \qquad \mathit{mul}_{A \stackrel{\text{fin}}{\rightharpoonup} B}(\pi, f) \stackrel{\text{def}}{=} \lambda x.\mathit{mul}_B(\pi, f(x))$$

The structure $\langle A \stackrel{\text{fin}}{\rightharpoonup} B, \oplus_{A \stackrel{\text{fin}}{\rightharpoonup} B}, \mathit{force}_{A \stackrel{\text{fin}}{\rightharpoonup} B}, \mathit{mul}_{A \stackrel{\text{fin}}{\rightharpoonup} B}\rangle$ is an SSA. ◁

Example 3.4.4 (Read-only permission)**.** Let $\langle A, \oplus_A, \otimes_S, \mathit{force}_A, \mathit{mul}_A\rangle$ be an SSA/WSSA and $r \notin A$ is a special *read-only* permission. Define $\langle A \cup \{r\}, \oplus_{A'}, \otimes_S, \mathit{force}_{A'}, \mathit{mul}_{A'}\rangle$ s.t.:

$$r \oplus_{A'} r \stackrel{\text{def}}{=} r \qquad r \oplus_{A'} \pi \text{ and } \pi \oplus_{A'} r \text{ are not defined for } \pi \in S$$

$$\mathit{force}_{A'}(\pi, r) \stackrel{\text{def}}{=} r \qquad \mathit{mul}_{A'}(\pi, r) \stackrel{\text{def}}{=} r$$

Then $\langle A \cup \{r\}, \oplus_{A'}, \otimes_S, \mathit{force}_{A'}, \mathit{mul}_{A'}\rangle$ is a new SSA/WSSA. ◁

Using the previous constructors it is immediate that $A \stackrel{\text{fin}}{\rightharpoonup} (S^+, V)$, that is finite partial maps from addresses to pairs of positive shares and values, is an SSA and thus can support a model for our logic. We can support other standard constructions such as sum types + as well.

Once we have a concrete model we prove the proof theory axioms about maps-to that we state in Figure 3.10. For concrete heaps all of these properties are easy to prove.

3.4.5 Model for inductive logic

What remains is to give the model that yields the inductive logic in Fig. 3.12. In §3.3.4 we gave the model for the key $\rhd_\pi$ inductive guard in equation (3.7). The model is a little subtle to enable the rules $\rhd_\pi\circledcirc$ and $\circledcirc\rhd_\pi$ that let us handle multiple recursive calls and simplify the engineering.

All of the rules follow from the definitions except for axiom W. To prove this axiom, we require that the heap model have an additional operator. The "π-quantum", written $|h|_\pi$, gives the number of times a non-empty π-sized piece can be taken out of h. For rational shares, $|h|_\pi$ is computed as the sum of floors of all permissions in h divided by π. For disjoint shares, the number of times is no more than the number of defined memory locations in h. We require two facts for $|h|_\pi$:

Q_1. First, that $h_1 \subseteq_H h_2 \Rightarrow |h_1|_\pi \leq |h_2|_\pi$, *i.e.* that subheaps do not have larger π-quanta than their parent.

Q_2. Second, that $h_1 \oplus_H h_2 = h_3 \Rightarrow (h_2 \models \mathsf{uniform}(\pi) \wedge \neg\mathsf{emp}) \Rightarrow |h_3|_\pi > |h_1|_\pi$, *i.e.* that taking out a π-piece strictly decreases the number of π-quanta.

Given this setup, axiom W follows immediately by induction on $|h|_\pi$. The axioms that require the longest proofs in the model are $\rhd_\pi\circledcirc$ and $\circledcirc\rhd_\pi$.

Theorem 3.4.2 ([Dev])**.** The induction proof theory in Fig. 3.12 is sound with respect to any SSA heap model that also satisfies Q_1 and Q_2. ◁

3.5 Lower bounds on predicate multiplication and disjoint shares

In §3.4 we showed that the logical axioms we presented on page 62 have a model (deferring until §3.6.2 the construction of the share model itself). In §3.3 we explained why those logical axioms in turn yield the proof rules for predicate multiplication discussed in §3.1.

Our goal for this section is to show that it is difficult to do better. Stated differently, §3.3 and §3.4 show what we can do; §3.5 discusses what we cannot do. We proceed in two parts.

First (§3.5.1) we explain how our key proof rules for predicate multiplication—and some stronger rules, such as a premise-free DotStar and bidirectional DotImpl—force properties on the share model. Second (§3.5.2) we show that disjointness puts meaningful restrictions on the class of share models. There are no nontrivial models that have left inverses or that satisfy both left and right distributivity.

3.5.1 Predicate multiplication's axioms force share model properties

The SSA structures we gave in §3.4.3, together with the connection to modal logic given in §3.3, are good for building models that enable the rules for predicate multiplication from Figure 3.2. However, since they impose intermediate algebraic and logical signatures between the concrete model and rules for predicate multiplication, they are not good for showing that we cannot do better. Accordingly here we disintermediate and focus on the concrete model $A \overset{\mathsf{fin}}{\rightharpoonup} (S^+, V)$, that is finite partial maps from addresses to pairs of positive shares and values. The join operations on heaps operates pointwise [DHA09], with $(\pi_1, v_1) \oplus (\pi_2, v_2) = (\pi_3, v_3) \overset{\text{def}}{=} \pi_1 \oplus_S \pi_2 = \pi_3 \wedge v_1 = v_2 = v_3$, from which we derive the usual SA model for $*$ and emp (§3.4.1). We define $h \models x \overset{\pi}{\mapsto} y \overset{\text{def}}{=} dom(h) = \{x\} \wedge h(x) = (\pi, y)$. We define scalar multiplication over heaps $\otimes_H$ pointwise as well, with $\pi_1 \otimes (\pi_2, v) \overset{\text{def}}{=} (\pi_1 \otimes_S \pi_2, v)$, and then define predicate multiplication by $h \models \pi \cdot P \overset{\text{def}}{=} \exists h'.\ h' = \pi \otimes_H h' = h \wedge h' \models P$. All of the above definitions are standard except for $\otimes_H$, which strikes us as the only choice (up to commutativity), and predicate multiplication itself, for which we have been unable to find any plausible alternative.

By §3.3 and §3.4 we already know that this model satisfies the rules for predicate multiplication, given the assumptions on the share model from §3.4.2. What is interesting is that we can prove the other direction: if we assume that the key logical rules from Figure 3.2 hold, they force axioms on the share model. The key correspondences are: DotFull forces that $\mathcal{F}$ is the left identity of $\otimes_S$; DotMapsTo forces that $\mathcal{F}$ is the right identity of $\otimes_S$; DotMapsTo forces the associativity of $\otimes_S$; the $\dashv$ direction of DotConj forces the right cancellativity of $\otimes_S$ (as does DotImpl and the $\dashv$ direction of DotUniv); and DotPlus, which forces right distributivity of $\otimes_S$ over $\oplus_S$. Additional details about these necessary

conditions can be found in Appendix A.1.

The following two rules force left distributivity of $\otimes_S$ over $\oplus_S$ and left $\otimes_S$ inverses, respectively:

$$\frac{}{\pi \cdot (P * Q) \dashv\vdash (\pi \cdot P) * (\pi \cdot Q)}\ \textsc{DotStar}' \qquad \frac{}{\pi \cdot (P \Rightarrow Q) \dashv (\pi \cdot P) \Rightarrow (\pi \cdot Q)}\ \textsc{DotImpl}'$$

The $\dashv$ direction of DotStar' also forces that $\oplus_S$ satisfies disjointness; this is the key reason that we cannot use $\langle Q, +, \times, 0, 1\rangle$ for our share model. Clearly the side-condition-free DotStar$'$ rule is preferable to the DotStar we give in Figure 3.2, and it would also be preferable to have bidirectionality for predicate multiplication over implication and negation. Unfortunately, as we will see shortly, the disjointness of $\oplus_S$ places strong multiplicative algebraic constraints on the share model. These constraints are the reason we cannot support the DotImpl$'$ rule and why we require the π'-uniformity side condition in our DotStar rule.

3.5.2 Disjointness in a multiplicative setting

Our goal now is to explore the algebraic consequences of the disjointness property in a multiplicative setting. Suppose $\langle S, \oplus\rangle$ is a CSA with a single unit $\mathcal{E}$, top element $\mathcal{F}$, and $\oplus$ complements $\overline{s}$. Suppose further that shares satisfy the disjointness property $a \oplus a = b \Rightarrow a = \mathcal{E}$. For the multiplicative structure, assume $\langle S, \otimes, \mathcal{F}\rangle$ is a monoid (*i.e.* the axioms forced by the DotDot, DotMapsTo, and DotFull rules). It is undesirable for a share model if multiplying two positive shares (*e.g.* the ability to read a memory cell) results in the empty permission, so we assume that when π_1 and π_2 are non-$\mathcal{E}$ then their product $\pi_1 \otimes \pi_2 \neq \mathcal{E}$.

Now add left or right distributivity. We choose right distributivity $(s_1 \oplus s_2) \otimes s_3 = (s_1 \otimes s_3) \oplus (s_2 \otimes s_3)$; the situation is mirrored with left. Let us show that we cannot have left inverses for $\pi \neq \mathcal{F}$. We prove by contradiction: suppose $\pi \neq \mathcal{F}$ and there exists π^{-1} such that $\pi^{-1} \otimes \pi = \mathcal{F}$. Then

$$\pi = \mathcal{F} \otimes \pi = (\pi^{-1} \oplus \overline{\pi^{-1}}) \otimes \pi = (\pi^{-1} \otimes \pi) \oplus (\overline{\pi^{-1}} \otimes \pi) = \mathcal{F} \oplus (\overline{\pi^{-1}} \otimes \pi).$$

Let $e = \overline{\pi^{-1}} \otimes \pi$. Now $\pi = \mathcal{F} \oplus e = (\overline{e} \oplus e) \oplus e$, which by associativity and disjointness forces $e = \mathcal{E}$, which in turn forces $\pi = \mathcal{F}$, a contradiction.

Now suppose that instead of adding multiplicative inverses we have both left and right distributivity. First we prove (lemma 1) that for arbitrary $s \in S$, $s \otimes \overline{s} = \overline{s} \otimes s$. We calculate:

$$(s \otimes s) \oplus (s \otimes \overline{s}) = s \otimes (s \oplus \overline{s}) = s \otimes \mathcal{F} = s = \mathcal{F} \otimes s = (s \oplus \overline{s}) \otimes s = (s \otimes s) \oplus (\overline{s} \otimes s).$$

Lemma 1 follows by the cancellativity of $\oplus$ between the far left and the far right.

Now we show (lemma 2) that $s \otimes \overline{s} = \mathcal{E}$. We calculate:

$$\mathcal{F} = \mathcal{F} \otimes \mathcal{F} = (s \oplus \overline{s}) \otimes (s \oplus \overline{s}) = (s \otimes s) \oplus (s \otimes \overline{s}) \oplus (\overline{s} \otimes s) \oplus (\overline{s} \otimes \overline{s}) = (s \otimes s) \oplus \underline{(s \otimes \overline{s}) \oplus (s \otimes \overline{s})} \oplus (\overline{s} \otimes \overline{s}).$$

The final equality is by lemma 1. The underlined portion implies $s \otimes \overline{s} = \mathcal{E}$ by disjointness.

The upshot of lemma 2, together with our requirement that the product of two positive shares be positive, is that we can have no more than the two elements $\mathcal{E}$ and $\mathcal{F}$ in our share model. Further details about the above proofs can be found in Appendix A.2.

Since the entire motivation for fractional share models is to allow ownership between $\mathcal{E}$ and $\mathcal{F}$, we must choose either left or right distributivity; we choose right since we are able to prove that the π'-uniformity side condition enables the bidirectional DotStar rule without left distributivity.

3.6 Share models

3.6.1 The shortcoming of rational permissions

The separating conjunction $*$ provides a convenient representation for disjointness when specifying assertion conditions. However, it would be mistaken to assume the disjointness property in fractional heaps with permissions in $(0, 1]$. We demonstrate this issue using the function createTree(Tree t1, Tree t2, int v) in Fig. 3.17 that creates a new binary tree with root value v, left subtree t1 and right subtree t2. If t1 and t2 are disjoint then in the standard

```
1 class Tree{Tree left,right; int value;}
2
3 Tree createTree (Tree t1, Tree t2, int v){
4   // {P}  Precondition that specifies two trees t1 and t2 are disjoint
5   Tree root = new Tree();
6   root.left = t1; root.right = t2; root.value = v;
7   // {Q}  Postcondition that specifies root is a tree
8   return root; }
```

Figure 3.17: A Java-like code that creates a binary trees from two disjoint trees

heap model, the precondition P can be precisely specified as $\mathsf{tree}(\mathsf{t1}) * \mathsf{tree}(\mathsf{t2})$. From there, it is straightforward to prove the postcondition Q as $\mathsf{tree}(\mathsf{root})$. Unfortunately, this proof cannot be generalized to rational permissions because we lose the disjointness property. For example, the predicate $0.25 \cdot \mathsf{tree}(\mathsf{t1}) * 0.25 \cdot \mathsf{tree}(\mathsf{t2})$ no longer captures the fact that $\mathsf{t1}, \mathsf{t2}$ are disjoint. Consequently, it is challenging to develop proof rules for automatic tools to deal with such circumstances.

Furthermore, the lack of disjointness in rational permissions significantly weakens the abductive inference when constructing the anti-frame. Consider the following abduction problem:

$$x \mapsto (v, x_1, x_2) * \mathsf{tree}(x_1) * [??] \vdash \mathsf{tree}(x).$$

Using the folding rule F_2, we can easily identify the anti-frame as $\mathsf{tree}(x_2)$. Now suppose we have a rational permission π distributed all over the two hand sides, *i.e.*:

$$x \overset{\pi}{\mapsto} (v, x_1, x_2) * \pi \cdot \mathsf{tree}(x_1) * [??] \vdash \pi \cdot \mathsf{tree}(x).$$

A naïve solution is to let the anti-frame be $\pi \cdot \mathsf{tree}(x_2)$. However, this entailment is false in general by the same reason for the precondition P in Fig. 3.17.

Last but not least, we recall the *overlap* operator $\uplus$ which was used in graph verification[HV13]. In detail, $h \models P \uplus Q$ if there exist disjoint heaps h_1, h_2, h_3 such that $h = h_1 \oplus h_2 \oplus h_3$, $h_1 \oplus h_2 \models P$ and $h_2 \oplus h_3 \models Q$. In [HV13], precision is one of the critical preconditions for proof rules and it was shown if P, Q are precise then $P \uplus Q$ is also precise. However, this property is lost in rational heaps, *e.g.*, $x \overset{0.6}{\longmapsto} 1 \uplus x \overset{0.6}{\longmapsto} 1$ is satisfied by any heap h s.t.

$\mathsf{dom}(h) = \{x\}$ and $h(x) = (1, \pi)$ for $1 \geq \pi \geq 0.6$.

3.6.2 The tree share model for fractional shares

Existing results on the tree share model of Dockins *et al.* [DHA09]. Here we recall several several key characteristics of the tree shares in §2.2.1. A tree share $\tau \in \mathbb{T}$ is a binary tree with Boolean leaves, *i.e.* $\tau = \bullet \mid \circ \mid \overset{\tau_1 \quad \tau_2}{\wedge}$, where $\circ$ is the empty share $\mathcal{E}$ and $\bullet$ is the full share $\mathcal{F}$. Trees must be in *canonical form*, *i.e.*, the most compact representation under the relation $\cong$:

$$\frac{}{\circ \cong \circ} \qquad \frac{}{\bullet \cong \bullet} \qquad \frac{}{\circ \cong \overset{\circ \quad \circ}{\wedge}} \qquad \frac{}{\bullet \cong \overset{\bullet \quad \bullet}{\wedge}} \qquad \frac{\tau_1 \cong \tau_1' \qquad \tau_2 \cong \tau_2'}{\overset{\tau_1 \quad \tau_2}{\wedge} \cong \overset{\tau_1' \quad \tau_2'}{\wedge}}$$

Tree shares are equipped with basic operators union $\sqcup$, intersection $\sqcap$ and complement $\bar{\cdot}$ that operate leafwise after unfolding the operands under $\cong$ into the same shape. Tree shares $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot}, \circ, \bullet \rangle$ form a countable atomless Boolean algebra and thus enjoy decidable existential and first-order theories with precisely known complexity bounds [LHL16]. The join operator $\oplus$ on trees is defined as $\tau_1 \oplus \tau_2 = \tau_3 \overset{\text{def}}{=} \tau_1 \sqcup \tau_2 = \tau_3 \wedge \tau_1 \sqcap \tau_2 = \circ$. Dockins *et al.* defined a notion of multiplication on trees that they wrote as $\tau_1 \bowtie \tau_2$; our notion of multiplication flips the operands: $\tau_1 \otimes \tau_2 \overset{\text{def}}{=} \tau_2 \bowtie \tau_1$, which makes our proof rules more pleasant to state. Multiplication $\otimes$ works by replacing all of the $\bullet$ leaves in τ_2 with copies of τ_1, *e.g.*, $\overset{}{\wedge} \otimes \overset{\circ \quad \bullet}{\wedge} = \overset{\overset{\circ \quad \bullet}{\wedge} \quad \circ}{\wedge}$. The structure $\langle \mathbb{T}, \otimes \rangle$ is isomorphic to word equations, implying a decidable existential and undecidable first-order theory. By restricting $\otimes$ to have a constant on the left-hand side, *i.e.* $\otimes_\tau(x) \overset{\text{def}}{=} \tau \otimes x$ then we recover a decidable first-order theory, even if we also admit the Boolean operators $\sqcup$, $\sqcap$, and $\bar{\cdot}$ [LHL16].

Our contribution to the tree share model. Our major contribution to the understanding of tree shares is proving that they have the axioms we require for share models (§3.4.2) and proving that they form an SSA (§3.4.4), thereby allowing them to be used with predicate multiplication. Their suitability as models has been verified in Coq:

Theorem 3.6.1 ([Dev])**.** The fractional heap model with tree share permissions is a SSA. ◁

We also show that in some important senses tree shares are difficult to improve upon algebraically (§3.5.2). We observe that the restriction to a family of unary multiplication operators—that is, to a decidable first-order subtheory—is acceptable for verification purposes because it allows us to use a share variable in our specifications, which we can then split into a constant number of pieces. Since each basic block of a program is of finite length, this is sufficient. Finally, in §3.2, we introduced the subtraction operator $\ominus$ to compute the residues in bi-abductive inference, which we can define as:

$$\pi_1 \ominus \pi_2 \stackrel{\text{def}}{=} \pi_1 \sqcap \overline{\pi_2}.$$

3.6.3 Applications of tree shares

While tree shares can be modeled for SSA, they require a certain amount of accounting to manipulate. Furthermore, different permission applications require different types of manipulations, *e.g.*, programs with semaphores/locks need token-counting permissions whereas fork/join programs need permissions for splitting and combining. As a result, we will show how to apply tree shares to reason various program types while help reduce the accounting nuisance by encapsulating tree shares into abstractions that are user-friendly yet still serve the desirable purposes.

Token-counting permissions Bornat *et al.* [BCOP05] introduced the token-counting permissions using integers $\mathcal{Z} = \langle \mathbb{Z}, + \rangle$ in which $x \overset{0}{\mapsto} v$ indicates the full permission over address x[*], $x \overset{n}{\mapsto} v$ denotes the *factory* permission[†] and $x \overset{-1}{\longmapsto} v$ is the read permission/token. A fresh address starts with the full/factory permission which can be subsequently split into a factory and write permission using the rule $x \overset{n}{\mapsto} v \dashv\vdash x \overset{n+1}{\longmapsto} v * x \overset{-1}{\longmapsto} v$. Such permissions are useful to reason about programs with semaphores such as readers-and-writers [CHP71] and pipeline processing [JK03] where read permissions to buffers are passed within the

[*]which is essential for memory allocation/deallocation.

[†]n also indicates the number of read permissions that have been generated.

thread pool. Here we model counting permissions using subsets of tree shares $\mathbb{F}$ for factory permissions and $\mathbb{R}$ for *read-only* permissions:

$$\mathbb{F} \stackrel{\text{def}}{=} \{f_i \mid f_0 = \bullet, f_{i+1} = \begin{array}{c}\wedge\\ \circ\ \bullet\end{array} \otimes f_i\} \qquad \mathbb{R} \stackrel{\text{def}}{=} \{r_i \mid r_i = \begin{array}{c}\wedge\\ \bullet\ \circ\end{array} \otimes f_i\}.$$

Consequently, a factory f_i can be split into a read token and another factory by the identities:

$$f_i \;=\; f_{i+1} \oplus r_i \qquad \frac{\text{precise}(P)}{f_i \cdot P \;\dashv\vdash\; (f_{i+1} \cdot P) * (r_i \cdot P)}\ \textsc{Split}$$

The first f_i and r_i are $f_0 = \bullet, f_1 = \begin{array}{c}\wedge\\ \circ\ \bullet\end{array}, f_2 = \begin{array}{c}\wedge\\ \circ\ \circ\ \bullet\end{array}, \ldots$ and $r_0 = \begin{array}{c}\wedge\\ \bullet\ \circ\end{array}, r_1 = \begin{array}{c}\wedge\\ \circ\ \bullet\ \circ\end{array}, \ldots$

Furthermore, the number of generated read tokens is encoded in the factory: exactly its height. For example, the height of f_2 is 2 and thus two read tokens are generated: r_0 and r_1.

Binary-string permissions While the previous read-only Ro provides a convenient abstraction over tree shares, its main shortcoming is that the original permission cannot be retrieved back once turned into read-only. As a result, it is inapplicable for programs with fork/join style [LCT15, JP11, HAZ08] where threads synchronize together with their resources and recover the write/deallocate permissions. We tackle this problem by proposing the binary-string structure $\mathcal{S} = \langle \{0,1\}^*, \oplus_s, \otimes_s \rangle$ that compromises between read-only permissions and tree shares. In detail, each binary string in $\mathcal{S}$ is essentially a tree share in $\mathcal{T}$ and thus $\mathcal{S}$ requires less accounting computation. On the other hand, it is more powerful than read-only permissions because all the inference rules are bi-directional.

To be precise, the empty string ϵ corresponds to full share $\bullet$; and if s is the tree τ then $0s$ and $1s$ correspond to $\begin{array}{c}\wedge\\ \bullet\ \circ\end{array} \otimes \tau$ and $\begin{array}{c}\wedge\\ \circ\ \bullet\end{array} \otimes \tau$. Also, addition $s_1 \oplus_s s_2 = s_3$ is defined if one of s_1, s_2 is $0s_3$ and the other is $1s_3$; while multiplication $s_1 \otimes_s s_2$ is simply string concatenation $s_1 \cdot s_2$. Consequently, the structure $\mathcal{S}$ is essentially a sub-structure of $\mathcal{T}$ with simpler representation and thus inherits properties of $\mathcal{T}$. Next, we define Ro_s with a binary string subscript as $h, \rho \models \mathsf{Ro}_s(P) \stackrel{\text{def}}{=} h, \rho \models s \cdot P$. Using scaling rules, we can derive the

following rules for our new read-only string permissions:

$$S_1 : P \dashv\vdash \mathsf{Ro}_\epsilon(P) \qquad S_2 : \mathsf{Ro}_{s_1}(\mathsf{Ro}_{s_2}(P)) \dashv\vdash \mathsf{Ro}_{s_1 \cdot s_2}(P)$$

$$S_3 : \mathrm{precise}(P) \Rightarrow (\mathsf{Ro}_{0s}(P) * \mathsf{Ro}_{1s}(P) \vdash \mathsf{Ro}_s(P)) \qquad S_4 : \mathsf{Ro}_s(P) \vdash \mathsf{Ro}_{0s}(P) * \mathsf{Ro}_{1s}(P)$$

$$S_5 : P_1, P_2 \vdash U(s') \Rightarrow (\mathsf{Ro}_s(P * Q) \dashv\vdash \mathsf{Ro}_s(P) * \mathsf{Ro}_s(Q)) \qquad S_6 : \mathsf{Ro}_s(P \wedge Q) \dashv\vdash \mathsf{Ro}_s(P) \wedge \mathsf{Ro}_s(Q)$$

$$S_7 : \mathsf{Ro}_s(P \vee Q) \dashv\vdash \mathsf{Ro}_s(P) \vee \mathsf{Ro}_s(Q) \qquad S_8 : \mathsf{Ro}_s(\exists v : \tau.\ P) \dashv\vdash \exists v.\ \mathsf{Ro}_s(P)$$

$$S_9 : \tau \neq \emptyset \Rightarrow (\mathsf{Ro}_s(\forall v : \tau.\ P) \dashv\vdash \forall v.\ \mathsf{Ro}_s(P))$$

3.7 The ShareInfer fractional biabduction engine

Having described our logical machinery in §3.1–§3.3, we now demonstrate that our techniques are well-suited to automation by documenting our ShareInfer prototype [Dev]. Our tool is capable of checking whether a user-defined recursive predicate such as list or tree is uniform and/or precise and then conducting biabductive inference over a separation logic entailment containing said predicates.

To check uniformity, the tool first uses heuristics to guess a potential tree share candidate π and then applies proof rules in Fig. 3.9 and 3.10 to derive the goal $\mathsf{uniform}(\pi)$. To support more flexibility, our tool also allows users to specify the candidate share π manually. To check precision, the tool maneuvers over the proof rules in Fig. 3.10 and 3.11 to achieve the desired goal. In both cases, recursive predicates are handled with the rules in Fig. 3.12. ShareInfer returns either Yes, No or Unknown together with a human-readable proof of its claim.

For bi-abduction, ShareInfer automatically checks precision and uniformity whenever it encounters a new recursive predicate. If the check returns Yes, the tool will unlock the corresponding rule, *i.e.*, DotPlus for precision and DotStar for uniformity. ShareInfer then matches fragments between the consequent and antecedent while applying folding and unfolding rules for recursive predicates to construct the antiframe and inference frame

Precision		Uniformity		Bi-abduction	
File name	Time (ms)	File name	Time (ms)	File name	Time (ms)
precise_map1	0.1	uni_map1	0.2	bi_map1	1.3
precise_map2	0.2	uni_map2	0.8	bi_map2	0.9
precise_map3	1.2	uni_map3	0.3	bi_map3	0.5
precise_list1	2.7	uni_list1	1.2	bi_list1	4.0
precise_list2	1.3	uni_list2	2.1	bi_list2	3.2
precise_list3	3.4	uni_list3	0.7	bi_list3	3.8
precise_tree1	1.4	uni_tree1	1.9	bi_tree1	5.1
precise_tree2	1.7	uni_tree2	1.0	bi_tree2	6.5
precise_tree3	12.2	uni_tree3	10.3	bi_tree3	7.9

Figure 3.18: Evaluation of our proof systems using ShareInfer

respectively. For instance, here is the biabduction problem contained in file bi_tree2 (see Fig. 3.18):

$$a \overset{\mathcal{F}}{\longmapsto} (b, c, d) \;\star\; \mathcal{L} \cdot \mathsf{tree}(c) \;\star\; \mathcal{R} \cdot \mathsf{tree}(d) \;\star\; [??] \;\vdash\; \mathcal{L} \cdot \mathsf{tree}(a) \;\star\; [??]$$

ShareInfer returns antiframe $\mathcal{L} \cdot \mathsf{tree}(d)$ and inference frame $a \overset{\mathcal{R}}{\longmapsto} (b, c, d) \star \mathcal{R} \cdot \mathsf{tree}(d)$.

ShareInfer is around 2.5k LOC of Java. We benchmarked it with 27 selective examples from three categories: precision, uniformity and bi-abduction. The benchmark was conducted with a 3.4 GHz processor and 16 GB of memory. Our results are given in Fig. 3.18. Despite the complexity of our proof rules our performance is reasonable: ShareInfer only took 75.9 milliseconds to run the entire example set, or around 2.8 milliseconds per example. Our benchmark is small, but this performance indicates that more sophisticated separation logic verifiers such as HIP/SLEEK [CDNQ12] or Infer [CDD+15] may be able to use our techniques at scale.

3.8 Related work and conclusion

Related work. Fractional permissions are essentially used to reason about resource ownership in concurrent programming. The well-known rational model $\langle [0,1], + \rangle$ by Boyland *et al.* [Boy03] is used to reason about join-fork programs. This structure has the disjointness problem which was first noticed by Bornat *et al.* [BCOP05], as well as other problems discussed in §3.1, §3.2, and §3.6.1. Boyland [Boy10] extended the framework to scale permissions uniformly over arbitrary predicates with multiplication, *e.g.*, he defined $\pi \cdot P$ as "multiply each permission π' in P with π". However, his framework cannot fit into SL and his scaling rules are not bi-directional. Jacobs and Piessens [JP11] also used rationals for scaling permissions $\pi \cdot P$ in SL but only obtained one direction for DotStar and DotPlus. A different kind of scaling permission was used by Dinsdale-Young *et al.* [DYDG+10] in which they used rationals to define permission assertions $[A]^r_\pi$ to indicate a thread with permission π can execute the action A over the shared region r.

Protocol-based logics like FCSL [NLWSD14] and Iris [JSS+15] have been very successful in reasoning about fine-grained concurrent programs, but their high expressivity results in a heavyweight logic. Automation (*e.g.* inference such as we do in §3.2) has been hard to come by. We believe that fractional permissions and protocol-based logics are in a meaningful sense complementary rather than competitors.

Conclusion. We proposed a modal proof framework in separation logic to reason about fractional permissions for resource sharing in concurrent programming. Our framework can support sophisticated verification tasks such as precision reasoning, inductive predicates and bi-abduction. We also developed scaling separation algebras as compositional models for our logic. We investigated why our logic cannot support certain desirable properties.

Chapter 4

Complete decision procedures for tree share constraints

> "*We all make choices, but in the end, our choices make us.*"
>
> Andrew Ryan, *Bioshock* (2007).

In this chapter, we will present two decision procedures over tree shares equipped with the join operator $\oplus$, namely the satisfiability problem (**SAT**) and implication problem (**IMP**). Our main motivation comes from the fact that verification tools such as HIP/SLEEK [NDQC07] and VST [App11b] actually use tree shares as permissions in their underlying logic and so automatic reasoning over tree shares constraints is essential to aid the overall verification process. To develop the decision procedures over $\langle \mathbb{T}, \oplus \rangle$, we need to establish some theoretical foundations over the structure. We discovered that tree shares with $\oplus$ satisfy the small model property that allows us to restrict the search space to be finite. As a result, our procedures are complete in the sense that they always return the correct answer for every **SAT** and **IMP** problem. Furthermore, we show that these problems can be equivalently reduced to Boolean formulas that can be handled by highly-optimized SMT solvers such as Z3 [dMB08] or MiniSat [ES03].

This chapter is organized as follows*:

1. In §4.1, we introduce the integration of tree shares into SL and demonstrate how to

*The materials in this chapter are taken from the paper "Decision Procedures Over Sophisticated Fractional Permissions" [LGH12], joint work with Cristian Gherghina and Aquinas Hobor. This work was submitted after my undergraduate study and before my PhD study in NUS.

extract the tree share constraints from SL proof. These constraints are represented by two query types **SAT** and **IMP** over tree shares.

2. In §4.2, we go though tee main components of our decision procedures for **SAT** and **IMP**. Along the way, we also justify their correctness.
3. In §4.3, we explain two key theoretical results that shape the correctness of our procedures.
4. In §4.4, we report on our experiment and result of the decision procedures in HIP/SLEEK.
5. In §4.5, we draw our conclusion.

Since the proofs of some lemmas are quite technical and tedious, we will explain the intuitions behind them together with illustrated examples instead. Their correctnesses are verified in Coq with appropriate pointer to the Coq development files.

4.1 Motivation: share constraints in SL formulas

In §4.1.1, we explain the extraction of share permissions from Separation Logic formulae. Then in 4.1.2 we define the types of permission constraints that our solver needs to handle.

4.1.1 Shares in HIP/SLEEK and their extraction procedure

Program verification tools, such as HIP, usually do not verify programs on their own. Instead, a verification tool usually applies Hoare rules to verify program commands and then emits the associated entailments to separate checkers such as SLEEK. Entailment checkers usually follow in the footsteps of SMT solvers by dividing the input formulae according to the background theories, and in turn rely on specialized provers for each theory, *e.g.* Omega for Presberger arithmetic.

We plan to follow the same pattern for fractional shares. The program verifier itself needs to know almost nothing about fractional shares, because it will simply emit entailments over formulas containing such shares to its entailment checker. The entailment checker needs to know a bit more: how to separate share information from formulas into a specialized

domain, *i.e.*, systems of equations over shares. The choice of this domain is an important modularity boundary because it allows the entailment prover to treat shares as an abstract type. The entailment checker only knows about certain basic operations such as equality testing, combining, and splitting shares. To check entailments over shares it calls our backend share prover (detailed in §4.2).

To demonstrate that the entailment checker can treat the shares abstractly, we will first outline the extraction of systems of equations over shares from separation logic formulas. Here we will just write χ for share constants; if our domain were rationals between 0 and 1, then an example χ would be 0.25. The tree share domain is more sophisticated but our point here is that extracting equations over shares can be done without actually knowing the underlying share model.

Entailment checkers are complicated, in part because information discovered in one sub-domain can impact another (*e.g.*, alias analysis can affect share constraints). Due to the tight link between heap-specific reasoning and share reasoning, extra share constraints are generated while discharging heap obligations. This information seepage prevents a modular and compositional description of the share constraint generation process. For brevity, we will illustrate share constraint extraction from a core separation logic; interested readers are referred to the description for a richer logic given in [HG12, §8.4]. Extracting share information from more complex formulas depends on the exact nature of said formulas but usually follows the pattern we give here in a straightforward way; the end result is just larger systems of equations.

The logic formulas we will consider here are of the form given in Figure 4.1. Here, v denotes variables (over shares, locations, and values) and $v \overset{\pi}{\mapsto} v$ is the fractional points-to predicate. Obtaining the share equation systems from the entailment $\Phi_a \vdash \Phi_c$ conceptually requires three steps.

First, the formulas are normalized in order to ensure that the heap component does not contain two distinct points-to predicates when the pointers are provably aliased. For example, give the constraint $v_1 \overset{\pi_1}{\mapsto} v_2 \ * \ v_3 \overset{\pi_2}{\mapsto} v_4 \ \wedge \ \beta$ and suppose we also know that β entails the equality between two addresses v_1, v_3, *i.e.*, $\beta \vdash v_1 = v_2$. Then the reduction step is applied

$$\Phi := \exists v.\ \kappa \wedge \beta \mid \kappa \wedge \beta \qquad \kappa := \kappa * \kappa \mid v \overset{\pi}{\mapsto} v$$
$$\beta := \beta \wedge \beta \mid v = \pi \mid \pi \oplus \pi = \pi \qquad \pi := v \mid \chi$$

Figure 4.1: SL formulae with shares

by combining two permissions π_1, π_2 and forcing the equality between v_2 and v_4:

$$v_1 \overset{\pi_1}{\mapsto} v_2 \ * \ v_3 \overset{\pi_2}{\mapsto} v_4 \ \wedge \ \beta \quad \xrightarrow{\beta \vdash v_1 = v_3} \quad \exists \pi_3.\ v_1 \overset{\pi_3}{\mapsto} v_2 \ \wedge \ (\beta \wedge \pi_3 = \pi_1 \oplus \pi_2 \wedge v_2 = v_4)$$

Second, formulas are partitioned based on the domains (*e.g.*, heaps, shares, arithmetics, bags) and all non heap related expressions are floated out of the heap fragment k. Share constants are floated out of the points-to relations by introducing a fresh share variable. Thus $v_1 \overset{\chi}{\mapsto} v_2$ becomes $\exists v'.\ v_1 \overset{v'}{\mapsto} v_2 \wedge v' = \chi$.

Third, heap-related obligations are discharged and any share constraint generated in the process is added to the share constraints previously extracted. SLEEK discharges heap constraints by pairing each points-to predicate $p_c \overset{s_c}{\mapsto} c_c$ in the consequent with a corresponding predicate in the antecedent $p_a \overset{s_a}{\mapsto} c_a$ when $p_a = p_c$. This pairing generates extra proof obligations over both the content of the memory ($c_a = c_c$) and the shares. For shares, SLEEK considers two possibilities: either the owned share s_a in the antecedent is equal to the one in the consequent ($s_a = s_c$), or s_a is strictly greater ($\exists s_r\ .\ s_a = s_c \oplus s_r$). This case split leads to the generation of two proof obligations, with the original entailment succeeding if at least one of the two new obligations is satisfied*.

Furthermore, it is common for separation logic entailment checkers to also infer a frame or residue—the part of the antecedent not required to prove the consequent. If s_a is larger than s_c, then there exists a non-empty share s_r such that $s_r \oplus s_c = s_a$. This share residue is captured by the instantiation of s_r.

*We are almost always able to avoid a serious exponential search by using the search prunings described in [HG12].

4.1.2 Problems over share equation system

After the heap constraints are discharged, the share relevant portion of the entailment consists of sets of formulas over **positive** shares (*i.e.* not $\circ$).

Definition 4.1.1 (Share constraint)**.** Our share constraints are the closure of $\oplus$-equations and equalities over conjunction $\wedge$ and existential quantifier $\exists$:

$$\phi \stackrel{\text{def}}{=} \exists v.\phi \mid \phi_1 \wedge \phi_2 \mid \pi_1 \oplus \pi_2 = \pi_3 \mid v_1 = v_2 \mid v = \chi$$

That is, share formulas ϕ contain share variables v, existential binders $\exists$, conjunctions $\wedge$, join facts $\oplus$, equalities between variables, and assignments of variables to constants χ. $\triangleleft$

Unless bound by an existential, variables are assumed to be universally bound, with universals bound before existentials ($\forall\exists$ rather than $\exists\forall$); despite implementing a translation for the feature-rich separation logic for SLEEK [HG12] we have not needed arbitrary nesting of quantifiers. For convenience, we represent the share formulas as equation systems of $\oplus$-equations and equalities together with a list of existential variables:

Definition 4.1.2. A tree share equation system Σ is a triple of three lists $(l^{\exists}, l^{=}, l^{\oplus})$ in which:

1. $l^{\exists}$ is the list of existential variables
2. $l^{=}$ is the list of equalities $v = w$
3. $l^{\oplus}$ is the list of $\oplus$-equations $a \oplus b = c$.

For **convenience**, we will represent a system of equation as $\Sigma = \{x_1, \ldots, x_n, e_1, \ldots, e_m\}$ in which x_i is existential variable and e_i is either $\oplus$-equation or equality. Furthermore, a *context* S of Σ is a (finite) mapping from the variables of Σ into tree shares. We say that S is a *solution* of Σ, written $S \models \Sigma$, when the mapping makes the equations and equalities in Σ true. $\triangleleft$

Example 4.1.1. The share constraint $x \oplus y = \bullet \ \wedge \ x =$  is represented by the share equation system $\Sigma = \{x \oplus y = \bullet, x =$  $\}$. Moreover, the context $S = \{x =$  , $y =$  $\}$

is a solution of Σ. $\triangleleft$

Remark. It is worth observing that equality is a special form of equation. Indeed, each equality $v = w$ is equivalent to the $\oplus$-equation $v \oplus \circ = w$. Therefore, from the theoretical viewpoint, one can combine the two lists into a single list of $\oplus$-equation. However, in practice, equalities usually outnumber $\oplus$-equations, plus they are significantly easier to handle and optimize. As a result, we distinguish the two lists for performance reasons.

To clarify the interaction between entailment checkers and the share solver, we outline extraction of share equations from two entailments:

$$x \overset{\chi_1}{\mapsto} v_a \ * \ x \overset{\chi_2}{\mapsto} v_a \ \vdash \ \exists s_c.\ x \overset{s_c}{\mapsto} v_c \qquad \| \qquad x \overset{\chi_a}{\mapsto} v_a \ \vdash \ x \overset{\chi_c}{\mapsto} v_c$$

First, the two entailments need to be normalized and the shares floated out*:

$$x \overset{s_a}{\mapsto} v_a \wedge \chi_1 \oplus \chi_2{=}s_a \ \vdash \ \exists s_c.\ x \overset{s_c}{\mapsto} v_c \qquad \| \qquad x \overset{s_a}{\mapsto} v_a{\wedge}s_a{=}\chi_a \ \vdash \ \exists s_c.\ x \overset{s_c}{\mapsto} v_c{\wedge}s_c{=}\chi_c$$

Discharging the heap obligations occurs by pairing the $x \overset{s_c}{\mapsto} v_c$ predicate with $x \overset{s_a}{\mapsto} v_a$, which generates the share obligations $s_a{=}s_c$ or $\exists s_r.\ s_a{=}s_c \oplus s_r$. These obligations are combined with the rest of the share constraints, resulting in two share proof obligations for each original entailment.

$$\begin{cases} \chi_1{\oplus}\chi_2{=}s_a \ \vdash \ \exists s_c \quad .\ s_a{=}s_c \\ \chi_1{\oplus}\chi_2{=}s_a \ \vdash \ \exists s_c, s_r.\ s_c{\oplus}s_r{=}s_a \end{cases} \qquad \| \qquad \begin{cases} s_a{=}\chi_a \vdash \exists s_c \quad .\ s_c{=}\chi_c \wedge s_a{=}s_c \\ s_a{=}\chi_a \vdash \exists s_c, s_r.\ s_c{=}\chi_c \wedge s_a{=}s_c{\oplus}s_r \end{cases}$$

Although simple, the first original entailment often occurs when verifying a method that requires only read access to a heap location; the existential allows callers to be flexible regarding which specific share of x they have. One technical point is that many separation logics (including those used in HIP/SLEEK [NDQC07], Heap-Hop [Vil11], and coreStar [BDD+11]) only allow *positive* (non-empty) fractional shares over a points-to predicate (the empty share over a points-to is equivalent to $\bot$); thus, the above existential must be restricted to never choose the empty share.

Problem formulation. We have now given two examples of extracting share equations

*The antecedent $\exists$ is automatically interpreted as a $\forall$ over the entailment using renaming when needed to avoid name clashes.

from separation logic formulas. Once the translation is finished, a separation logic entailment checker can ask our share prover two questions:

1. **SAT**(Σ): Given an equation system Σ, is Σ is satisfiable? That is:

$$\exists S.\ S \models \Sigma$$

   SLEEK uses SAT checks to help prune unfeasible proof searches.

2. **IMP**(Σ_1, Σ_2): Given two systems Σ_1 and Σ_2, does Σ_1 entail Σ_2? That is:

$$\Sigma_1 \vdash \Sigma_2 \quad \text{iff} \quad \forall S.\ S \models \Sigma_1 \Rightarrow S \models \Sigma_2.$$

Example 4.1.2. Consider two equation systems Σ_1 and Σ_2 s.t.:

1. $\Sigma_1 = \{x_2,\ x_1 = \overset{\frown}{\bullet\ \circ},\ x_1 \oplus x_2 = \bullet\}$
2. $\Sigma_2 = \{x_3,\ x_1 \oplus x_3 = \overset{\frown}{\bullet\ \circ}\}$.

Then checking **SAT**(Σ_1) and **IMP**(Σ_1, Σ_2) are equivalent to the following queries respectively:

1. $\exists x_1 \exists x_2.\ x_1 = \overset{\frown}{\bullet\ \circ} \ \wedge\ x_1 \oplus x_2 = \bullet$
2. $\forall x_1 (\exists x_2.\ x_1 = \overset{\frown}{\bullet\ \circ} \ \wedge\ x_1 \oplus x_2 = \bullet \rightarrow \exists x_3.\ x_1 \oplus x_3 = \overset{\frown}{\bullet\ \circ})$.

◁

In practice this is sufficient; we will detail how we answer these questions in subsequent sections.

4.2 Decision procedures over tree shares

Here we introduce a decision procedure for discharging tree share proof obligations generated by program verifiers. Recall from §4.1 that equation systems contain equations of the form

$a \oplus b = c$ and $v = w$, plus a list of variables that should be quantified existentially. Moreover, a *solution* S of Σ is a finite mapping from the variables of Σ into tree shares. We write $S \models \Sigma$ to mean that S is a solution of Σ; and the **SAT** query is then simply to check whether $\exists S. S \models \Sigma$. Furthermore, we write $\Sigma_1 \vdash \Sigma_2$ to mean that every solution S that satisfies Σ_1 also satisfies Σ_2, *i.e.*, $\forall S.\ S \models \Sigma_1 \Rightarrow S \models \Sigma_2$; this is exactly the **IMP** query.

The key reason **SAT** and **IMP** are nontrivial is that the space is dense*. That is, there exist trees of arbitrary height, seeming to rule out a brute force search. If we cannot find a solution to Σ at height 5, how do we know that one is not lurking at height 10,000? If we check $\Sigma_1 \vdash \Sigma_2$ when the variables are restricted to constants of height 5, how do we know that the entailment will continue to hold when the variables range over constants of arbitrary height?

Our key theoretical insight is that despite the infinite domain, both **SAT** and **IMP** are decidable by searching in the finite subdomain of trees with bounded height. Define the *system height* $|\Sigma|$ as the height of the highest tree constant in Σ or 0 if Σ contains only variables†. For solutions S, let $|S|$ be the highest constant in its codomain. In §4.3, we will prove our key theoretical result: that for both **SAT** and **IMP** queries, if the height of the system(s) of equations is n, then it is sufficient to restrict the search to solutions of height n.

Example 4.2.1. Let $\Sigma = \{x = \bullet, x \oplus y = \begin{smallmatrix} & \frown & \\ \bullet & & \circ \end{smallmatrix}\}$ then Σ is unsatisfiable because substituting $x = \bullet$ into the second equation will result in $\bullet \oplus y = \begin{smallmatrix} & \frown & \\ \bullet & & \circ \end{smallmatrix}$ which is not satisfiable by any y. Using the key theoretical result, it is sufficient to check for solution of height at most $|\Sigma| = 1$. There are 4 trees of height at most 1, namely $\{\circ, \bullet, \begin{smallmatrix} \frown \\ \bullet \quad \circ \end{smallmatrix}, \begin{smallmatrix} \frown \\ \circ \quad \bullet \end{smallmatrix}\}$. As a result, we need to check for $4^2 = 16$ such (x, y) pairs. ◁

Of course, we do not want to blindly search through an exponentially large space if we can avoid it! Our goal is to describe and prove sound the algorithms SAT and IMP for **SAT** and **IMP** given in Algorithms 2 and 3. The core of our decision procedures are the

*This is by design: density is needed to enable the "Infinite Splitability" axiom, which is needed to support the verification of divide-and-conquer algorithms.

†Since we are computer scientists, we start counting with 0, so $|\circ| = |\bullet| = 0$.

REDUCE and REDUCEI functions, which use the shape of the tree constants in the system to guide their search. There are four subroutines: SIMPLIFY, DECOMPOSE, FORMULA, and SMT_SOLVER. SMT_SOLVER is just a call into an off-the-shelf SAT/SMT solver; our prototype attaches to both MiniSat [ES03] and Z3 [dMB08]. The other three subroutines are discussed in detail below and their descriptions are mentioned in Algorithm 1.

4.2.1 Utility functions for SAT and IMP

We will discuss three subroutines SIMPLIFY, DECOMPOSE and FORMULA that are used in both SAT and IMP. Simply put, SIMPLIFY simplifies an equation system using heuristics. In practice, many tree share constraints extracted from HIP/SLEEK can usually be simplified further, *e.g.*, $x \oplus \circ = y$ is equivalent to $x = y$. Thus the main purpose of SIMPLIFY is to reduce the size of the equation system so that the overall performance can be improved. The second subroutine is DECOMPOSE that splits an equation system Σ into a pair of left-subsystem Σ^l and right-subsystem Σ^r. The splitting mechanism is generalized from the Split function in §2.2.2 that splits a tree τ into its left and right subtree. Here we can also split a variable x into two fresh variables x^l and x^r, or equality/equation in which the splitting is done argument-wise. Last but not least, FORMULA transforms an equation system of height zero into an equivalent Boolean formula that can be handled by external SMT solvers.

♣ SIMPLIFY (Algorithm 1). SAT/SMT solvers can require a lot of computation time. Accordingly, SIMPLIFY attempts reduce the size of the problem with a combination of several techniques. First, each equation that contains two or three tree constants is simplified into an equality (or $\top/\bot$). To do so, SIMPLIFY sometimes uses the inverse operation of $\oplus$, written $\ominus$, and which satisfies $a \oplus b = c$ iff $c \ominus a = b$. To calculate the (partial) operation $a \ominus b$, unfold a and b to the same shape (just as with $\oplus$); calculate the difference leafwise using the rules $\bullet \ominus \bullet = \circ$, $\bullet \ominus \circ = \bullet$, and $\circ \ominus \circ = \circ$; and then fold the result back into

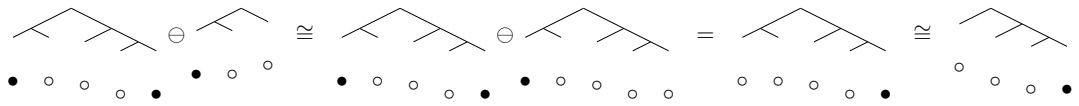
Algorithm 1 Common utility functions for SAT and IMP

function SIMPLIFY(Σ)
Ensure: Simplify Σ using a list of heuristics
  for each equation $e \stackrel{\text{def}}{=} x \oplus y = z$ contains at least two constants, or $\circ$, or $\bullet$ **do**
    Simplify e into e' using a list of heuristics
    if e' is a contradiction **then**
      return $\bot$
    else if e is an equality(or two equalities) of the form $x = t$ **then**
      Substitute x with t into Σ
    end if
  end for
end function

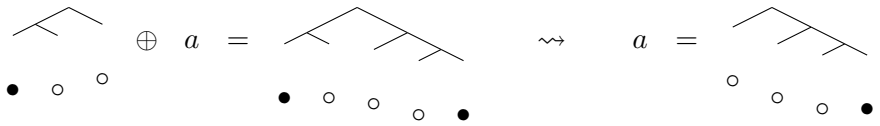
function DECOMPOSE(Σ)
Ensure: Return a subsystem pair (Σ^l, Σ^r) of Σ using Split from §2.2.2
  let Σ^l, Σ^r be two empty equation systems
  for each existential variable x in Σ **do**
    let $(x^l, x^r) \leftarrow \mathsf{Split}(x)$
    Add x^l to Σ^l
    Add x^r to Σ^r
  end for
  for each equation $x_1 \oplus x_2 = x_3$ **do**
    $(x_i^l, x_i^r) \leftarrow \mathsf{Split}(x_i)$
    Add $x_1^l \oplus x_2^l = x_3^l$ to Σ^l
    Add $x_1^r \oplus x_2^r = x_3^r$ to Σ^r
  end for
  for each equality $x_1 = x_2$ **do**
    let $(x_i^l, x_i^r) \leftarrow \mathsf{Split}(x_i)$
    Add $x_1^l = x_2^l$ to Σ^l
    Add $x_1^r = x_2^r$ to Σ^r
  end for
  return (Σ^l, Σ^r)
end function

function FORMULA(Σ)
Require: Σ is an equation system of height 0
Ensure: Transform Σ into an equivalent Boolean formula
  for each equality $x_1 = x_2$ **do**
    $\Psi \leftarrow (x_1 \wedge x_2) \ \vee \ (\neg x_1 \wedge \neg x_2)$
    $\Phi \leftarrow \Phi \ \wedge \ \Psi$
  end for
  for each equation $x_1 \oplus x_2 = x_3$ **do**
    $\Psi \leftarrow (x_1 \wedge \neg x_2 \wedge x_3) \ \vee \ (\neg x_1 \wedge x_2 \wedge x_3) \ \vee \ (\neg x_1 \wedge \neg x_2 \wedge \neg x_3)$
    $\Phi \leftarrow \Phi \ \wedge \ \Psi$
  end for
  $\Phi \leftarrow \exists x_1 \ldots \exists x_n.\ \Phi$ for each existential variable x_i in Σ
  return Φ
end function

canonical form, *e.g.*:



$\ominus$ is needed when one of the constants appears on the RHS of an equation, *e.g.*,*



If an equation reaches a tautology (*e.g.*, $\circ \oplus v = v$) then it is removed; if an equation reaches a contradiction (*e.g.*, $\bullet \oplus \bullet = v$) then we mark the entire system as equivalent to $\bot$. Second, SIMPLIFY will rewrite equalities; *e.g.*, if the equality $v = \chi$ is in the system then SIMPLIFY will substitute χ for v in the remainder of the system. Third, SIMPLIFY uses certain domain-specific knowledge to simplify equations with zero or one tree constant(s), including the following examples:

$$
\begin{array}{lll|lll}
v_1 \oplus v_2 = \circ & \rightsquigarrow & v_1 = \circ \wedge v_2 = \circ & v_1 \oplus \circ = v_2 & \rightsquigarrow & v_1 = v_2 \\
v_1 \oplus \bullet = v_2 & \rightsquigarrow & v_1 = \circ \wedge v_2 = \bullet & v_1 \oplus v_2 = v_1 & \rightsquigarrow & v_2 = \circ \\
v_1 \oplus v_1 = v_2 & \rightsquigarrow & v_1 = \circ \wedge v_2 = \circ & & &
\end{array}
$$

The result of SIMPLIFY is a new (in practice considerably smaller!) system of equations Σ' that has the same solutions, as expressed by the following Lemma:

Lemma 4.2.1 ([Sol])**.** For all solutions S, $S \models \Sigma$ iff $S \models \mathsf{SIMPLIFY}(\Sigma)$. ◁

Proof intuition. Intuitively, heuristics in SIMPLIFY are equivalent transformations and thus preserve the satisfiability property of the equation system. □

We will also need to know that SIMPLIFY does not increase the height of an equation system.

*In §4.2 we use the symbol $\rightsquigarrow$ to indicate a transformation taken by the subroutine currently under discussion, so here it is referring to one of the operations of SIMPLIFY.

To prove this, we need the following fact about $\oplus$ and $\ominus$:

Lemma 4.2.2 ([Sol])**.** If $a \oplus b = c$ or $a \ominus b = c$ then $|c| \leq \max(|a|, |b|)$. ◁

Proof intuition. Informally speaking, although trees can be unfolded from canonical form temporarily, they are never unfolded beyond the height of the highest tree and thus the height of the result tree is also bounded above by that value. □

Consequently, it is straightforward to prove the associated fact on SIMPLIFY:

Lemma 4.2.3 ([Sol])**.** $|\mathsf{SIMPLIFY}(\Sigma)| \leq |\Sigma|$. ◁

Proper equation systems. An equation system Σ is *proper* when all of the equations and equalities in Σ have no more than one constant. $\mathsf{SIMPLIFY}(\Sigma)$ is always proper, which simplifies some of our upcoming soundness proofs; accordingly, **hereafter we assume that all of our equation systems are proper.**

♣ DECOMPOSE (Algorithm 1). The heart of our decision procedure is DECOMPOSE, which takes an equation system Σ of height n and produces two independent systems Σ_l and Σ_r with heights at most $n - 1$. We decompose equalities and equations as follows:

$$v \rightsquigarrow (v^l, v^r) \qquad \textit{vars}$$

$$\circ \rightsquigarrow (\circ, \circ) \qquad \bullet \rightsquigarrow (\bullet, \bullet) \qquad \overset{\tau_1 \quad \tau_2}{\frown} \rightsquigarrow (\tau_1, \tau_2) \qquad \textit{consts}$$

$$\left.\begin{array}{l} a \rightsquigarrow (a^l, a^r) \\ b \rightsquigarrow (b^l, b^r) \\ c \rightsquigarrow (c^l, c^r) \end{array}\right\} \quad \begin{array}{ll} a \oplus b = c & \rightsquigarrow (a^l \oplus b^l = c^l,\ a^r \oplus b^r = c^r) \\ a = b & \rightsquigarrow (a^l = b^l,\ a^r = b^r) \end{array} \qquad \textit{eqs}$$

In addition, DECOMPOSE also transforms the list of existentially bound variables so that if v was existentially bound in Σ then v^l is existentially bound in Σ^l and v^r is existentially bound in Σ^r. Fresh variable names are chosen so that the system can determine which "parent" variables are associated with which "child" variables. We write $\hat{x}$ for the parent variable function, *e.g.*, $\hat{v^l} = \hat{v^r} = v$.

The key fact about DECOMPOSE is that the solution of the original system is tightly related to the solutions of the decomposed systems, as follows:

Lemma 4.2.4 ([Sol])**.** Given a system Σ and a solution S such that $\mathsf{DECOMPOSE}(\Sigma) = (\Sigma^l, \Sigma^r)$ and $\mathsf{DECOMPOSE}(S) = (S^l, S^r)$, then $S \models \Sigma$ iff $S^l \models \Sigma^l$ and $S^r \models \Sigma^r$. ◁

By $\mathsf{DECOMPOSE}(S)$ we mean the division of the solution S into two independent solutions:

$$\mathsf{DECOMPOSE}(S) \quad \equiv \quad (\lambda v.\mathsf{DECOMPOSE}(S(\hat{v})).1, \lambda v.\mathsf{DECOMPOSE}(S(\hat{v})).2)$$

Lemma 4.2.4 holds because the left and right subtrees of a binary tree are independent from each other. Moreover, DECOMPOSE decreases height:

Lemma 4.2.5 ([Sol])**.** If $\mathsf{DECOMPOSE}(\Sigma) = (\Sigma_l, \Sigma_r)$, then $|\Sigma| > \max(|\Sigma_l|, |\Sigma_r|)$ or we were at height 0 to begin with, *i.e.*, $|\Sigma| = |\Sigma_l| = |\Sigma_r| = 0$. ◁

Proof intuition. Direct from the fact that Split decreases tree height (Lemma 2.2.1). □

♣ FORMULA (Algorithm 1). After repeatedly applying DECOMPOSE, $|\Sigma| = 0$, *i.e.*, the embedded constants are only $\circ$ and $\bullet$. Tree constants at height zero have a natural interpretation as booleans, with $\circ$ as $\bot$ and $\bullet$ as $\top$. Likewise, solutions at height zero can be used as valuations (maps from variables to $\top$ and $\bot$) for logic formulas. Accordingly, FORMULA translates the equations and equalities in a system of equations of height zero into logic formulas as follows:

$$\begin{aligned} a \oplus b = c \quad &\rightsquigarrow \quad (\neg a \wedge \neg b \wedge \neg c) \ \vee \ (\neg a \wedge b \wedge c) \ \vee \ (a \wedge \neg b \wedge c) \\ a = b \quad &\rightsquigarrow \quad (\neg a \wedge \neg b) \ \vee \ (a \wedge b) \end{aligned}$$

Furthermore, if some of the arguments in the equation are constant then the Boolean formula is further simplified, *e.g.*. $a \oplus b = \bullet \rightsquigarrow (a \wedge \neg b) \ \vee \ (\neg a \wedge b)$. Each resulting formula is $\wedge$-conjoined together to get a single formula that represents the entire system, as indicated by the following lemma:

Lemma 4.2.6 ([Sol])**.** Let $|S| = |\Sigma| = 0$ and $v_1, \ldots, v_n$ be the existentially bound variables in Σ. Then $S \models \Sigma$ iff $S \models \exists v_1 \ldots \exists v_n.\ \mathsf{FORMULA}(\Sigma)$. ◁

To connect to a pure SAT solver (*e.g.*, MiniSat) we then compile the existential into a disjunction; *e.g.*, $\exists v.\ \phi \ \rightsquigarrow\ (v{=}\top \wedge \phi) \vee (v{=}\bot \wedge \phi)$. In contrast, SMT solvers such as Z3 can handle existentials over booleans directly.

The proof of Lemma 4.2.6 is by simple case analysis, but critics will rightly observe that the hypothesis $|S| = 0$, which is crucial to make the case analysis finite, is in general not true. We will see below how to overcome this difficulty.

4.2.2 Overview of SAT procedure

Algorithm 2 Decision procedure SAT for **SAT** problem

```
1: function REDUCE(Σ)
Ensure: Return a Boolean formula that is equivalent to the input equation system
2:     Σ′ ← SIMPLIFY(Σ)
3:     if |Σ′| = 0 then
4:         return FORMULA(Σ′)
5:     else
6:         (Σ^l, Σ^r) ← DECOMPOSE(Σ′)
7:         Φ ← REDUCE(Σ^l) ∧ REDUCE(Σ^r)
8:         return Φ
9:     end if
10: end function
11:
12: function SAT(Σ)
Ensure: Return true iff Σ is satisfiable
13:     Φ ← REDUCE(Σ)
14:     return SMT_SOLVER(Φ)
15: end function
```

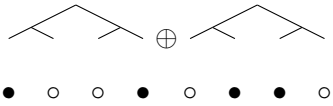
The procedure SAT is described in Algorithm 2. The heart of SAT is the function REDUCE that utilizes all subroutines from the previous subsection. The last puzzle piece for the correctness of SAT is one of the two major theoretical insights of this chapter:

Theorem 4.2.1 ([Sol])**.** Σ is satisfiable if and only if Σ can be satisfied with a solution S whose height is at most $|\Sigma|$, *i.e.*:

$$\exists S.\ S \models \Sigma \ \text{ iff } \ \exists S.\ |S| \leq |\Sigma| \ \wedge\ S \models \Sigma.$$

◁

Example 4.2.2. Let $\Sigma = \{x \oplus y = \bullet\}$ then $|\Sigma| = 0$. One can check that Σ has infinitely

many solutions of large height, *e.g.*,  $= \bullet$. Yet there exist solutions of height 0, *e.g.*, $\circ \oplus \bullet = \bullet$. ◁

We will defer the proof of Theorem 4.2.1 until §4.3.1; our task in this section is to show how it fits into our correctness proof for SAT, *i.e.*,

Theorem 4.2.2 ([Sol])**.** $\mathsf{SAT}(\Sigma) = \top$ iff Σ is satisfiable, *i.e.*, $\exists S.\ S \models \Sigma$. ◁

Proof. Given Σ, we call REDUCE and feed the result into the SMT solver, so Theorem 4.2.2 depends on REDUCE turning Σ into an equivalent logical formula.

The proof of REDUCE is by (complete) induction on $|\Sigma|$. Both the base case and the inductive case begin by applying applying SIMPLIFY to reach Σ'. By Lemma 4.2.1, Σ' is satisfiable iff Σ was satisfiable; moreover, by Lemma 4.2.3, $|\Sigma'| \leq |\Sigma|$. After simplification, the base case and the inductive case proceed differently.

In the base case, $|\Sigma'| = 0$ and REDUCE calls FORMULA to produce a logical formula that by Lemma 4.2.6 is equivalent to Σ' as long as the solution has height 0. Theorem 4.2.1 completes the base case by telling us that testing satisfiability at height 0 is sufficient to determine satisfiability in general.

In the inductive case, we apply DECOMPOSE over Σ' to yield Σ_l and Σ_r. Lemma 4.2.5 tells us that both new systems have lower height, so we can apply the induction hypothesis to verify the recursive call and get two new formulae whose truth are equivalent to Σ^l and Σ^r. Lemma 4.2.4 completes the inductive step by telling us that the conjunction of Σ^l and Σ^r is equivalent to Σ'. □

Example 4.2.3. Let $\Sigma = \{x \oplus y = \overset{\bullet}{\underset{\bullet}{\wedge}}\overset{\circ}{}, x \oplus y = z, z \oplus \circ = \bullet\}$. When calling REDUCE($\Sigma$), the subroutine SIMPLIFY(Σ) in line 2 reduces the third equation $z \oplus \circ = \bullet$ into $z = \bullet$ and substitutes the value of z into the remaining equations. Thus $\Sigma' = \{x \oplus y = \overset{\bullet}{\underset{\bullet}{\wedge}}\overset{\circ}{}, x \oplus y = \bullet\}$.

As $|\Sigma'| = 1 > 0$, DECOMPOSE(Σ') is called in line 6 which returns the pair of left and right subsystem of Σ':

1. $\Sigma^l = \{x^l \oplus y^l = \bullet, x^l \oplus y^l = \bullet\}$ and

2. $\Sigma^r \;=\; \{x^r \oplus y^r = \circ, x^r \oplus y^r = \bullet\}$

The two recursive calls in line 7 return two the equivalent Boolean formulas:

1. $\mathsf{REDUCE}(\Sigma^l) = \Psi \;\wedge\; \Psi$ where $\Psi = (x^l \wedge \neg y^l) \;\vee\; (\neg x^l \wedge y^l)$
2. $\mathsf{REDUCE}(\Sigma^r) = \bot$ because $x^r \oplus y^r \;\rightsquigarrow\; x^r = \circ \;\wedge\; y^r = \circ$ and their substitutions into the second equation yields $\circ \oplus \circ = \bullet$ which is a contradiction.

As a result, $\mathsf{REDUCE}(\Sigma)$ returns the Boolean formula $\Psi \;\wedge\; \Psi \;\wedge\; \bot$ which is not satisfiable when solved by SMT solver. ◁

4.2.3 Overview of IMP procedure

Algorithm 3 Decision procedure IMP for **IMP** problem

```
1: function REDUCEI(Σ1, Σ2)
Ensure: Return a pair of Boolean formulas that are equivalent to the input equation systems
2:     Σ1' ← SIMPLIFY(Σ1)
3:     Σ2' ← SIMPLIFY(Σ2)
4:     if (|Σ1'| = 0 AND |Σ2'| = 0) then
5:         return (FORMULA(Σ1'), FORMULA(Σ2'))
6:     else
7:         (Σ1^l, Σ1^r) ← DECOMPOSE(Σ1')
8:         (Σ2^l, Σ2^r) ← DECOMPOSE(Σ2')
9:         (Φ1^l, Φ2^l) ← REDUCEI(Σ1^l, Σ2^l)
10:        (Φ1^r, Φ2^r) ← REDUCEI(Σ1^r, Σ2^r)
11:        return (Φ1^l ∧ Φ1^r, Φ2^l ∧ Φ2^r)
12:    end if
13: end function
14:
15: function IMP(Σ1, Σ2)
Ensure: Return true iff Σ1 ⊢ Σ2
16:    (Φ1, Φ2) ← REDUCEI(Σ1, Σ2)
17:    return NOT SMT_SOLVER(Φ1 ∧ ¬Φ2)
18: end function
```

The implementation for IMP is elaborated in Algorithm 3, which is similar to SAT. We need the second major theoretical insight of this chapter to verify IMP.

Theorem 4.2.3 ([Sol]). $\Sigma_1 \vdash \Sigma_2$ iff $\Sigma_1 \vdash \Sigma_2$ for all solutions S s.t. $|S| \leq \max(|\Sigma_1|, |\Sigma_2|)$, *i.e.*:

$$\forall S.\ S \models \Sigma_1 \to S \models \Sigma_2 \quad \text{iff} \quad \forall S.\ |S| \leq \max(|\Sigma_1|, |\Sigma_2|) \to S \models \Sigma_1 \to S \models \Sigma_2.$$

◁

We will defer the proof until §4.3.2; just as we did with Theorem 4.2.1 above, our task here is to show how Theorem 4.2.3 fits into our correctness proof for IMPL, *i.e.*,

Theorem 4.2.4 ([Sol])**.** IMPL(Σ_1, Σ_2) iff $\Sigma_1 \vdash \Sigma_2$, *i.e.*, $\forall S.\ S \models \Sigma_1 \to S \models \Sigma_2$. ◁

Proof. The major effort is proving that REDUCEI correctly transforms Σ_1 and Σ_2 into equivalent logical formulae Φ_1 and Φ_2 such that $\Sigma_1 \vdash \Sigma_2$ iff $\Phi_1 \to \Phi_2$; afterwards we simply use the standard SAT/SMT solver trick of converting a validity check for $\Phi_1 \to \Phi_2$ into an **un**satisfiability check for $\Phi_1 \wedge \neg\Phi_2$.

The proof of REDUCEI is largely in parallel with the proof of REDUCE in Theorem 4.2.2. We proceed by complete induction, this time on $\max(|\Sigma_1|, |\Sigma_2|)$. Again the base and inductive cases begin in the same way. We apply SIMPLIFY to reach Σ_1' and Σ_2' and again use Lemma 4.2.1 to guarantee that $\Sigma_1' \vdash \Sigma_2'$ iff $\Sigma_1 \vdash \Sigma_2$; Lemma 4.2.3 ensures that $\max(|\Sigma_1'|, |\Sigma_2'|) \leq \max(|\Sigma_1|, |\Sigma_2|)$.

After simplification, the base and inductive cases diverge. In the base case, $\max(|\Sigma_1'|, |\Sigma_2'|) = 0$ and we call FORMULA to reach two logical formulae, the first equivalent to Σ_1' and the second equivalent to Σ_2', as long as the solutions are of height zero (Lemma 4.2.6). Theorem 4.2.3 completes the base case by observing that it is sufficient to check only the solutions of height $|\Sigma_1'|$, *i.e.* zero.

In the inductive case, we apply DECOMPOSE over Σ_1' and Σ_2' to decrease the maximum of their heights (Lemma 4.2.5), and thus letting us use the induction hypothesis for the recursive calls. Afterwards, we have four formulae (Φ_1^{l}, etc.); we then conjoin both antecedents and both consequents using Lemma 4.2.4. □

Example 4.2.4. Let $\Sigma_1 = \{x \oplus y = \bullet, x = \overset{\frown}{\bullet\ \circ}\}$ and $\Sigma_2 = \{y = \overset{\frown}{\circ\ \bullet}\}$. Suppose we want to check whether $\Sigma_1 \vdash \Sigma_2$ (which it is) using IMP. Both systems are already simplified so REDUCEI(Σ_1, Σ_2) calls the subroutine DECOMPOSE in line 7 and 8. As a result, we have 4 subsystems:

1. $\Sigma_1^l = \{x^l \oplus y^l = \bullet, x^l = \bullet\}$

2. $\Sigma_1^r = \{x^r \oplus y^r = \bullet, x^r = \circ\}$
3. $\Sigma_2^l = \{y^l = \circ\}$
4. $\Sigma_2^r = \{y^r = \bullet\}$

Using FORMULA, we compute their corresponding Boolean formulas:

1. $\Phi_1^l = [(x^l \wedge \neg y^l) \ \vee \ (\neg x^l \wedge y^l)] \ \wedge \ x^l$
2. $\Phi_1^r = [(x^l \wedge \neg y^l) \ \vee \ (\neg x^l \wedge y^l)] \ \wedge \ \neg x^l$
3. $\Phi_2^l = \neg y^l$
4. $\Phi_2^r = y^r$

Finally, we use SMT solver to check satisfiability of the formula $(\Phi_1^l \ \wedge \ \Phi_1^r) \ \wedge \ \neg(\Phi_2^l \ \wedge \ \Phi_2^r)$ which returns False. Hence the entailment $\Sigma_1 \vdash \Sigma_2$ is valid. ◁

4.2.4 Optimizations

The algorithms presented in Algorithms 2 and 3 get the job done but yield far from optimal performance. Our prototype incorporates a number of additional optimizations including optimizations during **SAT** that drop equalities after substitution and a lazier on-demand version of DECOMPOSE. In addition to utilizing the lazier version of DECOMPOSE, optimizations during **IMP** include dropping existentials from the antecedent, substituting equalities from the antecedent into the consequent, and stopping decomposition when the antecedent has reached height zero and performing a **SAT** check on the antecedent if the consequent has not also reached height zero. Several optimizations require some additional theoretical insight; *e.g.*, the last requires the following:

Lemma 4.2.7. Let S be a solution of Σ. Then $|S| \geq |\Sigma|$. ◁

Proof. Recall that we assume that Σ is proper, *i.e.*, each equation has at most one constant. If $|\Sigma| = 0$, we are done. Otherwise, by definition of $|\Sigma| = n$, there must be an equation σ containing a constant χ with height n. Since $S \models \Sigma$ we know that $S \models \sigma$. Assume both variables v_1 and $v_2 \in \sigma$ have height lower than n in S (*i.e.*, $\max(|S(v_1)|, |S(v_2)|) < |\chi|$).

By Lemma 4.2.2 we also know that $|\chi| \leq \max(|S(v_1)|, |S(v_2)|)$, so by transitivity we have $|\chi| < |\chi|$, a contradiction. Accordingly, at least one of the variables v_i must have had height at least n. □

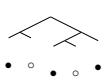
Unsurprisingly, the actual code used in the prototype is much more complicated than the algorithms presented above, and accordingly is much harder to verify. As a result, we will develop a verified implementation in the next chapter.

4.3 Sufficiency of finite search over tree shares

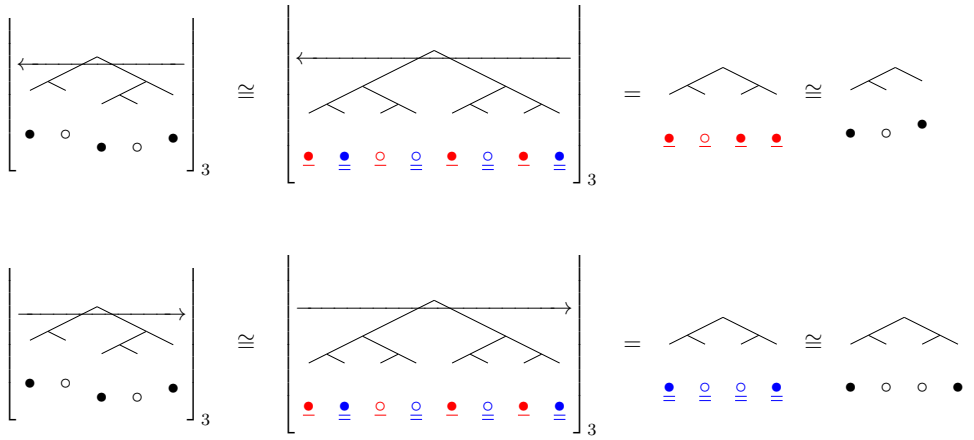
The SAT and IMP algorithms presented in §4.2.2 and §4.2.3 are basically doing a shape-guided search through a finite domain. Our key theoretical insight is that a finite search is sufficient, as formalized in the statement of Theorems 4.2.1 and 4.2.3 in §4.2. Our next task is to prove these theorems, which is the focus of the remainder of this section. The most technical parts—Lemmas 4.3.1 and 4.3.3—have been mechanically verified in Coq. The remaining proofs have been carefully checked on paper.

4.3.1 The sufficiency of finite search for SAT

We begin by explaining two related operations given a tree τ and natural n: left rounding, written $\lfloor \overleftarrow{\tau} \rfloor_n$; and right rounding, written $\lfloor \overrightarrow{\tau} \rfloor_n$. Because of the canonical form for tree shares, their associated formal definitions are somewhat unpleasant, but informally what is going on is simple. First, the tree τ is unfolded to height n. Second, we shrink the height of the tree by uniformly choosing the left (respectively, right) leaf from each pair of leaves at height n. Finally, we refold the resulting tree back into canonical form. For **convenience**, the subscript n is called the *rounding level*.

For illustration, here we left and right round the tree  to height 3. To help visually track what is going on, we have highlighted the left leaf in each pair with the color red and

the right leaf in each pair with the color blue.



Lemma 4.3.1 (Properties of rounding functions [Sol])**.** Let $\tau, \tau_l, \tau_r, \tau_1, \tau_2, \tau_3$ be tree shares and $n \in \mathbb{N}$ the rounding level then:

1. For any rounding level greater than the tree height, the two rounding functions are the identity function, *i.e.*:

$$\text{if} \quad n > |\tau| \quad \text{then} \quad \lfloor \overleftarrow{\tau} \rfloor_n = \lfloor \overrightarrow{\tau} \rfloor_n = \tau$$

2. If the rounding level is the tree height then the two rounding functions yield trees with smaller height, *i.e.*:

$$\text{if} \quad n = |\tau| \text{ and } \tau_l = \lfloor \overleftarrow{\tau} \rfloor_n \text{ and } \tau_r = \lfloor \overrightarrow{\tau} \rfloor_n \quad \text{then} \quad \max(|\tau_l|, |\tau_r|) < n$$

3. The rounding functions preserve the join relation, *i.e.*:

$$\text{if} \quad \tau_1 \oplus \tau_2 = \tau_3 \quad \text{then} \quad \lfloor \overleftarrow{\tau_1} \rfloor_n \oplus \lfloor \overleftarrow{\tau_2} \rfloor_n = \lfloor \overleftarrow{\tau_3} \rfloor_n \quad \text{and} \quad \lfloor \overrightarrow{\tau_1} \rfloor_n \oplus \lfloor \overrightarrow{\tau_2} \rfloor_n = \lfloor \overrightarrow{\tau_3} \rfloor_n$$

where $n \geq \max(|\tau_1|, |\tau_2|, |\tau_3|)$.

◁

Proof intuition. Lemma 4.3.1 states (1) that $\lfloor \overleftarrow{\tau} \rfloor_n$ and $\lfloor \overrightarrow{\tau} \rfloor_n$ do not affect τ if $n > |\tau|$; and

(2) will decrease the height if $n = |t|$. Most importantly, (3) $\lfloor \overleftarrow{\tau} \rfloor_n$ and $\lfloor \overrightarrow{\tau} \rfloor_n$ preserve the join relation when n is big enough. □

We override $\lfloor \overleftarrow{\cdot} \rfloor_n$ and $\lfloor \overrightarrow{\cdot} \rfloor_n$ to work over solutions S point-wise as follows:

$$\lfloor \overleftarrow{S} \rfloor_n \equiv \lambda v.\ \lfloor \overleftarrow{S(v)} \rfloor_n \qquad\qquad \lfloor \overrightarrow{S} \rfloor_n \equiv \lambda v.\ \lfloor \overrightarrow{S(v)} \rfloor_n$$

The key point of the rounding functions is given by the next lemma, a corollary of Lemma 4.3.1 after using a solution S to instantiate variables in a system Σ.

Lemma 4.3.2 ([Sol])**.** Let Σ be a equation system and S is its solution of height n. Suppose $n > |\Sigma|$ and let $S_l = \lfloor \overleftarrow{S} \rfloor_n$, $S_r = \lfloor \overrightarrow{S} \rfloor_n$ be two rounding contexts of level n computed from S. Then both S_l and S_r are solutions of Σ and their heights are both smaller than n. ◁

Proof intuition. The key to this lemma is that since we are rounding only at a height $n > |\Sigma|$, all of the constants in Σ are unchanged. Only the variables in S with height greater than $|\Sigma|$ are modified, but their new values are also solutions for Σ. □

With the preliminaries out of the way, we are finally ready to prove Theorem 4.2.1.

Theorem 4.2.1 ([Sol])**.** Σ is satisfiable if and only if Σ can be satisfied with a solution S whose height is at most $|\Sigma|$, *i.e.*:

$$\exists S.\ S \models \Sigma \ \text{ iff } \ \exists S.\ |S| \leq |\Sigma| \ \wedge \ S \models \Sigma.$$

Proof. $\Leftarrow$: Immediate. For $\Rightarrow$ direction: Suppose S is a solution of Σ. If $|S| \leq |\Sigma|$ then we are done, otherwise $|S| = |\Sigma| + n$ for some n. We proceed by strong induction on n. If $n = 0$ we are done. Otherwise, by Lemma 4.3.2 we know that $S_l = \lfloor \overleftarrow{S} \rfloor_{|\Sigma|+n}$ satisfies Σ and $|S_l| < |S|$, letting us apply the induction hypothesis. □

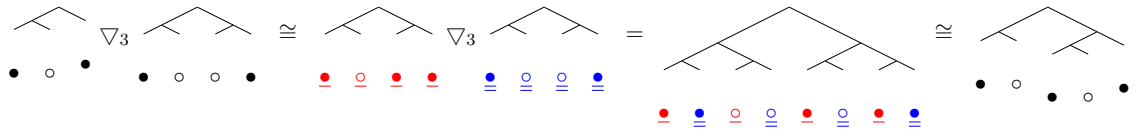
4.3.2 The sufficiency of finite search for IMP

The entailment problem **IMP** is more complicated than **SAT** due to the contravariance. Suppose we have computationally checked that all solutions S of height at most $|\Sigma_1|$ that

satisfy Σ_1 also satisfy Σ_2. Now suppose that $S \models \Sigma_1$ for some S such that $|S| = |\Sigma_1| + 1$, and we wish to know if $S \models \Sigma_2$. Lemma 4.3.2 tells us that $\lfloor \overleftarrow{S} \rfloor_{|\Sigma_1|+1} \models \Sigma_1$. Our computational verification then tells us that $\lfloor \overleftarrow{S} \rfloor_{|\Sigma_1|+1} \models \Sigma_2$, but then we are stuck: on its own, $\lfloor \overleftarrow{S} \rfloor_{|\Sigma_1|+1} \models \Sigma_2$ is too weak to prove $S \models \Sigma_2$.

The root of the problem is that $\lfloor \overleftarrow{\tau} \rfloor_n$ does not contain enough information about the original because half of the leaves are removed. Fortunately, the leaves that were dropped when we round left are exactly the leaves that are kept when we round right, and vice versa. We can define a third operation, written $\tau_l \bigtriangledown_n \tau_r$ and pronounced "average", that recombines the rounded trees back into the original. Just as was the case with the rounding functions, although the formal definition of $\tau_l \bigtriangledown_n \tau_r$ is somewhat unpleasant due to the necessity of managing the canonical forms, the core idea is straightforward. First, τ_l and τ_r are unfolded to height $n-1$. Second, each leaf in τ_l is paired with its corresponding leaf in τ_r. Finally, the resulting tree is folded back into canonical form. For **convenience**, we call subscript n the *averaging level*.

We illustrate with another example, highlighting again with red and blue:



Lemma 4.3.3 (Properties of averaging function [Sol])**.** Let τ, τ_i, τ_i' be tree shares and $n \in \mathbb{N}$ the averaging level then:

1. If the averaging level is greater than the tree height then averaging a tree with itself results in the same tree, *i.e.*:

$$\text{if} \quad n > |\tau| \quad \text{then} \quad \tau \bigtriangledown_n \tau = \tau$$

2. Averaging the left and right rounding tree of the same tree with the same rounding level and same averaging level results in the original tree, *i.e.*:

$$\text{if} \quad n \geq |\tau| \quad \text{then} \quad \lfloor \overleftarrow{\tau} \rfloor_n \bigtriangledown_n \lfloor \overrightarrow{\tau} \rfloor_n = \tau$$

3. The averaging function preserves the join relation, *i.e.*:

$$\text{if } \tau_1 \oplus \tau_2 = \tau_3 \text{ and } \tau_1' \oplus \tau_2' = \tau_3' \text{ then } (\tau_1 \bigtriangledown_n \tau_1') \oplus (\tau_2 \bigtriangledown_n \tau_2') = (\tau_3 \bigtriangledown_n \tau_3').$$

where $n > \max(|\tau_1|, |\tau_2|, |\tau_3|, |\tau_1'|, |\tau_2'|, |\tau_3'|)$.

◁

Proof intuition. The key points are (1) τ is an identity with itself, (2) $\bigtriangledown_n$ is the inverse of $\lfloor \overleftarrow{\tau} \rfloor_n$ and $\lfloor \overrightarrow{\tau} \rfloor_n$, and (3) $\bigtriangledown_n$ preserves the join operation $\oplus$ if n is big enough. □

Given a system Σ, Lemma 4.3.3 contains the facts we need to prove that the *averaging context* of two solutions S_l and S_r as defined below is also a solution.

$$S_l \bigtriangledown_n S_r \quad \equiv \quad \lambda v.\ S_l(v) \bigtriangledown_n S_r(v)$$

Lemma 4.3.4 (Properties of averaging context [Sol])**.** Let Σ be an equation system, $n \in \mathbb{N}$ the averaging level and S, S_l, S_r its contexts then:

1. Averaging the left and right rounding contexts at the same level results in the original context, *i.e.*:

$$\text{if } n \geq |S| \text{ then } \lfloor \overleftarrow{S} \rfloor_n \bigtriangledown_n \lfloor \overrightarrow{S} \rfloor_n = S$$

2. The averaging context of two solutions is also a solution, *i.e.*:

$$\text{if } S_l, S_r \text{ are solutions of } \Sigma \text{ then } S_l \bigtriangledown_n S_r \text{ is also a solution of } \Sigma$$

where $n > \max(|S_l|, |S_r|)$.

◁

Proof intuition. These properties are generalized from Lemma 4.3.3. □

We are now ready to attack the main theorem for **IMP**. Basically, we prove that **IMP** satisfies the small model property and thus $\mathbf{IMP}(\Sigma_1, \Sigma_2)$ can be restricted to solutions whose

height is up to the height of the whole system.

Theorem 4.3.1 ([Sol]). $\Sigma_1 \vdash \Sigma_2$ iff $\Sigma_1 \vdash \Sigma_2$ for all solutions S s.t. $|S| \leq \max(|\Sigma_1|, |\Sigma_2|)$, *i.e.*:

$$\forall S.\ S \models \Sigma_1 \rightarrow S \models \Sigma_2 \quad \text{iff} \quad \forall S.\ |S| \leq \max(|\Sigma_1|, |\Sigma_2|) \rightarrow S \models \Sigma_1 \rightarrow S \models \Sigma_2.$$

◁

Proof. $\Rightarrow$: Immediate. $\Leftarrow$: We apply complete induction, starting from $n = \max(|\Sigma_1|, |\Sigma_2|)$, on the height of solutions S of Σ_1. The base case ($|S| = n$) is immediate. For the inductive case, we know $S \models \Sigma_1$ and that all solutions S' of Σ_1 such that $|S'| < |S|$ are also solutions of Σ_2. By Lemma 4.3.2, we know that $\lfloor \overleftarrow{S} \rfloor_{|S|}$ and $\lfloor \overrightarrow{S} \rfloor_{|S|}$ are both solutions to Σ_1 with lower heights. The induction hypothesis yields that $\lfloor \overleftarrow{S} \rfloor_{|S|}$ and $\lfloor \overrightarrow{S} \rfloor_{|S|}$ are also both solutions of Σ_2. Lemma 4.3.4 completes the proof by telling us that $\lfloor \overleftarrow{S} \rfloor_{|S|} \ \triangledown_{|S|} \ \lfloor \overrightarrow{S} \rfloor_{|S|} = S$ is also a solution of Σ_2. □

4.4 Experiment evaluation

Here we discuss some implementation issues. Our prototype is an OCaml library that implements (an optimized version of) the algorithms from §4.2 to resolve the **SAT** and **IMP** queries issued by an entailment checker such as SLEEK.

Architecture. Our library contains four modules with clearly delimited interfaces so that each component can be independently used and improved:

1. An implementation of tree shares that exposes basic operations like equality testing, tree constructors, the join operation, and left/right projection.

2. The core: which reduces equation systems to boolean satisfiability. The bulk of the core module translates equation systems into boolean formulas via an optimized version of the procedures given in §4.2. As we will see, a considerable number of queries reduce to tautologies after repeated simplification/decomposition and can thus be discharged

without the SAT/SMT solver. If we are not that lucky, then the system is reduced to a list of existentially quantified variables, a list of variables that must be strictly positive, and a list of join facts over booleans of the form $v_1 \oplus v_2 = (\bullet | v_3)$.

3. The backend: tasked with interfacing with the SAT/SMT solver: translating the output format from the core to the input format of the SAT/SMT solver and retrieving the result. Our backend is quite lightweight so changing the underlying solver is a breeze. We provide backends to MiniSat [ES03] and Z3 [dMB08]; each add some final solver-specific optimizations.

4. A frontend: although the prover can be used as an OCaml library, we believe users may also want to query it as a standalone program. We provide a module for parsing input files and calling the core module.

Evaluation A: SLEEK embedding. Our OCaml library is designed to be easily incorporated into a general verification system. Accordingly, we tested our implementation by incorporating it into the SLEEK separation logic entailment prover and comparing its performance with our previous attempt at a share prover [HG12, §8.1]. That prover attempted to find solution by iteratively bounding the range of variables and trying to reach a fixed point; for example from  $\oplus\, x = y$ it would deduce $\circ \leq x \sqsubseteq$  and  $\sqsubseteq y$ where $a \sqsubseteq b \stackrel{\text{def}}{=} a \sqcup b = b$. The resulting incomplete solver was unable to prove most entailments containing more than one share variable, even for many extremely simple examples such as $v_1 \oplus v_2 = v_3 \vdash v_2 \oplus v_1 = v_3$.

We denote the implementation of the method presented here as ShP (Share Prover), and use BndP (Bound Prover) for the previous prover and present our results in Table 4.1. In the first column, we name our tests, which are broken into three test groups. The next five columns deal with the **SAT** queries generated by the tests, and the final five columns with the **IMP** queries.

The first two test groups were developed for BndP in [HG12] and so the share problems they generate are not particularly difficult. The first four tests verify increasingly precise

properties of a short (32-line) concurrent program in HIP, which calls SLEEK, which then calls BndP/ShP. In either case, the number of calls is the same and is given in the column labeled "call no."; *e.g.*, barrier-weak requires 116 **SAT** checks and 222 **IMP** checks.

The columns labeled "BndP (ms)" contain the cumulative time in milliseconds to run the BndP checker on all the queries in the associated test, *e.g.*, barrier-weak spends 0.4ms to verify 116 **SAT** problems and 2.1ms to verify 222 **IMP** checks. BndP may be highly incomplete, but at least it is *rapidly* highly incomplete. The columns labeled "ShP" contain the cumulative time in milliseconds to run the ShP checker, *e.g.*, barrier-weak spends 610ms verifying 116 **SAT** problems and 650ms verifying 222 **IMP** problems. Obviously this is quite a bit slower, but part of the context is that the rest of HIP/SLEEK is approximately 3,000ms on each of the first four tests—in other words, ShP, although much slower than BndP, is still considerably faster than the rest of HIP/SLEEK.

The remaining columns shed some light on what is going on; "SAT no." gives the number of queries that ShP actually submitted to the underlying SAT solver. For example, barrier-weak submitted 73 out of 116 queries to the underlying solver for **SAT** and 42 out of 222 queries to the underlying solver for **IMP**; the remaining 43+180 queries were solved during simplification/decomposition. Finally "SAT (ms)" gives the total amount of time spent in the underlying SAT solver itself; in every case this is the dominant timing factor. While it is not surprising that the SAT solver takes a certain amount of time to work its mojo, we suspect that most of the time is actually spent with process startup/teardown and hypothesize that performance would improve considerably with some clever systems engineering. Of course, another way to improve the timings in practice is to run BndP first and only resort to ShP when BndP gets confused.

Tests five through nine were also developed for BndP, but bypass HIP to test certain parts of SLEEK directly. Observe that when the underlying solver is not called, ShP is quite fast, although still considerably slower than BndP.

On the other hand, even if the total time is reasonable, what is the point of advocating a slower prover unless it can verify things the faster prover cannot? The tenth test tries to verify a simple 25-line **sequential** program whose verification uses fractional shares; we

	SAT					IMP				
test	call no.	BndP (ms)	ShP (ms)	SAT no.	SAT (ms)	call no.	BndP (ms)	ShP (ms)	SAT no.	SAT (ms)
barrier-weak	116	0.4	610	73	530	222	2.1	650	42	450
barrier-strong	116	0.6	660	73	510	222	2.2	788	42	460
barrier-paper	116	0.7	664	73	510	216	2.2	757	42	460
barrier-paper-ex	114	0.8	605	71	520	212	2.3	610	40	430
fractions	63	0.1	0.1	0	0	89	0.1	110	11	110
fractions1	11	0.1	0.1	0	0	15	0.1	31.3	3	30
barrier	68	0.1	0.9	0	0	174	1.2	3.9	0	0
barrier3	36	0.2	0.1	0	0	92	0.2	2.2	0	0
barrier4	59	0.1	0.7	0	0	140	0.9	2.4	0	0
read_ops	14	FAIL	210	14	208	27	FAIL	317	9	150
construct	4	FAIL	70	4	65	17	FAIL	880	17	270
join_ent	3	FAIL	70	3	30	3	FAIL	50	3	48

Table 4.1: Experimental timing results

write FAIL to indicate that BndP is unable to verify the queries. Finally, the eleventh and twelfth tests bypass HIP and instruct SLEEK to check entailments that BndP is unable to help verify.

For brevity, we report here the timings obtained only with the Z3 backend. Usually, the choice of backend does not make much difference, but in a few cases, *e.g.* read_ops and join_ent, choosing MiniSat can degrade the performance by a factor of 10. We leave the investigation of this behavior for future work.

Evaluation B: Standalone. While verifying programs, and their associated separation logic entailments is really the main goal, it is not so easy to casually develop HIP and SLEEK input files that exercise share provers aggressively. We designed a benchmark of 53 **SAT** and 50 **IMP** queries, many of which we specifically designed to stress a share prover in various tricky ways, including heavily skewed tree constants, deep heterogenous tree constants, numerous unconstrained variables, and a number of others.

ShP solved the entire test suite in 1.4s; 24 **SAT** checks and 18 **IMP** checks reached the

underlying solver. BndP could solve fewer than 10% of the queries.

4.5 Conclusion

We have shown how to extract a system of equations over a sophisticated fractional share model from separation logic formulae. We have developed a solver for the equation systems and proven that the associated problems are decidable. We have integrated our solver into the HIP/SLEEK verification toolset and benchmarked its performance to show that the system is usable in practice. In the related paper [LGH12], we claimed that our decision procedures could also handle non-zero constraints of the form $x \neq \circ$ by decomposing additionally $\log n$ steps where n is the number of such constraints. Unfortunately, our proof techniques for **SAT** and **IMP** could not be generalized accordingly. As we shall see in Chapter 5, the solvers in this chapter are indeed buggy for nonzero variables when being compared to our new certified solvers that can handle negative constraints of the form $\neg(a \oplus b = c)$.

Chapter 5

Complete certified procedures for tree share constraints

Norther Winslow: *I've been working on this poem for 12 years.*
Young Ed Bloom: *Really?*
Norther Winslow: *There's a lot of expectation. I don't wanna disappoint my fans.*
Young Ed Bloom: *May I?*
Young Ed Bloom: [Edward reads the poem on the notebook] *The grass so green. Skies so blue. Spectre is really great!*
Young Ed Bloom: *It's only three lines long.*
Norther Winslow: *This is why you should never show a work in progress.*

Big Fish (2003).

In Chapter 4, we developed two decision procedures SAT and IMP to solve the satisfiability (**SAT**) and entailment problem (**IMP**) over $\oplus$-equations. Our decision procedures are complete in the sense that they always return the correct result for any formatted input. Their correctness is carefully justified on paper whereas the key theoretical results are checked in Coq. The main drawback of our procedures is that they do not handle properly disequation constraints of the form $\neg(a \oplus b = c)$. We would like to clarify that it is not an

implementation complication but rather a technical difficulty: the key theoretical results are no longer true for disequations. Therefore, we need a different treatment to handle disequations. Nevertheless, automatic tools like HIP/SLEEK actually have disequations in their proofs, *e.g.*, $\neg(\pi = \circ)$, to assert that a share is positive as a side condition for maps-to $x \overset{\pi}{\mapsto} v$.

In this chapter, we introduce two decision procedures to handle **SAT** and **IMP** with disequations. To tackle this problem, we provide several new key theoretical results that help shape the correctness of our new procedures. To distinguish from the old procedures, we will refer our new procedures as GSAT and GIMP as for "general satisfiability" and "general implication". Correspondingly, two new problems are called **GSAT** and **GIMP**. We make another major contribution by implementing, optimizing and certifying GSAT and GIMP in Coq native environment. Lastly, our procedures are extracted into OCaml and benchmarked in HIP/SLEEK with better performance compared to the old tools.

We divide the chapter into the following sections*:

1. In §5.1, we introduce two new problems over share equation systems with disequations, namely the general satisfiability (**GSAT**) and general implication (**GIMP**).
2. In §5.2, we overview the architecture of our two solvers GSAT and GIMP.
3. In §5.3 and §5.4, we explain the mechanism of our solvers together with illustrated examples.
4. In §5.5, we carry out the formal soundness proof of our solvers.
5. In §5.6, we discuss several performance-enhancing components that help optimize the solvers.
6. In §5.7, we report our benchmarking in HIP/SLEEK.
7. In §5.8, we summarize the implemented Coq files together with the certified proof.
8. In §5.9, we draw our conclusion.

*The materials in this chapter are taken from the paper "A Certified Decision Procedure for Tree Shares" [LNHC17], a joint work with Thanh-Toan Nguyen, Aquinas Hobor and Wei-Ngan Chin.

5.1 Disequations over shares and their motivative problems

5.1.1 Disequations over tree shares

Recall from §4.1.1 that, given a SL entailment $P \vdash Q$ with fractional permissions, we can extract a heap constraint $H_1 \vdash H_2$ and a share constraint $\Sigma_1 \vdash \Sigma_2$ in which Σ_1, Σ_2 are share equation systems. On one hand, there are standard techniques and proof systems in SL that help handle the heap constraint. On the other hand, it is not obvious how to solve these share constraints, especially when the tree share domain is infinite and thus trivial brute-force is not applicable. Often, automatic tools also require shares to be positive (not $\circ$) to indicate the read permission. This condition is implicitly assumed in the SL entailments and only becomes explicit when the share solver is called.

Example 5.1.1. The entailment $x \overset{v_1}{\mapsto} 1 \ * \ y \longmapsto 2 \ \vdash \ x \overset{v_2}{\mapsto} 1 \ * \top$ yields the following entailment over tree shares (where v_1, v_2, v_3 are universal variables):

$$v_1 \neq \circ \ \wedge \ v_3 = \begin{array}{c} \frown \\ \circ \ \ \bullet \end{array} \ \vdash \ \ \exists v_4.\ v_2 \oplus v_4 = v_1 \ \wedge \ v_2 \neq \circ.$$

Notice that inequality $a \neq b$ is a special case of disequation, *i.e.*, $\neg(a \oplus \circ = b)$. This share entailment is valid because we can choose $v_2 = v_1$ and $v_4 = \circ$. ◁

Tree constraints are nontrivial to solve because the search space is infinite, *e.g.*, $v_1 \oplus v_2 = \bullet$ has infinitely many solutions $(v_1, v_2) \in \{(\bullet, \circ), (\begin{array}{c} \frown \\ \bullet \end{array}, \begin{array}{c} \frown \\ \circ \ \ \bullet \end{array}), \ldots\}$. Thus in order to solve these constraints, it is essential to obtain theoretical insights about the tree structures. In Chapter 4, we proved that $\oplus$-equations for **SAT** and **IMP** satisfy the small model property (Theorems 4.2.1 and 4.2.3) in which the search space can be restricted to trees of height at most the height of the constraint. Unfortunately, this key result is no longer true for $\oplus$-disequation as demonstrated below.

Example 5.1.2. Consider the constraint of height zero $x \oplus y = \bullet \ \wedge \ x \neq \circ \ \wedge \ y \neq \circ$. As one might check, it has no solution of height zero. But it has solution of height one (and

higher), *e.g.*, $x =$  and $y =$. ◁

We overload the share equation system Σ to contain disequations:

Definition 5.1.1. A share equation system Σ is a quadruple $(l^{\exists}, l^{=}, l^{+}, l^{-})$ in which:

1. $l^{\exists}$ is the list of existential variables.
2. $l^{=}$ is the list of equalities $\pi_1 = \pi_2$.
3. l^{+} is the list of equations $\pi_1 \oplus \pi_2 = \pi_3$.
4. l^{-} is the list of disequations $\neg(\pi_1 \oplus \pi_2 = \pi_3)$.

For **convenience**, we will usually illustrate an equation system as $\Sigma = \{x_1, \ldots, x_n, g_1, \ldots, g_m\}$ in which x_i is an existential variable and g_i is either equality, equation or disequation. Moreover, the definitions of context and solution remain unchanged as in Definition 4.1.2. For convenience, there is a **small change** in notation, namely we will use ρ instead of S for context. ◁

Example 5.1.3. The constraint in Example 5.1.2 is represented by the equation system $\Sigma = \{x \oplus y = \bullet, x \neq \circ, y \neq \circ\}$. The context $\rho = \{x =$ $, y =$  $\}$ is a solution of Σ. ◁

5.1.2 Problem formulation

Here we provide formal descriptions of the two problems over tree shares, namely the general satisfiability **GSAT** and general implication **GIMP**. Let $\Sigma, \Sigma_1, \Sigma_2$ be share equation systems defined in 5.1.1. We are interested in constructing sound and complete procedures to handle the following queries:

1. **GSAT**(Σ): Is Σ satisfiable, *i.e.*, is there a solution for Σ:

$$\exists \rho.\ \rho \models \Sigma.$$

2. **GIMP**(Σ_1, Σ_2): Does Σ_1 entail Σ_2, *i.e.*, are all solutions of Σ_1 also solutions of Σ_2:

$$\Sigma_1 \vdash \Sigma_2 \ \text{ iff } \ \forall \rho.\ \rho \models \Sigma_1 \ \rightarrow \ \rho \models \Sigma_2.$$

Example 5.1.4. The entailment in Example 5.1.2 is an instance of **GIMP**:

$$\{v_1 \neq \circ, v_3 = \overset{\frown}{\circ\ \bullet}\} \vdash \{v_4, v_2 \oplus v_4 = v_1, v_2 \neq \circ\}.$$

◁

Despite allowing negative clauses, entailment is not subsumed by satisfiability because of the quantifier alternation $\forall\exists$ in the consequent. One interesting exercise is to examine the metatheoretical properties of tree shares given in Figure 2.3. Several of these are the standard properties of separation algebras [COY07], but others are part of what make the tree share model special. In particular, tree shares are the only model of fractional permissions that simultaneously satisfy Disjointness (forces the tree predicate—equation 1.3—to behave properly), Cross-split (used *e.g.* in settings involving overlapping data structures), and Infinite splittability (to verify divide-and-conquer algorithms). Encouragingly, all of the axioms except for "Unit" are expressible as entailments in our format; *e.g.* associativity is:

$$\{x \oplus a = b, y \oplus z = a\} \quad \vdash \quad \{c, x \oplus y = c, c \oplus z = b\}.$$

Unit requires the order of quantifiers to swap; our format can express the weaker "Multi-unit axiom" $\forall x.\ \exists u.\ x \oplus u = x$ as well as $\forall x.\ x \oplus \circ = x$:

$$\emptyset \vdash \{u, x \oplus u = x\} \quad \text{and} \quad \emptyset \vdash \{x \oplus \circ = x\}.$$

where $\emptyset$ indicates the empty share equation system.

5.2 Overview of our decision procedures

5.2.1 The architecture of GSAT and GIMP

For convenience, we use **GSAT** and **GIMP** to refer to the problems GSAT and GIMP for the decision procedures themselves. Although we are mainly interested in the construction of the entailment checker GIMP (as the tree share constraints extracted from HIP/SLEEK are

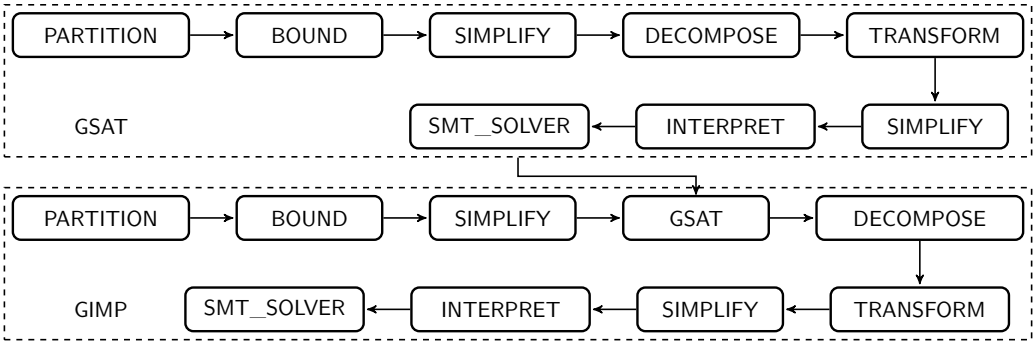


Figure 5.1: Two decision procedures GSAT and GIMP implemented in Coq

of entailment form), we also need the satisfiability checker GSAT for two main reasons. First, GSAT helps to prune the search space; *e.g.*, if the antecedent Σ_1 for GIMP is unsatisfiable, we can immediately conclude $\Sigma_1 \vdash \Sigma_2$. Second, the correctness of some subroutines in $\mathsf{GIMP}(\Sigma_1, \Sigma_2)$ requires the outcome of $\mathsf{GSAT}(\Sigma_1)$ as one of the critical conditions.

The architecture of our system is given in Figure 5.1. We have two procedures to solve problems over share formulas, one for satisfiability and the other for entailment, both written in Gallina and certified in Coq. Identically-named components in the two procedures are similar in spirit but not identical in operation; thus for example there are two different SIMPLIFY components, one for GSAT and another for GIMP. The PARTITION, BOUND, and SIMPLIFY components substantially improve the performance of our procedures in practice but are not complete solvers: in the worst case they do nothing. Since they are included for performance we will discuss them in more detail in §5.6.

The DECOMPOSE and TRANSFORM components form the heart of our procedure. While the $\oplus$ operation has many useful properties that enable sophisticated reasoning about shared ownership in program verifications (*e.g.* Figure 2.3), they are not strong enough for techniques like Gaussian elimination (which even in $\mathbb{Q}$ cannot handle negative clauses). In §5.5 we will describe DECOMPOSE in detail after developing the necessary theory. Briefly, DECOMPOSE takes a system of equations with constants of arbitrary complexity and eventually produces a much larger equivalent system in which each constant is either $\circ$ or $\bullet$ (*i.e.*, the final system has *height* zero).

TRANSFORM is a very sophisticated component mathematically, yet also the simplest computationally: it just changes the **type** of the system. That is, it inputs a **tree** system of height zero and outputs an equivalent, essentially identical **Boolean** system. The only actual computational content is by swapping $\circ$ for $\bot$ and $\bullet$ for $\top$. Recall from §2.2.1, the join relation on Booleans is simply disjoint disjunction:

$$\top \oplus \bot = \top \qquad \bot \oplus \top = \top \qquad \bot \oplus \bot = \bot.$$

The last option, $\top \oplus \top$, is undefined.

INTERPRET translates Boolean systems of equations into Boolean sentences by rewriting equations and disequations using the rules:

$$\begin{aligned} \pi_1 \oplus \pi_2 = \pi_3 &\rightsquigarrow (\pi_1 \wedge \neg\pi_2 \wedge \pi_3) \vee (\neg\pi_1 \wedge \pi_2 \wedge \pi_3) \vee (\neg\pi_1 \wedge \neg\pi_2 \wedge \neg\pi_3) \\ \neg(\pi_1 \oplus \pi_2 = \pi_3) &\rightsquigarrow (\neg\pi_1 \vee \pi_2 \vee \neg\pi_3) \wedge (\pi_1 \vee \neg\pi_2 \vee \neg\pi_3) \wedge (\pi_1 \vee \pi_2 \vee \pi_3) \end{aligned}$$

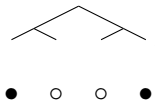
Next, it adds the appropriate quantifiers depending on the query type to reach a closed sentence. INTERPRET's code and correctness proof are straightforward.

SMT_SOLVER uses simple quantifier elimination to check the validity of boolean sentences. Our SMT solver is rather naïve, and thus is the performance bottleneck of our tool, but we could not find a suitable Gallina alternative. As discussed in §5.6, despite its naiveté our overall performance seems acceptable in practice due to the heuristics in PARTITION, BOUND, and SIMPLIFY.

5.2.2 Basic notations and definitions

We use nil to denote the empty list, $[e_1, \ldots, e_n]$ to represent a list's content, and $l +\!\!+ l'$ for list concatenation. We use the metavariable η to represent a single disequation. The symbols Σ and Π are reserved for systems and pairs of systems respectively; if the exact form of our systems is not important or is clear from the context, we may refer it as Γ. The symbol ρ and S are for contexts and solutions respectively. We use $|\tau|$ to indicate the height of τ.

Also, we will overload the height function $|\cdot|$ for equation systems and contexts to indicate the height of the highest tree constant. For a tree τ, we let τ_l and τ_r be the left and right sub-trees of τ, *i.e.*, $\tau = \tau_l = \tau_r$ if $\tau \in \{\circ, \bullet\}$ and $\tau =$  otherwise.

Example 5.2.1. Let $\Sigma_1 = \{x \oplus$  $= y\}$, $\Sigma_2 = \{y \neq \circ\}$ and $\Pi = (\Sigma_1, \Sigma_2)$ then $|\Sigma_1| = 1$, $|\Sigma_2| = 0$ and $|\Pi| = 1$. Also, the context $\rho = \{x = \circ, y =$  $\}$ is a solution of Σ_1. For the tree $\tau =$  , its left and right subtree are $\tau_l =$  and $\tau_r =$  . ◁

We define several basic systems for **GSAT** and **GIMP** as the building blocks of the decision procedures:

Definition 5.2.1. Let $\Sigma, \Sigma_1, \Sigma_2$ be share equation systems and η, η_1, η_2 disequations. Let l be a list of disequations. We define Σ^l to be the new equation system in which the disequation list in Σ is replaced with l. For convenience, we write Σ^η as shortcut for $\Sigma^{[\eta]}$, and Σ^+ as shortcut for Σ^{nil}. Then:

1. If the disequation list in Σ is empty then Σ is called a *positive system*.
2. If there is exactly one disequation in Σ then Σ is called a *singleton system*.
3. If Σ_1 is positive and Σ_2 is singleton then (Σ_1, Σ_2) is called a *Z-system*.
4. If both Σ_1 and Σ_2 are singleton then (Σ_1, Σ_2) is called a *S-system*.

In particular, Σ^+ is always a positive system, Σ^η is always a singleton system, $(\Sigma_1^+, \Sigma_2^\eta)$ is always a Z-system, and $(\Sigma_1^{\eta_1}, \Sigma_2^{\eta_2})$ is always an S-system. ◁

Example 5.2.2. Let $\Sigma_1 = \{x \oplus y = z, x \neq \circ\}$, $\Sigma_2 = \{w, z \oplus w = \bullet, z \neq \circ\}$, $\eta_1 : y \neq \circ$, $\eta_2 : w \neq \circ$ then:

1. $\Sigma_1^+ = \{x \oplus y = z\}$ and $\Sigma_1^{\eta_1} = \{x \oplus y = z, y \neq \circ\}$.
2. $\Sigma_2^+ = \{w, z \oplus w = \bullet\}$ and $\Sigma_2^{\eta_2} = \{w, z \oplus w = \bullet, w \neq \circ\}$.

◁

5.3 Decision procedure GSAT

5.3.1 Overview of GSAT

Algorithm 4 Solver GSAT for systems with disequations

1: **function** GSAT(Σ)
2: **if** SAT(Σ^+) $= \bot$ **then return** $\bot$
3: **else if** $l^- =$ nil **then** ▷ l^- is the disequation list in Σ
4: **return** $\top$
5: **else** let $l^- = [\eta_1, \ldots, \eta_n]$
6: **return** $\bigwedge_{i=1}^{n}$ SSAT(Σ^{η_i})
7: **end if**
8: **end function**

We propose the procedure GSAT to solve **GSAT** of equation systems with disequations. For **GSAT**(Σ), the existential list in Σ is redundant and thus will be ignored. Our new decision procedure GSAT also makes use of the old decision procedure SAT from §4.2 for systems without disequations, *e.g.*, positive systems. To help the readers gain intuition of the procedure, we will abstract away all the tedious low-level implementations and only discuss about the high-level structure. The execution of GSAT consists of two major steps which are described in Algorithm 4. At first, the system Σ is separated into a list of singleton systems; each contains a single disequation taken from the disequation list of Σ. In the second step, each singleton system is solved individually using the subroutine SSAT, then their results are conjoined to determine the result of GSAT(Σ).

Algorithm 5 Solver SSAT for singleton systems

1: **function** SSAT(Σ^η)
Require: Σ^η is singleton and Σ^+ is satisfiable
2: $[\Sigma_1, \ldots, \Sigma_n] \leftarrow$ DECOMPOSE(Σ^η)
3: transform each Σ_i into Boolean formula Φ_i
4: $\Phi \leftarrow \bigvee_{i=1}^{n} \Phi_i$
5: **return** SMT_SOLVER(Φ)
6: **end function**

The solver SSAT for singleton system (Algorithm 5) calls another subroutine DECOMPOSE (Algorithm 6) that helps decompose a share system into sub-systems of height zero. These subs-systems subsequently go though a two-phase process to be transformed into boolean formulas. In the first phase, the subroutine TRANSFORM trivially converts tree type into

boolean type using the conversions $\bullet \rightsquigarrow \top$ and $\circ \rightsquigarrow \bot$. Correspondingly, the share system is converted into the boolean system. In the second phase, the subroutine INTERPRET helps to interpret the boolean system into an equivalent boolean formula by adding necessary quantifiers ($\exists$ for **GSAT**, $\forall$ for **GIMP**) and conjunctives among equations and disequations. For skeptical readers, the correctness of GSAT is mentioned in Theorem 5.3.1 and its proof is verified completely in Coq.

Theorem 5.3.1 ([Sol])**.** Let Σ be a share system then Σ is satisfiable iff $\mathsf{GSAT}(\Sigma) = \top$. $\triangleleft$

Algorithm 6 Decompose system into sub-systems of height zero

```
function DECOMPOSE(Γ)
Require: Γ is either one system (GSAT) or pair of systems (GIMP)
Ensure: A list of sub-systems of height zero
    if |Γ| = 0 then return [Γ]
    else
        (Γ1, Γ2) ← SINGLE_DECOMPOSE(Γ)
        return DECOMPOSE(Γ1) ++ DECOMPOSE(Γ2)
    end if
end function

function SINGLE_DECOMPOSE(Γ)
Require: Γ is either one system (GSAT) or pair of systems (GIMP)
Ensure: A pair of left and right sub-system
    if |Γ| = 0 then return (Γ, Γ)
    else
        Γl ← replace each tree constant τ in Γ with its left sub-tree τl
        Γr ← replace each tree constant τ in Γ with its right sub-tree τr
        return (Γl, Γr)
    end if
end function
```

5.3.2 Example of GSAT

Let $\Sigma = \{v_1 \oplus v_2 = \bullet, v_1 \neq \begin{smallmatrix}\frown\\ \bullet\ \circ\end{smallmatrix}, v_2 \neq \circ\}$ then **GSAT**(Σ) is equivalent to the formula:

$$\Phi \;=\; \exists v_1 \exists v_2.\; v_1 \oplus v_2 = \bullet \;\wedge\; v_1 \neq \begin{smallmatrix}\frown\\ \bullet\ \circ\end{smallmatrix} \;\wedge\; v_2 \neq \circ.$$

The formula Φ is valid because the interpretation $v_1 = \begin{smallmatrix}\frown\\ \circ\ \bullet\end{smallmatrix}$, $v_2 = \begin{smallmatrix}\frown\\ \bullet\ \circ\end{smallmatrix}$ satisfies Φ.

We now simulate the execution of GSAT using Σ as input. First, the old solver SAT is called to check the system $\Sigma^+ = \{v_1 \oplus v_2 = \bullet\}$ which is equivalent to $\Phi^+ = \exists v_1 \exists v_2.\ v_1 \oplus v_2 = \bullet$. As Σ^+ is satisfiable (*e.g.* $v_1 = \circ$, $v_2 = \bullet$) and there are two disequations in Σ, we split Σ into two singleton systems:

$$\Sigma^1 = \{v_1 \oplus v_2 = \bullet, v_1 \neq \overset{\frown}{\bullet\ \circ}\} \text{ and } \Sigma^2 = \{v_1 \oplus v_2 = \bullet, v_2 \neq \circ\}.$$

When the solver $\mathsf{SSAT}(\Sigma^1)$ is called, the singleton system Σ^1 is split into two sub-systems of height zero Σ^1_1 and Σ^1_2 by $\mathsf{DECOMPOSE}$:

$$\Sigma^1_1 = \{v_1 \oplus v_2 = \bullet, v_1 \neq \bullet\} \text{ and } \Sigma^1_2 = \{v_1 \oplus v_2 = \bullet, v_1 \neq \circ\}.$$

In the next step, both Σ^1_1 and Σ^1_2 are transformed into Boolean formulas Φ^1_1 and Φ^1_2:

$$\begin{aligned} \Phi^1_1 &= \exists v_1 \exists v_2.\ ((v_1 \wedge \neg v_2) \ \vee\ (\neg v_1 \wedge v_2)) \ \wedge\ \neg v_1. \\ \Phi^1_2 &= \exists v_1 \exists v_2.\ ((v_1 \wedge \neg v_2) \ \vee\ (\neg v_1 \wedge v_2)) \ \wedge\ v_1. \end{aligned}$$

As both Φ^1_1 and Φ^1_2 are valid, we have $\mathsf{SSAT}(\Sigma^1) = \top$. Similarly, one can verify that $\mathsf{SSAT}(\Sigma^2) = \top$. Hence, by combining the two results, we get $\mathsf{GSAT}(\Sigma) = \top$.

We finish §5.3 by pointing out a decidability result of $\oplus$:

Corollary 5.3.1. The $\exists$-theory of the structure $\langle \mathbb{T}, \oplus \rangle$ is decidable. ◁

Proof. Let Ψ be a quantifier-free formula, we transform Ψ into Disjunctive Normal Form $\bigvee_{i=1}^{n} \Psi_i$ then each Ψ_i can be represented as a system Σ_i. Consequently, Ψ is satisfiable iff some Σ_i is satisfiable which can be solved by GSAT. Thus the result follows. □

Algorithm 7 Solver GIMP for entailment of share systems with disequations

1: **function** IMP(Σ_1, Σ_2)
2: **if** GSAT$(\Sigma_1) = \bot$ **then return** $\bot$
3: **else if** IMP$(\Sigma_1^+, \Sigma_2^+) = \bot$ **then return** $\bot$
4: **else** let l_1^-, l_2^- be disequation lists of Σ_1, Σ_2
5: **if** $l_2^- =$ nil **then return** $\top$
6: **else** let $l_2^- = [\eta_2^1, \ldots, \eta_2^n]$
7: **if** $l_1^- =$ nil **then return** $\bigwedge_{i=1}^n$ ZIMP$(\Sigma_1^+, \Sigma_2^{\eta_2^i})$
8: **else** let $l_1^- = [\eta_1^1, \ldots, \eta_1^m]$
9: **for** $i = 1 \ldots n$ and $j = 1 \ldots m$ **do**
10: let $Z_i \leftarrow$ ZIMP$(\Sigma_1^+, \Sigma_2^{\eta_2^i})$ and $S_i^j \leftarrow$ SIMP$(\Sigma_1^{\eta_1^j}, \Sigma_2^{\eta_2^i})$
11: **end for**
12: **return** $\bigwedge_{i=1}^n \left(Z_i \vee (\bigvee_{j=1}^m S_i^j)\right)$
13: **end if**
14: **end if**
15: **end if**
16: **end function**

5.4 Decision procedure GIMP

5.4.1 Overview of GIMP

Our GIMP procedure (Algorithm 7) deploys a similar strategy as for GSAT by reducing the entailment into several entailments of the basic systems (*e.g.* Z-system and S-system). In particular, GIMP verifies the entailment $\Sigma_1 \vdash \Sigma_2$ by first calling two solvers GSAT(Σ_1) and IMP(Σ_1^+, Σ_2^+) (line 2 and 3). Then the lengths of the two disequation lists (l_1^- in Σ_1 and l_2^- in Σ_2) critically determine the subsequent flow of GIMP. In detail, there are three different cases of l_1^- and l_2^- that fully cover all the scenarios:

1. If the list l_2^- is empty (line 5) then the answer is equivalent to IMP(Σ_1^+, Σ_2^+), *i.e.* $\top$.

2. Otherwise, we check whether l_1^- is empty (line 7) in which the answer is conjoined from several entailments of Z-systems* $(\Sigma_1, \Sigma_2^{\eta_2^i})$; each is constructed from (Σ_1, Σ_2) by removing all disequations in Σ_2 except one. Here we call the subroutine ZIMP which is a specialized solver for entailment of Z-systems.

*Recall from Definition 5.1.1 that a Z-system is a pair of share systems (Σ_a, Σ_b) in which Σ_a has no disequation while Σ_b has exactly one disequation

3. The third case is when neither l_1^- nor l_2^- is empty (line 8). Then the result of $\Sigma_1 \vdash \Sigma_2$ is computed by taking the conjunction of several entailments of Z-systems and S-systems*. We use SIMP as a specialized solver to check entailment for S-systems.

Algorithm 8 Solvers for entailment of Z-systems and S-systems

```
1: function ZIMP(Σ1, Σ2)
Require: (Σ1, Σ2) is Z-system, Σ1 is satisfiable and Σ1 ⊢ Σ2+
2:     [Γ1, ..., Γn] ← DECOMPOSE(Σ1, Σ2)
3:     transform each Γi into Boolean formula Φi
4:     Φ ← ⋁_{i=1}^{n} Φi
5:     return SMT_SOLVER(Φ)
6: end function
7:
8: function SIMP(Σ1, Σ2)
Require: (Σ1, Σ2) is S-system, Σ1+ is satisfiable, Σ1+ ⊢ Σ2+ and Σ1+ ⊬ Σ2
9:     [Γ1, ..., Γn] ← DECOMPOSE(Σ1, Σ2)
10:    transform each Γi into Boolean formula Φi
11:    Φ ← ⋀_{i=1}^{n} Φi
12:    return SMT_SOLVER(Φ)
13: end function
```

Two specialized solvers ZIMP and SIMP are described in Algorithm 8. For ZIMP, we first call the subroutine DECOMPOSE to split the Z-system into sub-systems of height zero. Next, each sub-system is transformed in to Boolean formula by adding necessary quantifiers and logical connectives. These Boolean formulas are then combined using disjunctions to form a single boolean formula; and this formula is solved using standard SMT solvers to determine the result of the entailment. The procedure for SIMP has a similar structure, except that the final Boolean formula is formed using conjunctions. Also, it is worth noticing that there are certain preconditions for both solvers; and all of them are important to shape the correctness of the solvers. The correctness of GIMP is mentioned in Theorem 5.4.1; and its proof is verified entirely in Coq.

Theorem 5.4.1 ([Sol])**.** Let Σ_1, Σ_2 be share systems then $\Sigma_1 \vdash \Sigma_2$ iff $\mathsf{GIMP}(\Sigma_1, \Sigma_2) = \top$. ◁

*Recall from Definition 5.1.1 that a S-system is a pair of share systems (Σ_a, Σ_b) in which each Σ_a and Σ_b has exactly one disequation

5.4.2 Example of GIMP

The infinite splitability of tree share in Figure 2.3:

$$\forall v.\ (v \neq \circ \ \rightarrow\ \exists v_1 \exists v_2.\ v_1 \oplus v_2 = v\ \wedge\ v_1 \neq \circ\ \wedge\ v_2 \neq \circ).$$

can be represented as the entailment $\Sigma_1 \vdash \Sigma_2$ s.t.:

$$\Sigma_1 = \{v \neq \circ\} \quad \vdash \quad \Sigma_2 = \{v_1, v_2, v_1 \oplus v_2 = v, v_1 \neq \circ, v_2 \neq \circ\}.$$

We will simulate the execution of GIMP using (Σ_1, Σ_2) as input. This pair of system goes though Algorithm 7 until line 8 as both disequation lists are nonempty. As there are two disequations in Σ_2, namely $\eta_1 : v_1 \neq \circ$ and $\eta_2 : v_2 \neq \circ$, we need to verify the conjunction $P_1 \wedge P_2$ s.t.:

$$P_1 = \mathsf{ZIMP}(\Sigma_1^+, \Sigma_2^{\eta_1}) \vee \mathsf{SIMP}(\Sigma_1, \Sigma_2^{\eta_1}) \text{ and } P_2 = \mathsf{ZIMP}(\Sigma_1^+, \Sigma_2^{\eta_2}) \vee \mathsf{SIMP}(\Sigma_1, \Sigma_2^{\eta_2}).$$

For P_1, $\mathsf{ZIMP}(\Sigma_1^+, \Sigma_2^{\eta_1})$ is equivalent to the validity of the formula:

$$\forall v.\ (\top \rightarrow \exists v_1 \exists v_2.\ v_1 \oplus v_2 = v\ \wedge\ v_1 \neq \circ).$$

Informally, the above sentence is invalid by choosing $v = \circ$ to force both v_1 and v_2 be $\circ$. Formally, it is transformed into the following boolean formula:

$$\forall v.\ (\top \rightarrow \exists v_1 \exists v_2.\ [(v_1 \wedge \neg v_2 \wedge v_3) \vee (\neg v_1 \wedge v_2 \wedge v_3) \vee (\neg v_1 \neg v_2 \neg v_3)]\ \wedge\ \neg v_1).$$

which is reported to be invalid by SMT solver. Likewise, $\mathsf{SIMP}(\Sigma_1, \Sigma_2^{\eta_1})$ is equivalent to the validity of the formula:

$$\forall v.\ (v \neq \circ\ \rightarrow\ \exists v_1 \exists v_2.\ v_1 \oplus v_2 = v\ \wedge\ v_1 \neq \circ).$$

which is transformed into the Boolean formula:

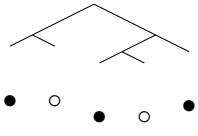
$$\Phi_1 \;=\; \forall v.\ (v \rightarrow \exists v_1 \exists v_2.\ ((\neg v_1 \wedge \neg v_2 \wedge \neg v) \vee (v_1 \wedge \neg v_2 \wedge v) \vee (\neg v_1 \wedge v_2 \wedge v)) \wedge v_1).$$

As Φ_1 is valid, P_1 is true. Similarly, one may check that P_2 also holds and thus $\Sigma_1 \vdash \Sigma_2$. □

Having described the heart of our decision procedures, what remains is to describe the practical aspects of their development and evaluation. In §5.5, we provide the key theoretical results that shape the correctness of GSAT and GIMP. In §5.6 we describe various techniques that enable good performance in practice. In §5.7 we describe how we benchmarked our tool running inside Coq, running as a standalone compiled program, and after incorporating it into the HIP/SLEEK verification toolset. In §5.8 we document the files in the development itself; we have approximately 38.6k lines of code in 31 files.

5.5 Correctness of GSAT and GIMP

The correctness of our decision procedures makes use of rounding functions and averaging function defined in §4.3. We recall *left rounding* $\lfloor \overleftarrow{\tau} \rfloor_n$ (*right rounding* $\lfloor \overrightarrow{\tau} \rfloor_n$), which unfolds τ into full binary tree of height n, removes all left (or right) leaves and then folds it back to canonical form; and *averaging* $\tau_1 \bigtriangledown_n \tau_2$, which unfolds τ_1, τ_2 into full binary trees of height $n-1$, combines their leaves pairwise and then folds the combined tree back to canonical form. We introduce for clarity two new functions: *pair rounding* $\lfloor \tau \rfloor_n = (\lfloor \overleftarrow{\tau} \rfloor_n, \lfloor \overrightarrow{\tau} \rfloor_n)$ and *combine* $\mathsf{Combine}(\tau_1, \tau_2)$, which is simply the inverse function of Split in §2.2.2.

We illustrate these functions in Figure 5.2 to help readers gain intuition of these functions' mechanism. Our example is the tree  at height 3 together with its derived trees. To help track what is going on in $\lfloor \overleftarrow{\tau} \rfloor_n$, $\lfloor \overrightarrow{\tau} \rfloor_n$, and $\tau_1 \bigtriangledown_{\tau_2} n$ we have highlighted the left leaf in each pair with the color red and the right leaf in each pair with the color blue. For $\mathsf{Split}(\tau)$ and $\mathsf{Combine}(\tau_1, \tau_2)$, we color red for leaves from the left subtree and blue for leaves from the right subtree.

Lemma 5.5.1 ([Sol])**.** Let $\mathbb{T}_n$ be the set of tree shares of height at most n. We overload

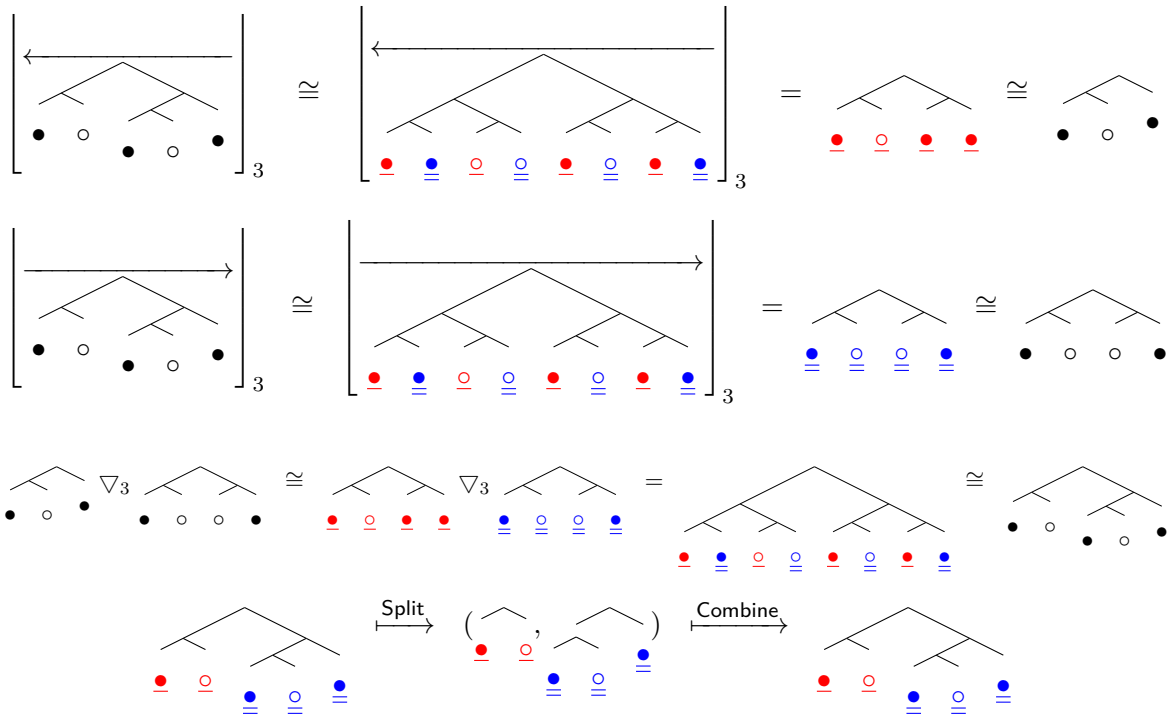


Figure 5.2: Illustrated examples of applying the tree operators

the join $\oplus$ over the domain $\mathbb{T}_n \times \mathbb{T}_m$ by applying the normal join component-wise, *i.e.*:

$$(\tau_1, \tau_2) \oplus (\tau_1', \tau_2') \quad \stackrel{\text{def}}{=} \quad (\tau_1 \oplus \tau_1', \tau_2 \oplus \tau_2').$$

Then both the pair rounding function $\lfloor \cdot \rfloor_{n+1} : \langle \mathbb{T}_{n+1}, \oplus \rangle \mapsto \langle \mathbb{T}_n \times \mathbb{T}_n, \oplus \rangle$ and split function $\mathsf{Split} : \langle \mathbb{T}_{n+1}, \oplus \rangle \mapsto \langle \mathbb{T}_n \times \mathbb{T}_n, \oplus \rangle$ are isomorphisms, namely they are both bijections $\mathbb{T}_{n+1} \mapsto \mathbb{T}_n \times \mathbb{T}_n$, and they preserve the join $\oplus$:

$$\tau_1 \oplus \tau_2 = \tau_3 \;\text{ iff }\; \mathsf{Split}(\tau_1) \oplus \mathsf{Split}(\tau_2) = \mathsf{Split}(\tau_3) \;\text{ iff }\; \lfloor \tau_1 \rfloor_{n+1} \oplus \lfloor \tau_2 \rfloor_{n+1} = \lfloor \tau_3 \rfloor_{n+1}.$$

Furthermore:

1. If $|\tau| \leq n$ then $\lfloor\ \tau\ \rfloor_{n+1} = (\tau, \tau)$.
2. If $|\tau| = n + 1$ then $|\ \lfloor \overleftarrow{\tau} \rfloor_{n+1}\ | < |\tau|$ and $|\ \lfloor \overrightarrow{\tau} \rfloor_{n+1}\ | < |\tau|$.
3. Let $\mathsf{Split}(\tau) = (\tau_l, \tau_r)$. If $|\tau| = 0$ then $\tau_l = \tau_r = \tau$, otherwise $|\tau_l| < |\tau|$ and $|\tau_r| < |\tau|$.

4. $\triangledown_{n+1}$ is the inverse function of $\lfloor\cdot\rfloor_{n+1}$.

◁

Proof intuition. By induction on the tree height. □

Although both the pair rounding function $\lfloor\cdot\rfloor_n$ and split function Split are isomorphism, they are critically different in terms of functionality. In brief, the function $\lfloor\cdot\rfloor_n$ only affects trees of height n whereas the function Split affects all trees except $\bullet$ and $\circ$. As a result, $\lfloor\cdot\rfloor_n$ helps us 'refine' big solutions into smaller ones while $\mathsf{Combine}$ is used to decompose big systems into smaller ones:

Corollary 5.5.1 ([Sol])**.** Let ρ be a context of Σ then:

1. If $n \geq |\rho| > |\Sigma|$ and $\lfloor\rho\rfloor_n = (\rho_l, \rho_r)$ then $\rho \models \Sigma$ iff both $\rho_l \models \Sigma$ and $\rho_r \models \Sigma$.
2. If $n > |\rho|$ then $\rho_l = \rho_r = \rho$, otherwise if $n = |\rho|$ then $|\rho_l| < n$ and $|\rho_r| < n$.
3. If $\mathsf{Split}(\Sigma) = (\Sigma_l, \Sigma_r)$ and $\mathsf{Split}(\rho) = (\rho_l, \rho_r)$ then $\rho \models \Sigma$ iff both $\rho_l \models \Sigma_l$ and $\rho_r \models \Sigma_r$.
4. If $|\rho| = 0$ then $\rho_l = \rho_r = \rho$, otherwise $|\rho_l| < |\rho|$ and $|\rho_r| < |\rho|$. If $|\Sigma| = 0$ then $\Sigma_l = \Sigma_r = \Sigma$, otherwise if $|\Sigma_l| < |\Sigma|$ and $|\Sigma_r| < |\Sigma|$.

◁

Proof intuition. These results are directly generalized from Lemma 5.5.1. □

5.5.1 Domain reduction

We propose the *domain reduction* technique to reduce the search space from infinite to finite.

Lemma 5.5.2. We use uppercase letters to denote sets, *e.g.*, T and S. Then:

1. **Emptiness**: Let $T \subseteq S$ and $f : S \mapsto T$. Then S is empty iff T is empty

   Generally, let $T_i \subseteq T_{i+1}$ and $f_i : T_{i+1} \mapsto T_i$ for $i = 1 \ldots n$ then

   $$\bigcup_{i=1}^{n+1} T_i \text{ is empty} \quad \text{iff} \quad T_1 \text{ is empty.}$$

2. **Inclusion**: Let $T \subseteq S$ and $f : S \mapsto T^k$ a k-ary bijection s.t. $f^{-1}((T \cap S')^k) \subseteq S'$ then:

$$S \subseteq S' \quad \text{iff} \quad T \subseteq S'.$$

Generally, let $T_i \subseteq T_{i+1}$ and $f_i : T_{i+1} \mapsto T_i^{k_i}$ be k_i-ary bijection for $i = 1 \ldots n$ s.t. $f_i^{-1}((T_i \cap S')^{k_i}) \subseteq S'$ then

$$T_{n+1} = \bigcup_{i=1}^{n+1} T_i \ \subseteq \ S' \quad \text{iff} \quad T_1 \ \subseteq \ S'.$$

◁

Proof. The emptiness problem is trivial so we only focus on the inclusion problem. Only the $\Leftarrow$ direction is nontrivial: assume $T \subseteq S'$ and let $x \in S$ then $f(x) \in f(S) = T^k$, Thus $f(x) \in T^k \cap S'^k = (T \cap S')^k$ which implies $x \in f^{-1}((T \cap S')^k) \subseteq S'$. This concludes $S \subseteq S'$. The general case is done by induction over n. In detail, the base case $n = 0$ is trivial. Assume $T_k \subseteq S'$ iff $T_1 \subseteq S'$. The previous result gives us $T_{k+1} \subseteq S'$ iff $T_k \subseteq S'$. Thus $T_{k+1} \subseteq S'$ iff $T_1 \subseteq S'$ and the result follows by induction principle. □

For **convenience**, we denote $S(\Sigma)$ to be the set of all solutions of Σ, and $S_i(\Sigma) \subseteq S(\Sigma)$ to be the set of all solutions of height at most i.

Proof of Theorem 4.2.1 and 4.2.3. To demonstrate the usefulness of our domain reduction technique, we will use it to simplify the proof of finite height for **SAT** and **IMP** in Chapter 4. Notice that the share systems do not contain disequations yet.

Let $n = |\Sigma|$, we classify the solution space of Σ into $\{S_i(\Sigma)\}_{i=n}^{\infty}$ according to their heights. Then it follows that $S_i(\Sigma) \subseteq S_{i+1}(\Sigma)$ and $\lfloor \overleftarrow{\rho} \rfloor_{i+1}$ maps solutions in $S_{i+1}(\Sigma)$ (*i.e.* at most height $i+1$) into solutions in $S_i(\Sigma)$ (*i.e.* at most height i). Thus by emptiness property in Lemma 5.5.2, $\mathbf{SAT}(\Sigma)$ iff there exists a solution in $S_n(\Sigma)$.

For $\mathbf{IMP}(\Sigma, \Sigma')$, let $n = |(\Sigma, \Sigma')|$ be the height of the entailment $\Sigma \vdash \Sigma'$. Then for each $m > n$, the function $f_{m+1}(\rho) \stackrel{\text{def}}{=} \lfloor \rho \rfloor_{m+1}$ is a bijection from $S_{m+1}(\Sigma)$ to $S_m^2(\Sigma')$. Here we overload $\lfloor \cdot \rfloor_{m+1}$ over contexts by applying the pair rounding component-wise, *i.e.*, $\lfloor \rho \rfloor_{m+1} = (\rho_l, \rho_r)$

s.t.:

$$\rho_l(v) = \tau_1 \ \wedge \ \rho_r(v) = \tau_2 \quad \text{iff} \quad \lfloor \rho \rfloor_{m+1}(v) = (\tau_1, \tau_2).$$

Furthermore, let ρ, ρ' be solutions of both Σ and Σ' and their height is at most m, *i.e.*:

$$\rho, \rho' \in S_m(\Sigma) \cap S_m(\Sigma').$$

Then by Lemma 5.5.1, the context $\rho'' \stackrel{\text{def}}{=} f_{m+1}^{-1}(\rho, \rho') = \rho \bigtriangledown_{m+1} \rho'$ is a solution of Σ' and its height is at most $m+1$. Thus:

$$f_{m+1}^{-1}((S_m(\Sigma) \cap S(\Sigma'))^2) \ = \ f_{m+1}^{-1}((S_m(\Sigma) \cap S_m(\Sigma'))^2) \ \subseteq \ S(\Sigma').$$

By inclusion property in Lemma 5.5.2, it is sufficient to consider only solutions of height at most n in $S_n(\Sigma)$. □

5.5.2 Correctness proof of Theorem 5.3.1

We now proceed to verify the correctness of Algorithm 4 for GSAT. The heart of GSAT is the specialized solver SSAT for singleton systems. As a result, we need to verify that SSAT is sound. First, we provide several key insights about singleton systems, *e.g.*, how a 'big' solution is decomposed into smaller ones:

Lemma 5.5.3 ([Sol])**.** Let Σ^η be a singleton system and ρ a context of Σ^η s.t. $|\rho| = n > |\Sigma^\eta|$ and $\lfloor \rho \rfloor_n = (\rho_l, \rho_r)$ then:

1. If $\rho \models \Sigma^\eta$ then both $\rho_l \models \Sigma^+$ and $\rho_r \models \Sigma^+$, and either $\rho_l \models \Sigma^\eta$ or $\rho_r \models \Sigma^\eta$.

   Conversely, if one of ρ_l, ρ_r is a solution of Σ^+ and the other is a solution of Σ^η then the context $\rho = \rho_l \bigtriangledown_n \rho_r$ is a solution of Σ^η.

2. Let $\mathsf{Split}(\rho) = (\rho_l, \rho_r)$ and $\mathsf{Split}(\Sigma^\eta) = (\Sigma_l, \Sigma_r)$. If $\rho \models \Sigma^\eta$ then $\rho_l \models \Sigma_l^+$ and $\rho_r \models \Sigma_r^+$. Also, either $\rho_l \models \Sigma_l$ or $\rho_r \models \Sigma_r$.

   Conversely, if either $\rho_l \models \Sigma_l^+ \ \wedge \ \rho_r \models \Sigma_r^\eta$ or $\rho_l \models \Sigma_l^\eta \ \wedge \ \rho_r \models \Sigma_r^+$ then $\rho \models \Sigma^\eta$.

3. If $\mathbf{SAT}(\Sigma^+) = \top$ then:

$$\mathbf{SSAT}(\Sigma^\eta) = \top \quad \text{iff} \quad \mathbf{SSAT}(\Sigma_i) = \top \text{ for some } \Sigma_i \in \mathsf{DECOMPOSE}(\Sigma^\eta).$$

4. If $|\Sigma^\eta| = 0$ then $\mathbf{SSAT}(\Sigma^\eta) = \top$ iff it has a solution of height zero.

◁

Proof intuition. Prop. 1 and 2 can be derived directly from Lemma 5.5.1.

For Prop. 3, it is sufficient to prove for Split instead of $\mathsf{DECOMPOSE}$ as the latter can be generalized by induction. For $\Rightarrow$, let $\rho \models \Sigma^\eta$, $\mathsf{Split}(\rho) = (\rho_l, \rho_r)$, $\mathsf{Split}(\Sigma^\eta) = (\Sigma^\eta_l, \Sigma^\eta_r)$. Then from Prop. 2, either $\rho_l \models \Sigma_l$ or $\rho_r \models \Sigma_r$. For $\Leftarrow$, w.l.o.g. assume $\rho_l \models \Sigma_l$. As $\mathsf{SAT}(\Sigma^+) = \top$, there exists $\rho' \models \Sigma^+$. Let $\mathsf{Split}(\rho') = (\rho'_l, \rho'_r)$ then $\mathsf{Combine}(\rho_l, \rho'_r) \models \Sigma$.

For Prop. 4, notice that if $|\Sigma^\eta| = 0$ then Split is the identity function,*i.e.*, $\mathsf{Split}(\Sigma^\eta) = (\Sigma^\eta, \Sigma^\eta)$. We conduct the proof using our domain reduction technique. First, we classify the solutions of Σ^η into $\{S_i(\Sigma^\eta)\}_{i=0}^{\infty}$ according to their heights. Then, for each $i \in \mathbb{N}$, it follows that $S_i(\Sigma^\eta) \subseteq S_{i+1}(\Sigma^\eta)$, and if $\rho \in S_{i+1}(\Sigma^\eta)$, then either $\rho_l \in S_i(\Sigma^\eta)$ or $\rho_r \in S_i(\Sigma^\eta)$ by Prop. 3 (*i.e.* one of them is a solution of height at most i for the singleton system Σ^η). As a result, we can define a mapping from $S_{i+1}(\Sigma^\eta)$ to $S_i(\Sigma^\eta)$ by choosing the appropriate ρ_l/ρ_r which is in $S_i(\Sigma^\eta)$. By the emptiness property in Lemma 5.5.2, it suffices to search for solutions of height zero. □

Using the above properties, we show that the specialized solver SSAT is sound with respect to appropriate pre-condition:

Lemma 5.5.4 ([Sol])**.** If $\mathbf{SAT}(\Sigma^+) = \top$ then $\mathsf{SSAT}(\Sigma^\eta) = \top$ iff Σ^η is satisfiable. ◁

Proof. By Prop. 3 in 5.5.3, it is equivalent to the satisfiability of one of the singleton sub-systems of height zero in $\mathsf{DECOMPOSE}(\Sigma^\eta)$. By Prop 4 in 5.5.3, such sub-system must have solution of height zero and thus can be transformed into a Boolean formula to be solved by SMT solver. Hence the result follows. □

Our next step is to show that $\mathbf{GSAT}(\Sigma)$ is equivalent to the conjunction of $\mathbf{SSAT}(\Sigma^{\eta_i})$

for each disequation η_i in Σ. Consequently, the main solver GSAT can just simply call the specialized solver SSAT multiple times:

Lemma 5.5.5 ([Sol]). Let Σ be a share system then

$$\mathbf{GSAT}(\Sigma) = \top \text{ iff } \bigwedge_{\eta_i \in \Sigma} \mathbf{SSAT}(\Sigma^{\eta_i}) = \top.$$

◁

Proof. Let $[\eta_1, \ldots, \eta_n]$ be the disequation list in Σ and $A_i = S(\Sigma^{\eta_i})$ the solution space of each singleton system Σ^{η_i} then:

$$S(\Sigma) = \bigcap_{i=1}^{n} A_i.$$

The original statement can be restated as "$S(\Sigma)$ is nonempty iff each A_i is nonempty". Only the $\Leftarrow$ direction is nontrivial as for the other direction, any solution in $S(\Sigma)$ is also a solution in A_i. Let $\rho_i \in A_i$ be a solution of Σ^{η_i}. To construct a solution of Σ from $\{\rho_i\}_{i=1}^n$, we define a sequence of contexts $\{\rho'_i\}_{i=1}^n$ as follows:

$$\rho'_1 \stackrel{\text{def}}{=} \rho_1, \ \rho'_k \stackrel{\text{def}}{=} \rho'_{k-1} \bigtriangledown_{n_k} \rho_k.$$

where $n_k = \max(|\rho'_{k-1}|, |\rho_k|, |\Sigma|) + 1$. Intuitively, the averaging function helps to 'accumulate' the satisfiability of the disequations while preserves the satisfiability of the remaining equations and equalities, *i.e.* ρ'_j satisfies all equations in Σ and disequations $\eta_1, \ldots, \eta_j$ by Lemma 5.5.3:

$$\rho'_j \in \bigcap_{i=1}^{j} A_i.$$

As a result, $\rho'_n \in \bigcap_{i=1}^{n} A_i = S(\Sigma)$ is a solution of Σ. □

Finally, we are ready to justify the correctness of Theorem 5.3.1:

Theorem 5.3.1 ([Sol]). Let Σ be a share system then Σ is satisfiable iff $\mathsf{GSAT}(\Sigma) = \top$.

Proof. As $\mathbf{SAT}(\Sigma^+)$ is a necessary condition for $\mathbf{GSAT}(\Sigma)$, if $\mathbf{SAT}(\Sigma^+) = \bot$ then we also have $\mathbf{GSAT}(\Sigma) = \bot$. This helps explain lines 2-4. We need this fact to activate the condition for Lemma 5.5.4.

By Lemma 5.5.5, the system Σ is satisfiable iff $\mathbf{SSAT}(\Sigma^{\eta_i}) = \top$ for each disequation η_i in Σ. This justifies the conjunction in line 6. Last but not least, Lemma 5.5.4 provides the correctness for the specialized solver SSAT and thus completes the correctness proof for GSAT as well. □

5.5.3 Correctness proof of Theorem 5.4.1

Our decision procedure **GIMP** for entailments makes use of two specialized solvers ZIMP and SIMP for Z-systems and S-systems respectively. Hence the correctness of GIMP also relies on the correctness of ZIMP and SIMP. As a result, we first verify that our two specialized solvers are sound. First, we prove several essential properties about two entailment problems **ZIMP** and **SIMP**:

Lemma 5.5.6 ([Sol])**.** Let Σ_1, Σ_2 be share systems. Then:

1. If $\mathbf{SAT}(\Sigma_1^+) = \top$ and $\Sigma_1^+ \vdash \Sigma_2^+$, then:

$$\Sigma_1^+ \vdash \Sigma_2^\eta \quad \text{iff} \quad \Sigma_a \vdash \Sigma_b \text{ for some } (\Sigma_a, \Sigma_b) \in \mathsf{DECOMPOSE}(\Sigma_1^+, \Sigma_2^\eta).$$

2. If $|(\Sigma_1^+, \Sigma_2^\eta)| = 0$, then:

$$\Sigma_1^+ \vdash \Sigma_2^\eta \quad \text{iff} \quad \Sigma_1^+ \vdash \Sigma_2^\eta \text{ for all contexts of height zero.}$$

3. If $\mathbf{SAT}(\Sigma_1^+) = \top$ and $\Sigma_1^+ \vdash \Sigma_2^+$ and $\Sigma_1^+ \nvdash \Sigma_2^{\eta_2}$, then:

$$\Sigma_1^{\eta_1} \vdash \Sigma_2^{\eta_2} \quad \text{iff} \quad \Sigma_a \vdash \Sigma_b \text{ for each } (\Sigma_a, \Sigma_b) \in \mathsf{DECOMPOSE}(\Sigma_1^{\eta_1}, \Sigma_2^{\eta_2}).$$

4. If $|(\Sigma_1^{\eta_1}, \Sigma_2^{\eta_2})| = 0$ and $\Sigma_1^+ \vdash \Sigma_2^+$, then:

$$\Sigma_1^{\eta_1} \vdash \Sigma_2^{\eta_2} \quad \text{iff} \quad \Sigma_1^{\eta_1} \vdash \Sigma_2^{\eta_2} \text{ for all contexts of height zero.}$$

◁

Proof. The preconditions here are essential for the soundness proof of our specialized solvers. Fortunately, all of them are also necessary conditions for **GIMP** and can be checked by known decision procedures. To make things simple, we will prove over Split instead of DECOMPOSE as the latter is simply a repetitive application of the former and thus its correctness can be proved by induction.

1. For $\Rightarrow$, assume $\Sigma_{1l}^{+} \not\vdash \Sigma_{2l}^{\eta_l}$ and $\Sigma_{1r}^{+} \not\vdash \Sigma_{2r}^{\eta_r}$. Then we can find two contexts ρ_l and ρ_r s.t.:

$$\rho_l \models \Sigma_{1l}^{+} \ \wedge \ \rho_l \not\models \Sigma_{2l}^{\eta_l} \quad \text{and} \quad \rho_r \models \Sigma_{1r}^{+} \ \wedge \ \rho_r \not\models \Sigma_{2r}^{\eta_r} \tag{5.1}$$

   Let $\rho = \mathsf{Combine}(\rho_l, \rho_r)$ then $\rho \models \Sigma_1^{+}$. As $\Sigma_1^{+} \vdash \Sigma_2^{\eta}$, we derive that $\rho \models \Sigma_2^{\eta}$. Thus either $\rho_l \models \Sigma_{2l}^{\eta_l}$ or $\rho_r \models \Sigma_{2r}^{\eta_r}$. This is a contradiction to equation 5.1.

   For $\Leftarrow$, w.l.o.g. assume $\Sigma_{1l}^{+} \vdash \Sigma_{2l}^{\eta_l}$. Let $\rho \models \Sigma_1^{+}$ then $\rho_l \models \Sigma_{1l}^{+}$ and thus $\rho_l \models \Sigma_{2l}^{\eta_l}$ by the entailment assumption. From the premise $\mathbf{SAT}(\Sigma_1^{+}) = \top$, we can find $\rho' \models \Sigma_1^{+}$. Combine with the entailment premise $\Sigma_1^{+} \vdash \Sigma_2^{+}$, this gives us $\rho' \models \Sigma_2^{+}$. As a result, let $\mathsf{Split}(\rho') = (\rho'_l, \rho'_r)$ then $\rho'_r \models \Sigma_{2r}^{+}$. Finally, let $\rho'' \stackrel{\text{def}}{=} \mathsf{Combine}(\rho_l, \rho'_r)$. From the two results $\rho_l \models \Sigma_{1l}^{\eta_l}$ and $\rho'_r \models \Sigma_{2r}^{+}$, we conclude $\rho'' \models \Sigma_2^{\eta}$.

2. We prove using domain reduction. We classify the solution space $S(\Sigma_1^{+})$ into $\{S_i(\Sigma_1^{+})\}_{i=1}^{\infty}$ s.t. $S_i(\Sigma_1^{+}) \subseteq S_{i+1}(\Sigma_1^{+})$ and let Split be the bijection from $S_{i+1}(\Sigma_1^{+})$ to $S_i^2(\Sigma_1^{+})$. Notice that Split is the identity function for systems of height zero, *i.e.*:

$$\mathsf{Split}(\Sigma_1^{+}, \Sigma_2^{\eta}) \ = \ ((\Sigma_1^{+}, \Sigma_2^{\eta}), (\Sigma_1^{+}, \Sigma_2^{\eta})).$$

   Let ρ_l, ρ_r be two contexts of height at most k that both satisfy Σ_1^{+} and Σ_2^{η}, *i.e.*:

$$\rho_l, \rho_r \ \in \ S_k(\Sigma_1^{+}) \cap S_k(\Sigma_2^{\eta}).$$

   Then it follows that the context $\rho \stackrel{\text{def}}{=} \mathsf{Combine}(\rho_l, \rho_r)$ is also a solution of Σ_2^{η}. By the inclusion property of domain reduction, it suffices to consider only height-0 solutions in $S_0(\Sigma_1^{+})$.

3. For $\Rightarrow$, it suffices to prove $\Sigma_{1l}^{\eta_{1l}} \vdash \Sigma_{2l}^{\eta_{2l}}$. Let ρ_l be a context s.t. $\rho_l \models \Sigma_{1l}^{\eta_1}$, we will

prove that $\rho_l \models \Sigma_{2l}^{\eta_{2l}}$. From two premises $\mathbf{SAT}(\Sigma_1^+)$ and $\mathbf{IMP}(\Sigma_1^+, \Sigma_2^+)$, we can find a context ρ' s.t.:

$$\rho' \models \Sigma_1^+ \quad \text{and} \quad \rho' \models \Sigma_2^+.$$

Let $\mathsf{Split}(\rho') = (\rho'_l, \rho'_r)$ and $\rho \stackrel{\text{def}}{=} \mathsf{Split}(\rho_l, \rho'_r)$. Then:

$$\rho'_r \models \Sigma_{1r}^+ \quad \text{and} \quad \rho_r \models \Sigma_{2r}^+ \quad \text{and} \quad \rho \models \Sigma_1^{\eta_1}.$$

As a result, it follows from $\Sigma_1^{\eta_1} \vdash \Sigma_2^{\eta_2}$ that $\rho \models \Sigma_2^{\eta_2}$. Consequently, either $\rho_l \models \Sigma_{2l}^{\eta_{2l}}$ or $\rho'_r \models \Sigma_{2r}^{\eta_{2r}}$. On the other hand, the premise $\Sigma_1^+ \not\vdash \Sigma_2^{\eta_2}$ gives us $\rho' \not\models \Sigma_2^{\eta_2}$ and thus $\rho'_r \not\models \Sigma_{2r}^{\eta_{2r}}$*. Hence it must be the case that $\rho_l \models \Sigma_{2l}^{\eta_{2l}}$.

For $\Leftarrow$, let $\rho \models \Sigma_1^{\eta_1}$ then either $\rho_l \models \Sigma_{1l}^{\eta_{1l}}$ or $\rho_r \models \Sigma_{1r}^{\eta_{1r}}$. W.l.o.g., assume $\rho_l \models \Sigma_{1l}^{\eta_{1l}}$. Then the entailment $\Sigma_{1l}^{\eta_{1l}} \vdash \Sigma_{2l}^{\eta_{2l}}$ implies $\rho_l \models \Sigma_{2l}^{\eta_{2l}}$. From the premise $\Sigma_1^+ \vdash \Sigma_2^+$, we deduce $\rho \models \Sigma_2^+$. As a result, $\rho_r \models \Sigma_{2r}^+$. By combining two results $\rho_l \models \Sigma_{2l}^{\eta_{2l}}$ and $\rho_r \models \Sigma_{2r}^+$, we arrive at $\rho \models \Sigma_2^{\eta_2}$.

4. Only $\Leftarrow$ is nontrivial. It suffices to prove the case when solution has height 1 and the general case will follow by induction. As $|(\Sigma_1^{\eta_1}, \Sigma_2^{\eta_2})| = 0$, the Split returns the same share equation, *i.e.*:

$$\mathsf{Split}(\Sigma_1^{\eta_1}, \Sigma_2^{\eta_2}) \;=\; ((\Sigma_1^{\eta_1}, \Sigma_2^{\eta_2}), (\Sigma_1^{\eta_1}, \Sigma_2^{\eta_2})).$$

Let $\rho \in S_1(\Sigma_1^{\eta_1})$ be a height-1 solution of $\Sigma_1^{\eta_1}$ and $\mathsf{Split}(\rho) = (\rho_l, \rho_r)$ then either $\rho_l \in S_0(\Sigma_1^{\eta_1})$ or $\rho_r \in S_0(\Sigma_1^{\eta_1})$. W.l.o.g., assume $\rho_l \in S_0(\Sigma_1^{\eta_1})$ is a height-0 solution of $\Sigma_1^{\eta_2}$. Then by the premise, we deduce that $\rho_l \models \Sigma_2^{\eta_2}$.

On the other hand, the entailment $\Sigma_1^+ \vdash \Sigma_2^+$ gives us $\rho \models \Sigma_2^+$ and thus $\rho_r \models \Sigma_2^+$. By combining two results $\rho_l \models \Sigma_2^{\eta_2}$ and $\rho_r \models \Sigma_2^+$, we conclude that $\rho \models \Sigma_2^{\eta_2}$.

□

We proceed to verify the correctness of ZIMP and SIMP (Algorithm 8):

*otherwise together with the fact $\rho_l \models \Sigma_{2l}^+$ we can deduce the contradiction $\mathsf{Combine}(\rho_l, \rho'_r) = \rho' \models \Sigma_2^{\eta_2}$

Lemma 5.5.7 ([Sol]). Let Σ_1, Σ_2 be share systems and η, η_1, η_2 disequations. Then:

1. $\mathsf{ZIMP}(\Sigma_1^+, \Sigma_2^\eta) = \top$ iff $\Sigma_1^+ \vdash \Sigma_2^\eta$.
2. $\mathsf{SIMP}(\Sigma_1^{\eta_1}, \Sigma_2^{\eta_2}) = \top$ iff $\Sigma_1^{\eta_1} \vdash \Sigma_2^{\eta_2}$.

◁

Proof. For entailment $\Sigma_1^+ \vdash \Sigma_2^\eta$, we activate the preconditions in Lemma 5.5.6 by first checking two necessary conditions $\mathbf{SAT}(\Sigma_1^+)$ and $\Sigma_1^+ \vdash \Sigma_2^+$. By Prop. 1 in Lemma 5.5.7, it is sufficient to find a height-0 sub-system (Σ_a, Σ_b) in $\mathsf{DECOMPOSE}(\Sigma_1^+, \Sigma_2^\eta)$ s.t. $\Sigma_a \vdash \Sigma_b$. This justifies the disjunction form in line 4. Finally, Prop. 2 says that we only need to consider solutions of height zero and thus the entailment $\Sigma_a \vdash \Sigma_b$ can be transformed into Boolean formula and handled by SMT solver. This completes the soundness proof for ZIMP.

For entailment $\Sigma_1^{\eta_1} \vdash \Sigma_2^{\eta_2}$, we first check two necessary conditions $\mathbf{SAT}(\Sigma_1^+)$ and $\Sigma_1^+ \vdash \Sigma_2^+$ and the sufficient condition $\Sigma_1^+ \vdash \Sigma_2^{\eta_2}$. If no trivial result can be drawn from these conditions then by Prop. 3, it suffices to check $\Sigma_a \vdash \Sigma_b$ for each pair (Σ_a, Σ_b) in $\mathsf{DECOMPOSE}(\Sigma_1^{\eta_1}, \Sigma_2^{\eta_2})$. This justifies the conjunction form in line 11 of SIMP. By Prop. 4, each entailment $\Sigma_a \vdash \Sigma_b$ only needs to hold for solutions of height zero and thus they can be handled by SMT solver. Thus the correctness of SIMP is justified. □

The second ingredient for soundness proof of **GIMP** is the reduction from general entailment **GIMP** to the three special entailments **IMP**, **ZIMP** and **SIMP**:

Lemma 5.5.8 ([Sol]). Let Σ_1, Σ_2 be share systems and l_1^-, l_2^- their disequation lists. Then:

1. If l_2^- is empty and $\mathbf{GSAT}(\Sigma_1) = \top$, then:

$$\Sigma_1 \vdash \Sigma_2 \quad \text{iff} \quad \Sigma_1^+ \vdash \Sigma_2^+.$$

2. If $l_2^- = [\eta_1, \ldots, \eta_n]$ for $n > 0$ and l_1^- is empty, then:

$$\Sigma_1 \vdash \Sigma_2 \quad \text{iff} \quad \Sigma_1 \vdash \Sigma_2^{\eta_i} \text{ for } i = 1 \ldots n.$$

3. If $l_1^- = [\eta_1, \ldots, \eta_n]$ and $l_2^- = [\eta'_1, \ldots, \eta'_m]$ for $n, m > 0$. then:

$$\Sigma_1 \vdash \Sigma_2 \quad \text{iff} \quad \forall \eta'_i \in l_2^-.\ \exists \eta_j \in l_1^-.\ \Sigma_1^{\eta_j} \vdash \Sigma_2^{\eta'_i}.$$

◁

Proof. Similar to the proof of Lemma 5.5.5.

1. Only $\Rightarrow$ is nontrivial. As l_2^- is empty, we have $\Sigma_2^+ = \Sigma_2$. Let $\rho \models \Sigma_1$, we will show that $\rho \models \Sigma_2$. As $\mathbf{GSAT}(\Sigma_1) = \top$, we can find a context ρ' s.t. $\rho' \models \Sigma_1$. Let $n \stackrel{\text{def}}{=} \max(|\rho|, |\rho'|, |\Sigma_1|, |\Sigma_2|) + 1$ (*i.e.* we need to make sure n is sufficiently big to apply the average function), we combine two contexts ρ, ρ' into a single context ρ_n using the averaging function $\triangledown_n$, *i.e.*:

$$\rho_n \ = \ \rho \ \triangledown_n \ \rho'.$$

   Then ρ_n is a solution of Σ_1. From the premise $\Sigma_1 \vdash \Sigma_2$, we derive that $\rho_n \models \Sigma_2 = \Sigma_2^+$. As $\lfloor \rho \rfloor_n = (\rho, \rho')$ and $n > |\Sigma_2|$, both ρ and ρ' are solutions of Σ_2 by Corollary 5.5.1. Thus the result follows.

2. As $l_2^- = [\eta_1, \cdots, \eta_n]$, we have $S(\Sigma_2) \ = \ \bigcap_{i=1}^n S(\Sigma_2^{\eta_i})$. The entailment $\Sigma_1 \vdash \Sigma_2$ is equivalent to the set inclusion $S(\Sigma_1) \subseteq S(\Sigma_2)$. Similarly, $\Sigma_1 \vdash \Sigma_2^{\eta_i}$ is equivalent to $S(\Sigma_1) \subseteq S(\Sigma_2^{\eta_i})$ for $i = 1 \ldots n$. On the other hand, we have the following fact from basic set theory:

$$A \subseteq \bigcap_{i=1}^{n} B_i \quad \text{iff} \quad A \subseteq B_i \text{ for } i = 1 \ldots n.$$

   Hence the result trivially follows from these observations.

3. Let $A_j \stackrel{\text{def}}{=} S(\Sigma_1^{\eta_j})$ for $j = 1 \ldots n$ and $B_i \stackrel{\text{def}}{=} S(\Sigma_2^{\eta'_i})$ for $i = 1 \ldots m$. The entailment $\Sigma_1 \vdash \Sigma_2$ is equivalent to:

$$\bigcap_{j=1}^{n} A_j \ \subseteq \ \bigcap_{i=1}^{m} B_i.$$

By simple set theory argument, the above set inclusion is equivalent to:

$$\bigcap_{j=1}^{n} A_j \ \subseteq \ B_i \text{ for } i = 1 \ldots m.$$

Fix a value for i. If there exists some A_j s.t. $A_j \subseteq B_i$ then we are done. This also means that $\Sigma_1^{\eta_j} \vdash \Sigma_2^{\eta'_i}$. Otherwise, assume $A_j \not\subseteq B_i$ for all $j = 1 \ldots n$, it suffices to show that $\bigcap_{j=1}^{n} A_j \ \not\subseteq \ B_i$. Let $\rho_j \in A_j \backslash B_i$, *i.e.*, ρ_j is a solution of $A_j = S(\Sigma_1^{\eta_j})$ but it is not a solution of $B_i = S(\Sigma_2^{\eta'_i})$. We define the sequence $\{\rho'_k\}$ as follows:

$$\rho'_1 \ \stackrel{\text{def}}{=} \ \rho_1, \ \rho'_{k+1} \ \stackrel{\text{def}}{=} \ \rho'_k \ \triangledown_{h_{k+1}} \ \rho_{k+1}.$$

where $h_{k+1} = \max(|\rho'_k|, |\rho_{k+1}|, |\Sigma_1|, |\Sigma_2|) + 1$. By Lemma 5.5.3, we deduce that ρ'_n is a solution of A_j for $j = 1 \ldots n$ and it is not a solution of B_i. It follows that $\rho'_n \in \bigcap_{j=1}^{n} A_j \backslash B_i$. Hence $\bigcap_{j=1}^{n} A_j \ \not\subseteq \ B_i$ is a contradiction. Thus the result follows.

□

We are now ready to justify the soundness of GIMP (Algorithm 7) in Theorem 5.4.1:

Theorem 5.4.1 ([Sol]). Let Σ_1, Σ_2 be share systems then $\Sigma_1 \vdash \Sigma_2$ iff $\mathsf{GIMP}(\Sigma_1, \Sigma_2) = \top$.

Proof. We first activate several necessary conditions $\mathsf{GSAT}(\Sigma_1)$ (line 2) and $\Sigma_1^+ \vdash \Sigma_2^+$ (line 3). Then the subsequent flow of GIMP follows from Lemma 5.5.5 which depends on the length of two disequation lists l_1^- and l_2^-. Here our main solver calls the two specialized solvers ZIMP and SIMP (line 7 and 10) whose soundness is verified in Lemma 5.5.7. As a result, the soundness of GIMP is justified. □

5.6 Performance-enhancing components

The architecture of our tool was given in §5.2.1 (Figure 5.1). The key DECOMPOSE, TRANSFORM and INTERPRET components were discussed in §5.2.1, §5.3.1, and §5.4.1. Here we give details on the PARTITION, BOUND, and SIMPLIFY modules. Their principal goal is to shrink the search space and uncover contradictions, although they each do so

in a very different way. Although in practice they can substantially improve performance, none of these components is a complete solver. The key ideas in these components were developed previously [HG12, LGH12], although not all together. We have made a number of incremental enhancements, but our major contribution for these is components is the development of high-performing general-purpose certified implementations.

PARTITION. The goal of this module is to separate a constraint system into *independent subsystems*. Two systems are independent of each other if they do not share any common variable (with existential variables bound locally).

The partition function is implemented *generically*: in other words it does not assume very much about the underlying domain. To build the module, we must specify types of *variables* V, *equations* E, and contexts C. We also provide a parameterized function $\sigma : E \to L(V)$ that extracts a list of variables from an equation, an overriding function written $\rho'[\rho \Leftarrow l]$, and an evaluation relation written $c \models e$. The soundness proof requires two axioms that relate these inputs as follows:

$$\frac{\rho \models e \qquad \sigma(e) \cap l = \emptyset}{\rho[\rho' \Leftarrow l] \models e}\ \textit{disjointness} \qquad\qquad \frac{\rho \models e \qquad \sigma(e) \subset l}{\rho'[\rho \Leftarrow l] \models e}\ \textit{inclusion}$$

Disjointness and inclusion jointly specify that satisfaction of an equation only depends on the variables it contains: overriding variables not in the equation does not matter; and from any context, if we overload all of the variables that are in an equation then we can ignore the original context.

It is simple to use PARTITION for **GSAT**, but to handle **GIMP** is harder. We can "tag" equations and variables as coming from the antecedent or consequent before partitioning and then use these tags to separate the resulting partitioned systems into antecedents and consequents afterwards.

The implementation of PARTITION is nontrivial in purely functional languages like Coq. One reason is that we need a purely functional union-find data structure, which we obtain via the impure-to-pure transformation of Pippenger [Pip96] applied to the canonical imperative algorithm [CLRS09]. In other words, we substitute red-black trees for memory (mapping

"addresses" to "cell contents") and pay a logarithmic access penalty, yielding an $O(n \cdot \mathsf{log}(n) \cdot \alpha(n))$ algorithm.

The termination of "find" turns out to be subtle. Parent pointers are represented as cells that "point to" other cells; however, those parent cells can be anywhere in the red-black tree (*e.g.* item 5 can be the parent of item 10, or the other way around.) Accordingly, an important invariant of the structure is that "nonlocal links" form acyclic chains, which is the key termination argument.

Given union-find, the algorithm is straightforward: each variable is put into a singleton set, and then while processing each equation we union the corresponding sets. Lastly, we extract the sets and filter the equations into components.

BOUND. The bound module uses order theory to prune the space. Each variable v is given an initial bound $\circ \subseteq v \subseteq \bullet$. The bounder then tries to narrow these bounds by forward and backward propagation. For example, if $\tau_1 \subseteq v_1 \subseteq \bullet$, $\tau_2 \subseteq v_2 \subseteq \bullet$, and $\circ \subseteq v_3 \subseteq \bullet$, then if $v_1 \oplus v_2 = v_3$ is an clause we can conclude that v_3's lower bound can be increased from $\circ$ to $\tau_1 \sqcup \tau_2$ (where $\sqcup$ computes the union in an underlying lattice on trees). In some cases, the bounds for a variable can be narrowed all the way to a point, in which case we can substitute the variable away. In other cases we can find a contradiction (when the upper bound goes below the lower bound), allowing us to terminate the procedure.

The bound algorithm is an updated version of the incomplete solver developed by Hobor *et al.* [HG12]. Although our main contribution here is the certified implementation, we managed to tighten the bounds in certain cases.

SIMPLIFY. The simplify module is a combination of a substitution engine and several effective heuristics for reducing the overall difficulty via calculation. For example, from $v \oplus \tau_1 = \tau_2$, where τ_i are constants, we can compute an exact value for v using an inverse of $\oplus$: $v = \tau_2 \ominus \tau_1$. SIMPLIFY also hunts for contradictions: for example, from $v \oplus v = \bullet$ we can reach a contradiction due to the "disjointness" axiom from Figure 2.3. The core idea of simplifier was contained in the work of Le *et al.* [LGH12], so our main contribution here is

our certified implementation.

5.7 Experimental evaluation

Our procedures are implemented and certified in Coq. Users who wish to use our code outside of Coq can use Coq's extraction feature to generate code in OCaml, although at present a small bug in Coq 8.4pl5's extraction mechanism requires a small human edit to the generated code.

We benchmarked our code in three ways using an Intel i7 with 8GB RAM. First, we used a suite of 102 standalone test cases developed for our uncertified solvers in Chapter 4 (53 **SAT** and 49 **IMP**) and all metatheoretic properties described in Figure 2.3. These tests cover a variety of tricky cases such as large number of variables, deep tree constants, etc. Even running as interpreted Gallina code within Coq, the time is extremely encouraging at **17 seconds** to check all 111 tests. After we port to Coq 8.5 we can use the native_compute tactic to increase performance.

Second, we compiled the extracted OCaml code with ocamlopt. The total running time to test all 111 previous tests is **0.02 seconds**, despite our naïve SMT solver; our previous tool took 1.4 seconds. Since our SMT solver is a separate module, it can be replaced with a more robust external solver such as Z3 [dMB08] if performance bottlenecks in that spot in the future.

Finally, we incorporated our solver into the HIP/SLEEK verification toolset, which was previously using the uncertified solvers SAT and IMP in Chapter 4. We did so by writing a short (approximately 150 line) "shim" that translated the format used by the previous tool into the format expected by the new tool.

We then benchmarked our tool against a suite of 23 benchmark programs as shown in Table 5.1. 15 of those programs were developed by Gherghina [Ghe12] and utilize a concurrent separation logic for pthreads-style barriers that exercise share provers extensively. Another 7 tests were developed for the HipCAP project [CLQ17], which extended HIP/SLEEK to reason in a Concurrent Abstract Predicate [DYDG+10] style. Finally, we wrote a simple fork/join

File	LOC	# calls	# wrong	Uncertified tool	Certified tool
MISD_ex1_th1.ss	36	294	48	2.21	2.37
MISD_ex1_th2.ss	36	495	67	4.36	4.48
MISD_ex1_th3.ss	36	726	94	6.95	6.58
MISD_ex1_th4.ss	36	1,003	123	9.09	8.36
MISD_ex1_th5.ss	36	1,320	134	15.74	12.38
MISD_ex2_th1.ss	47	837	107	16.77	18.97
MISD_ex2_th2.ss	52	1,044	157	29.34	26.02
MISD_ex2_th3.ss	87	1,841	260	69.09	64.21
MISD_ex2_th4.ss	105	3,023	374	194.17	194.64
PIPE_ex1_th2.ss	35	283	7	2.49	2.78
PIPE_ex1_th3.ss	44	467	12	4.92	4.65
PIPE_ex1_th4.ss	56	678	15	7.00	7.53
PIPE_ex1_th5.ss	66	931	18	9.67	9.37
SIMD_ex1_v2_th1.ss	74	1,167	281	18.46	17.64
SIMD_ex1_v2_th2.ss	95	2,029	392	63.83	53.50
cdl-ex1a-fm.ss	49	7	0	0.10	0.08
cdl-ex2-fm.ss	50	9	0	0.12	0.09
cdl-ex3-fm.ss	51	10	0	0.11	0.12
cdl-ex4-race.ss	50	5	0	0.09	0.09
cdl-ex4a-race.ss	50	9	0	0.10	0.08
cdl-ex5-deadlock.ss	42	5	0	0.10	0.10
cdl-ex5a-deadlock.ss	42	9	0	0.08	0.08
ex-fork-join.ss	25	47	22	0.19	0.16
total		**10,252**	**534**	**455.01**	**434.30**

Table 5.1: Evaluation of our procedures using HIP/SLEEK

program for our initial testing.

The results are rather interesting! The left column gives the input file name to HIP/SLEEK and the second the number of lines in that file. The third column is the total number of calls into the solver (both **GSAT** and **GIMP**). **The fourth column is the number of times the previous solvers answered the query incorrectly.** The fifth column gives the time (in seconds) spent by our uncertified solvers in Chapter 4 (SAT and IMP) and the sixth column gives the time spent by our new certified solver. HIP/SLEEK was benchmarked on a more powerful machine with 16 cores and 64GB RAM.

The uncertified solver got approximately 5.2% of the queries wrong! In our subsequent investigation, we discovered a number of bugs in the original solver: code rot (due to a change in the correct mechanism to call the SMT backend), improper error handling and signaling, general coding errors, and the incorrect treatment of nonzero variables. We also discovered bugs in HIP/SLEEK itself, which did not always use the result of the solver in the correct way; this is why the regression tests were passing even though the solver was reporting the incorrect answer. Our discovery of bugs on this scale, despite the large benchmarks developed for our uncertified solvers and by Gherghina [Ghe12], illustrates the value of developing certified decision procedures.

Our timing results are reasonable: despite our naïve SMT solver backend and the difficulties in writing the algorithms in a purely functional style, our tool is approximately 4.6% faster than the time spent in the uncertified solvers.

5.8 Development file list

Table 5.2 contains a file-by-file summary of our development. Overall we have approximately 38.6k lines of code. These include roughly 5k lines of modifications to the Mechanized Semantic Library [ADH09], which contained the original mechanized definitions of tree shares by Dockins *et al.* [DHA09]. We have significantly expanded the theory to account *e.g.* for the operations of rounding and averaging explained and formalized in §5.5. We did some of these MSL modifications to justify our previous decision procedure in Chapter 4. The rest

folder	file	LOC
`/msl/`(2 modified files)		≈ 5,000
	`boolean_alg.v`	≈ 500
	`tree_shares.v`	≈ 4,500
`/rbt/` (3 files)		5,259
`/uf/` (4 files)		3,881
	`base.v`	268
	`UF_interface.v`	935
	`UF_base.v`	2,258
	`UF_implementation.v`	420
`/part/` (4 files)		2,729
	`base.v`	268
	`partition_base.v`	434
	`partition_ibase.v`	1,094
	`partition_implementation.v`	771
`/` (20 files)		21,740
	`base.v`	268
	`share_dec_base.v`	524
	`base_properties`	786
	`share_equation_system.v`	1,133
	`share_dec_interface.v`	536
	`bool_to_formula.v`	643
	`fbool_solver.v`	1,497
	`bool_solver.v`	417
	`share_correctness_base.v`	1,594
	`share_correctness.v`	447
	`share_decompose_base.v`	2,265
	`share_decompose.v`	670
	`share_to_bool.v`	637
	`borders.v`	464
	`bound_map.v`	228
	`share_bounder.v`	2,510
	`partition_modules.v`	3,908
	`share_simplifier.v`	2,337
	`share_solver.v`	792
	`share_solver_with_partition.v`	84
Total (31 files)		≈ 38,609

Table 5.2: Our development

of MSL adds about another 30k lines of code, but we do not include these lines in our table as we only use a small portion. The directory `/rbt/` contains a general and well-performing red-black tree implementation [App11a].

All of the other files are new for the present work. The `/uf/` directory uses the red-black trees to build our purely functional union-find implementation. The `/part/` directory uses union find to build our generic partition module.

The bulk of the files are in the main directory `/`. We define share equation system in `share_equation_system.v` and state the formal correctness property of our procedures in `share_dec_interface.v`. The INTERPRET component is in `bool_to_formula.v` and `fbool_solver.v` contains the SMT solver; `bool_solver.v` chains these together.

The key theoretical proofs for domain reduction are in the files associated with §5.5. The code for the DECOMPOSE is in `share_decompose_base.v`, which defines the function ϕ, and `share_decompose.v`, which defines the Split function. The code of TRANSFORM is in `share_to_bool.v`; its correctness proof uses all §5.5 files. Lemmas 5.5.5 and 5.5.8 are proven in `share_correctness_base.v` and `share_correctness.v`.

The BOUND module is in `borders.v`, `bound_map.v`, and `share_bounder.v`. The PARTITION module is in `partition_modules.v`, which uses our generic partitioner. The SIMPLIFY module is in `share_simplifier.v`.

The main procedures (GSAT and GIMP) and correctness proofs are in `share_solver.v`. An optimized version in which we use the module PARTITION to divide the system into independent components are implemented in `share_solver_with_partition.v`.

5.9 Conclusion

We have used tree shares to model permissions for integration into program logics. We proposed two decision procedures for tree shares and proved their correctness in Coq. The two algorithms perform well in practice and have been integrated into a sizable verification toolset. In subsequent chapters, we will examine the theory further to support general logical

formulae (including arbitrary quantifier use) and perhaps monadic 2nd order logic. Also, we will investigate the decidability of bowtie $\bowtie$, which is a kind of multiplicative operation between shares.

Chapter 6

Decidability and complexity of tree shares

> "*Life is about accepting the challenges along the way, choosing to keep moving forward, and savoring the journey.*"
>
> Roy T. Bennett, *The Light in the Heart.*

In previous chapters, we discussed about decision procedures over the tree share structure $\langle \mathbb{T}, \oplus \rangle$. Our decision procedures are capable of solving a subclass of first-order tree formulas, *i.e.*, existential and universal formulas. To an extent, these formulas are sufficiently expressive for practical purposes as required by verification tools that support tree shares such as HIP/SLEEK [NDQC07] and VST [App11b]. As a result, it is essential to build decision procedures to handle such constraints. As $\oplus$ is constructed from $\sqcup$ and $\sqcap$, we are interested in developing decision procedure over the general structure $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot} \rangle$. We discovered that there is a strong connection between the above structure and Boolean Algebra: this structure is a model for Countable Atomless Algebra. We then are able to derive the complexity and decidability for the structure $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot} \rangle$ from these established connection.

The structure of this chapter is divided into the following sections*:

1. In §6.1, we mention all the necessary backgrounds and key results about Boolean Algebra.
2. In §6.2, we prove that the structure $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot} \rangle$ is a model for Countable Atomless Boolean Algebra and derive a lower bound $\mathsf{STA}(*, 2^{n^{O(1)}}, n)$ for its first-order theory,

*The materials in this chapter are taken from two papers "Decidability and Complexity of Tree Shares Formulas" [LHL16] and "Complexity Analysis of Tree Share Operations" [LHL17], joint work with my supervisor Aquinas Hobor and my mentor Anthony W. Lin.

i.e., the class of alternating Turing machines that use $2^{n^{O(1)}}$ times and n alternations.

3. In §6.3, we derive the upper bound for first-order theory of $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot} \rangle$ which is $\mathsf{STA}(*, 2^{n^{O(1)}}, n)$. Hence, the first-order theory of $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot} \rangle$ is $\mathsf{STA}(*, 2^{n^{O(1)}}, n)$-complete.

4. In §6.4, we draw our conclusion.

6.1 Preliminaries

6.1.1 Language and structure

A *signature* is a triple $\sigma = (\mathcal{F}, \mathcal{P}, \mathsf{arity})$ of function symbols $\mathcal{F} = \{f_1, \ldots, f_n\}$, predicate symbols $\mathcal{P} = \{P_1, \ldots, P_m\}$ and the arity function $\mathsf{arity} : \mathcal{F} \cup \mathcal{P} \mapsto \mathbb{N}$ that specifies the number of arguments for functions and predicates in $\mathcal{F} \cup \mathcal{P}$. A variable instance v is *bound* if it is within the scope of some quantifier $\forall v$ or $\exists v$, otherwise v is *free*. A σ-formula Φ is a *sentence* if it does not contain any free variables. A *σ-theory* is simply a set of *first-order σ-sentences*. A σ-theory T is *complete* if for each σ-sentence Φ, either Φ or $\neg\Phi$ is in T. We say T *decidable* if membership testing of σ-sentences in T is decidable, *i.e.*, there exists a halted Turing machine that can decide whether a given formula is in T.

We recall the formula hierarchy as follows. Let Σ_1 be the set of existential formulas of the form $\exists v_1 \ldots \exists v_n.\ \Phi$ and Π_1 the set of universal formulas $\forall v_1 \ldots \forall v_n.\ \Phi$ in which Φ is quantifier-free. Generally, Σ_{i+1} is the set of formulas $\exists v_1 \ldots \exists v_n.\Phi$ for $\Phi \in \Pi_i$ and Π_{i+1} is the set of formulas $\forall v_1 \ldots \forall v_n.\Phi$ for $\Phi \in \Sigma_i$.

A *σ-structure* is an interpretation of the symbols in σ. Formally, a σ-structure is the triple $\mathcal{A} = \langle \mathcal{U}, \mathcal{F}^{\mathcal{A}}, \mathcal{P}^{\mathcal{A}} \rangle$ in which $\mathcal{U}$ is the *universe of discourse* while $\mathcal{F}^{\mathcal{A}}$ and $P^{\mathcal{A}} \subseteq \mathcal{U}^k$ contain functions and predicates whose symbols are from the signature σ. The structure $\mathcal{A}$ satisfies a σ-formula Φ, denoted by $\mathcal{A} \models \Phi$, if Φ is true under the interpretation of $\mathcal{A}$ (please refer to §2.1.1 for a formal definition). We let $\mathsf{Th}(\mathcal{A})$ denote the *first-order theory of* $\mathcal{A}$, *i.e.*, the set of σ-sentences that are satisfied by $\mathcal{A}$. Two σ-structures $\mathcal{A}_1$ and $\mathcal{A}_2$ are *elementarily equivalent* if they satisfy the same set of first-order σ-sentences, *i.e.*, $\mathsf{Th}(\mathcal{A}_1) = \mathsf{Th}(\mathcal{A}_2)$. Furthermore, let $A = \{\Psi_1, \Psi_2, \ldots\}$ be a set of σ-sentences called *axioms* then a structure $\mathcal{A}$

is a *model* of A if $\mathcal{A}$ satisfies all axioms in A. Also, the *first-order theory of* A, denoted by $\mathsf{Th}(A)$, is the set of all sentences that are satisfied by all models of A.

If the signature σ is clear from the context or not important, we will usually omit the prefix σ in the corresponding names. For convenience, we will usually abuse a function (predicate) with its symbol, *i.e.*, f represents both the function symbol in σ and the function $f^{\mathcal{A}}$ in $\mathcal{A}$. As a result, we will usually mention structures without introducing their signatures as such signatures can be derived from the structures themselves. For the purpose of this thesis, **the universe of the structure is a part of the signature**, *i.e.*, it is also the set of constant symbols in the signature. In addition, we will introduce a structure as $\mathcal{A} = \langle \mathcal{U}, \mathcal{X}_1, \ldots, \mathcal{X}_n \rangle$ in which $\mathcal{A}$ is the name of the structure, $\mathcal{U}$ is its universe and $\mathcal{X}_i$ is either a function or predicate whose arity is implicitly known. Also, we reuse some notations in different domains as long as there is no confusion.

6.1.2 Computational complexity

First, we recall the definition of Turing machine, an abstract vehicle to measure the decidability and complexity (*e.g.* [Koz06, Pap03]). Simply speaking, a Turing machine M is an infinite tape consists of writeable cells and its main mechanism is to read an input and produce an output. Initially, the input is written in the middle of the tape and is surrounded by infinitely many blank cells at both ends. There is a head pointer to indicate the current read cell and it first points to the first cell of the input. Each time the machine starts to read the value of the cell then chooses either to move the head to the left or right. After a while, the machine may declare "finished" and whatever written on the tape is considered the output. Formally:

Definition 6.1.1. A deterministic Turing machine is the 7-tuple $M = \langle Q, \Sigma, \Gamma, \delta, q_0, b, F \rangle$ in which:

1. Q is the set of states.
2. Σ is the set of input symbols, *i.e.* the alphabet describes the input.
3. Γ is the set of of tape symbols, *i.e.*, the alphabet describes the written output (together

with Σ).

4. $\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$ is the transition function that helps mechanize the behavior of the head. In short, δ takes in a pair of the current state and symbol pointed to the head and then returns the next state together with the written symbol and the action L or R (*i.e.* move the head to left or right).

5. $q_0 \in Q$ is the initial state.

6. b is the special 'blank symbol' that helps separate the written and unwritten part.

7. $F \subseteq Q$ is the set of accepting states.

Furthermore, if we change the transition function δ to relation $(Q \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$ then we obtain the definition of non-deterministic Turing machine. The key difference between nondeterministic and deterministic Turing machines is that the former allows multiple choices at each step of computation while the latter has at most one choice. Nondeterministic Turing machines are as powerful as the deterministic version in term of computation as one can simulate the other and vice-versa. ◁

Remark. From this basic definition, we can construct other generalized versions of Turing machine which are as powerful as the original one, *e.g.*:

1. Idle-head Turing machine: the head has another *idle* state that allows it remain stationary.

2. Multi-tape Turing machine: the machine has multiple tapes for different purposes. Each tape has its own head which moves individually at each step. Also, there is a special tape that contains the input and a special tape to write the output.

3. One-end Turing machine: We consider a cell to be the boundary. Initially, the input is placed on the right of the boundary cell and the head is not allowed to trespass to the left of the boundary cell.

Computation. A Turing *configuration* helps capture the current image of the Turing tape that indicates the current state, the location of the head together with the tape content. Formally, it is defined as a triple (T_1, q, T_2) in which $T_1 \in (\Sigma \cup \Gamma)^*$ is the prefix string of the

tape before head, T_2 is the suffix string of the tape since the head inclusively, and q is the current state. A Turing *step* updates the tape's content under the effect of transition function. Particularly, let c_1, c_2 be configurations. Then a step from c_1 to c_2, denoted $c_1 \overset{M}{\longmapsto} c_2$, is defined as:

$$\frac{c_1 = (W'a', q, aW) \quad c_2 = (W', r, a'a''W) \quad \delta(q,a) = (r, a'', L)}{c_1 \overset{M}{\longmapsto} c_2}\ \mathsf{left}$$

$$\frac{c_1 = (W'a', q, aW) \quad c_2 = (W'a'a'', r, W) \quad \delta(q,a) = (r, a'', R)}{c_1 \overset{M}{\longmapsto} c_2}\ \mathsf{right}$$

We say configuration c_a leads to c_b (or c_b is reachable from c_a), denoted as $c_a \overset{M^*}{\longmapsto} c_b$, if there exists a sequence of configurations $\{c_k\}_0^n$ such that $c_a \overset{M}{\longmapsto} c_0 \overset{M}{\longmapsto} \ldots \overset{M}{\longmapsto} c_n \overset{M}{\longmapsto} c_b$. An initial (accepting) configuration is a configuration with initial (accepting) state. Furthermore, an accepting Turing *computation* is a sequence of steps $c_0 \overset{M}{\longmapsto} c_1 \overset{M}{\longmapsto} \ldots \overset{M}{\longmapsto} c_n$ in which c_0 and c_n are the initial and accepting configuration respectively. We say M accepts input w iff there exists an accepting computation $c_0 \overset{M^*}{\longmapsto} c_n$ such that $c_0 = (\epsilon, q_0, w)$. By definition, nondeterministic Turing machines only need one accepting computation path to accept an input.

Complexity class and decidability. A complexity class C consists of problems $A = (P, Q)$ in which P contains the problem instances and Q represents the query over P. For example, the Boolean satisfiability problem is the pair (P, Q) in which P is the set of quantifier-free Boolean formulas and Q is the query whether an formula $\Phi \in P$ is satisfiable. A problem A is decidable if there exists a halted Turing machine that can answer the query of A and halts on all inputs. The complexity of problem A (or class C) is then measured and represented by the (time or space, desirably minimal) complexity of the halted (deterministic or nondeterministic) Turing machines that answer/decide the query Q. A theory T with signature σ is decidable if there exists a halted Turing machine that can check whether a σ-formula is in T. Also, complexity class C_1 is subsumed by C_2, denoted by $\mathsf{C}_1 \subseteq \mathsf{C}_2$, if all problems in C_1 can be solved by Turing machines that decide problems in C_2.

Example 6.1.1. The class NP contains problems decided by nondeterministic polynomial time Turing machines. On the other hand, PSPACE are problems decided by deterministic

polynomial space Turing machines. It is also known that $\mathsf{NP} \subseteq \mathsf{PSPACE}$ but whether the inclusion is strict remains unknown. ◁

Reduction. Let R, C be complexity classes, a problem P is $\leq_{\mathsf{R}}$-hard for C if each problem in C can be reduced to P by a many-one reduction in R. Similarly, P is $\leq_{\mathsf{R}}$-complete for C if it is in C and $\leq_{\mathsf{R}}$-hard for C. In addition, we use $\leq_{\mathsf{R}\text{-lin}}$ to assert *linear reduction* that is in R and only changes the problem's size by a constant factor. In particular, $\leq_{\text{log-lin}}$ is linear log-space reduction.

For example, the most two common reductions are $\leq_{\mathrm{p}}$ (deterministic polynomial time) and $\leq_{\log}$ (deterministic logarithmic space) reductions. By a well-known complexity result, it is known that log-space reduction is also polynomial-time reduction but it is not known whether the inclusion is strict.

If the reduction is in polynomial time, we will usually omit the type of reduction and simply say P is C-hard or C-complete. Notice that C-hard problems are unnecessary in complexity class C; otherwise, they are C-complete by definition.

Exponential and elementary complexity. Let the *exponent function* $\mathsf{exp} : \mathbb{N}^2 \mapsto \mathbb{N}$ be:

$$\mathsf{exp}(n, 0) = n \qquad \text{and} \qquad \mathsf{exp}(n, k+1) = 2^{\mathsf{exp}(n,k)}.$$

The complexity class $k\mathsf{EXP}$ contains problems which can be decided by a halting deterministic Turing machines of time complexity $\mathsf{exp}(cn, k)$ for input of length n and some constant c. Similarly, we use $k\mathsf{NEXP}$ and $k\mathsf{EXSPACE}$ for exponential time complexity of nondeterministic halted Turing machines and space complexity of deterministic halted Turing machines respectively. A problem is *elementary* if it is in $k\mathsf{EXP}$ for some k, otherwise it is called *non-elementary*.

Alternating Turing machines. Alternating Turing machines (ATM) are generalized from nondeterministic Turing machines. Informally speaking, an ATM has two disjoint sets of states, *i.e.*, existential states and universal states. If the machine is currently at one of the existential states, it only needs one computation path to the accepting state in order to accept the input. Likewise, a machine in universal states will need all its computation paths

lead to accepting states. Formally, an ATM is a tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, b, h\rangle$ such that:

1. Q is the set of states.
2. Σ is the input alphabet.
3. Γ is the finite tape alphabet.
4. $\delta : Q \times \Gamma \mapsto \mathcal{P}(Q \times \Gamma \times \{L, R\})$ is the transition function in which L, R indicate the movement of the head to the left and right respectively.
5. q_0 is the initial state.
6. b is the special blank symbol for tape.
7. $h : Q \mapsto \{\forall, \exists, \text{acc}, \text{rej}\}$ is the state classifier for universal, existential, acceptance and rejection states.

Let c be a configuration and q be its state. We override the classifier h over c so that it is either *accepting* ($h(c) = \text{acc}$) or *rejecting* ($h(c) = \text{rej}$) by the following rules:

1. If $h(q) = \text{acc}$ then $h(c) = \text{acc}$; or if $h(q) = \text{rej}$ then $h(c) = \text{rej}$.
2. If $h(q) = \forall$ (*i.e.* q is universal state) then c is accepting if all configurations reachable from c in one step are accepting; and rejecting if some configuration reachable in one step is rejecting, *i.e.*:

$$h(c_\forall) = \text{acc} \stackrel{\text{def}}{=} \forall c'.\, c_\forall \stackrel{M}{\longmapsto} c' \Rightarrow h(c') = \text{acc} \quad h(c_\forall) = \text{rej} \stackrel{\text{def}}{=} \exists c'.\, c_\forall \stackrel{M}{\longmapsto} c' \wedge h(c') = \text{rej}.$$

3. If $h(q) = \exists$ (*i.e.* q is existential state) then c is accepting if there is some configuration reachable from c in one step that is accepting; and rejecting if all configurations reachable in one step are rejecting, *i.e.*:

$$h(c_\exists) = \text{acc} \stackrel{\text{def}}{=} \exists c'.\, c_\exists \stackrel{M}{\longmapsto} c' \wedge h(c') = \text{acc} \quad h(c_\exists) = \text{rej} \stackrel{\text{def}}{=} \forall c'.\, c_\exists \stackrel{M}{\longmapsto} c' \Rightarrow h(c') = \text{rej}.$$

The complexity of an ATM is measured via three parameters: time, space and number of alternations. While time and space are standard, number of alternations are the number

Identity :	$a \cup 0 = a$	$a \cap 1 = a$	(6.1)
Null :	$a \cup 1 = 1$	$a \cap 0 = 0$	(6.2)
Idempotency :	$a \cup a = a$	$a \cap a = a$	(6.3)
Involution :	$\bar{\bar{a}} = a$		(6.4)
Complementary :	$a \cup \bar{a} = 1$	$a \cap \bar{a} = 0$	(6.5)
Commutativity :	$a \cup b = b \cup a$	$a \cap b = b \cap a$	(6.6)
Associativity :	$(a \cup b) \cup c = a \cup (b \cup c)$	$(a \cap b) \cap c = a \cap (b \cap c)$	(6.7)
Distributivity :	$(a \cap b) \cup c = (a \cup c) \cap (b \cup c)$	$(a \cup b) \cap c = (a \cap c) \cup (b \cap c)$	(6.8)

Figure 6.1: Axioms of BA (variables a, b, c are universal)

of times the machine switch from $\forall$-state to $\exists$-state or vice-versa. As a result, we let $\mathsf{STA}(p(n), t(n), a(n))$ denote the class of problems decided by an ATM that uses at most $p(n)$ space, $t(n)$ time and $a(n)$ alternations for input of length n. If one of the three bound is not specified, we represent it with $*$. Last but not least, it is well-known that complexity of ATM is between nondeterministic time and deterministic space:

Proposition 6.1.1 ([CKS81])**.** Let $S(n)$ be a function such that $S(n) \geq n$ for $n \in \mathbb{N}$ then:

$$\mathsf{NTIME}(S(n)) \subseteq \mathsf{STA}(*, S(n), *) \subseteq \mathsf{SPACE}(S(n)).$$

where NTIME and SPACE denote the complexity classes for nondeterministic time and deterministic space respectively. ◁

6.1.3 Boolean Algebra

The language of Boolean Algebra (BA), *e.g.* [PH09, Whi61], consists of symbols $\sigma_{\mathsf{BA}} = (\cap, \cup, \bar{\cdot}, 0, 1)$ in which $\cup$ is AND (conjunction), $\cap$ is OR (disjunction), $\bar{\cdot}$ is NOT (negation), 0 is the OR identity and 1 is the AND identity. A σ_{BA}-structure is a BA model if it satisfies the axioms in Figure 6.1.

The simplest BA model contains a single element when 0 and 1 are collapsed into one element. On the other hand, one of the most popular BA models is the binary BA that contains two elements, *i.e.*, 0 and 1. If we interpret $\cap, \cup, \bar{\cdot}$ as set intersection, union and complement

respectively while 0 is the empty set and 1 is the power set of some set A then we have the set BA model.

Atomless BA. We can define an order $\leq$ from the signature σ_{BA} as:

$$a \leq b \quad \stackrel{\text{def}}{=} \quad a \cap b = a^{\dagger}.$$

Lemma 6.1.1. The order $\leq$ in BA is partial, *i.e.*, it satisfies the following properties:

1. Reflexivity: $\forall a.\ a \leq a$.
2. Anti-symmetry: $\forall a, b.\ a \leq b \to b \leq a \to a = b$.
3. Transitivity: $\forall a, b, c.\ a \leq b \to b \leq b \to a \leq c$.

◁

Proof. Reflexivity follows from axiom idempotency. For anti-symmetry, we have $a \cap b = a$ and $b \cap a = b$. As $\cap$ is commutative, we infer $a = b$. For transitivity, let $a \cap b = a$ and $b \cap c = b$ then together with associativity, we have $a \cap c = (a \cap b) \cap c = a \cap (b \cap c) = a \cap b = a$. Thus $a \leq c$. □

From the partial order $\leq$, we can define the strict order $<$ by excluding the equality case:

$$a < b \quad \stackrel{\text{def}}{=} \quad a \leq b \quad \wedge \quad b \not\leq a.$$

We use $<$ to define atomless BA by adding the following axioms into the BA axiom set:

$$\text{Atomless: } \forall a.\ 0 < a \to \exists b.\ 0 < b \quad \wedge \quad b < a.$$

Informally, the axiom says that for any element a different from 0, we can find an element b between 0 and a exclusively. One atomless BA model is the periodic binary string structure where each element is an infinite string of the form $s^{\omega} = sss\ldots$ in which $s \in \{0,1\}^{+}$ is the infinite periodic string pattern. In this structure, the constant 0 and 1 are the strings 0^{ω}

[†]An alternative definition is $a \cup b = b$ which can be proved to be equivalent from BA axioms.

and 1^ω. Furthermore, the operators $\cap/\cup$ are pair-wise binary conjunction/disjunction and $\bar{\cdot}$ is the component-wise binary complement. For example, let $a = (10)^\omega = 101010\ldots$ and $b = (100)^\omega = 100100\ldots$ then:

1. $\bar{a} = \overline{(10)^\omega} = (01)^\omega$.
2. $a \cap b = (10)^\omega \cap (100)^\omega = (101010)^\omega \cap (100100)^\omega = (101110)^\omega$*.

To see why this structure satisfies the atomless property, let $a = s^\omega$ be an element different from 0^ω. Then $s \neq 0$ and we can pick $b = (s0^n)^\omega$ in which n is the length of s. As $0^n < s$, we have $s0^n < ss$ and thus $b = (s0^n)^\omega < (ss)^\omega = s^\omega = a$.

Another atomless BA model is the structure of propositional formulae in which we have a countably infinite set of variable propositions $P = \{p_1, p_2, \ldots\}$ and each element is a quantifier-free formula constructed from P together with logical connectives $\wedge, \vee$, complement $\neg$, and constants $\bot, \top$. Here 0 and 1 are interpreted as the the contradiction $\bot$ and tautology $\top$. The interpretations of $\cup, \cap, \bar{\cdot}$ are $\vee, \wedge, \neg$ respectively. In this structure, equality should be understood as propositional equivalence, *i.e.*, $a = b$ iff $a \leftrightarrow b$. While it is straightforward to verify this structure satisfies the BA axioms, we will prove that it also satisfies the atomless property. Let a be a quantifier-free formula such that $a \neq \bot$ then there exists a proposition $p_k \in P$ such that p_k is not in a. We pick $b = a \wedge p_k$ then $b \neq \bot$ as we can pick the proposition assignment $f : P(A) \mapsto \{\top, \bot\}$ such that $f(A) = \top$† and extend it to p_k by letting $f(p_k) = \top$ so that $f(B) = \top$ as well. On the other hand, we have $b \leq a$ by construction and thus it remains to show $b \neq a$, *i.e.* they are not equivalent. Extend the assignment f over p_k but this time we let $f(p_k) = \bot$ then $f(A) = \top$ while $f(B) = \bot$ and the result follows.

Both the structures above are countable atomless BA (CABA), *i.e.*, their domains are infinitely countable. It turns out that there is only one CABA model up to isomorphism by the following well-known result:

Proposition 6.1.2 (Forklore *e.g.* [Hal74])**.** The first-order theory of atomless BA is complete and ω-categorical, *i.e.*, any two models are elementarily equivalent and the theory has exactly

*We need to "unfold" a and b so that their periodic strings have the same length before applying the binary disjunction pair-wise.

†Because a is not equivalent to $\bot$.

one countably infinite model up to isomorphism. ◁

Lastly, we recall some classical complexity results for Boolean Algebras that we will need in subsequent sections:

Proposition 6.1.3 ([MO96])**.** Let $\mathcal{B}$ be an infinite BA (*i.e.* its domain is infinite) then the existential theory of $\mathcal{B}$ is $\leq_{\log}$-complete for NP. ◁

Proposition 6.1.4 ([Koz80])**.** The first-order theory of atomless BAs is in $\mathsf{STA}(*, 2^{O(n)}, n)$. Furthermore, it is $\leq_{\log}$-complete for $\mathsf{STA}(*, 2^{n^{O(1)}}, n)$ in which $\mathsf{STA}(*, 2^{O(n)}, n)$ is the complexity of an alternating Turing machine that uses $2^{O(n)}$ time and n alternations*. ◁

6.2 Connection to countable atomless Boolean Algebra

In this section, we show that tree shares $\mathcal{M} = \langle \mathbb{T}, \sqcap, \sqcup, \bar{\cdot}, \circ, \bullet \rangle$ in §2.2.1 is a model for Countable Atomless Boolean Algebra (CABA). We recall the partial order $\sqsubseteq$ and strict order $\sqsubset$ as:

$$a_1 \sqsubseteq a_2 \stackrel{\text{def}}{=} a_1 \sqcap \overline{a_2} = \circ \qquad a_1 \sqsubset a_2 \stackrel{\text{def}}{=} a_1 \sqsubseteq a_2 \wedge a_2 \not\sqsubseteq a_1.$$

The structure $\langle \mathbb{T}, \sqcap, \sqcup, \bar{\cdot}, \circ, \bullet \rangle$ is *atomless* if it satisfies the atomless property:

$$\forall a.\ \circ \sqsubset a \to \exists a'.\ \circ \sqsubset a' \sqsubset a.$$

On the the hand, $\mathcal{M}$ is *countable* if its domain $\mathbb{T}$ is countable. Dockins *et al.* [DHA09] proved that $\mathcal{M}$ is a BA model and thus it remains to show $\mathcal{M}$ is countable and atomless. First, the atomless property can be derived from properties of tree shares:

Lemma 6.2.1. The BA structure $\mathcal{M}$ satisfies the atomless property. ◁

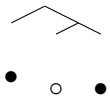
Proof. Let a be a tree share s.t. $\circ \sqsubset a$, we will find a_1 s.t. $\circ \sqsubset a_1 \sqsubset a$. We cheat a little bit

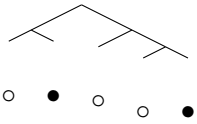
*This is a simple generalization from the result in the paper that states the theory is $\leq_{\log}$-complete for $\mathsf{STA}(*, 2^{O(n)}, n)$. However, this complexity class is bad for completeness because it is not robust under log-space reduction as input's size can increase polynomially.

by using the following $\bowtie$ properties in [DHA09]:

$$\forall b_1, b_2, b_3, a.\ b_1 \sqcap b_2 = b_3 \to (a \bowtie b_1) \sqcap (a \bowtie b_2) = a \bowtie b_3.$$

In particular, we choose $b_1 =$  , $b_2 =$  , $b_3 = \bullet$ then $b_1 \sqcup b_2 = b_3$. Also, let $a_i = a \bowtie b_i$ then $\circ \sqsubset a_i$ and $a_1 \sqcap a_2 = a$. On the other hand, by using BA axioms we have $a \sqcap a_1 = (a_1 \sqcup a_2) \sqcap a_1 = (a_1 \sqcup a_2) \sqcap (a_1 \sqcup \bullet) = a_1 \sqcap (a_2 \sqcup \bullet) = a_1 \sqcap \bullet = a_1$ and thus $a_1 \sqsubseteq a$. As $a_2 \neq \circ$, we imply $a_1 \sqsubset a$. Hence $\mathcal{M}$ is atomless. □

Example 6.2.1. Let $a =$  then

$a_1 = a \bowtie$ $=$ 

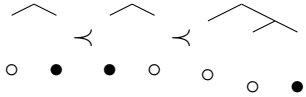
and this gives us $\circ \sqsubset a_1 \sqsubset a$. ◁

The proof that $\mathbb{T}$ is countable is achieved by enumerating $\mathbb{T}$ in the ascending order of tree height $|\tau|$ using the following total strict order $\prec$.

Lemma 6.2.2. Let $|\tau|$ denote the height of τ, we define an order $\prec$ over tree shares as:

$$\frac{}{\circ \prec \bullet} \qquad \frac{|\tau_1| < |\tau_2|}{\tau_1 \prec \tau_2} \qquad \frac{|\tau_1| = |\tau_2| \quad \tau_1 \prec \tau_2}{\tau_1 \prec \tau_2} \qquad \frac{|\tau| = |\tau| \quad \tau_1 \prec \tau_2}{\tau_1 \prec \tau_2}.$$

Then $\prec$ is a total strict order over tree shares. ◁

Proof. The first few elements in the order chain are: $\circ \prec \bullet \prec$  $\prec \ldots$.

To see why $\prec$ is total, we show for any two distinct trees τ_1 and τ_2, either $\tau_1 \prec \tau_2$ or $\tau_2 \prec \tau_1$. If their heights are different then we can use second rule to decide the order. Otherwise, assume $|\tau_1| = |\tau_2| = n$ and we prove by induction over the height n. The base case $n = 0$

can be determined using the first rule. Otherwise, they differs either in their left subtrees or right subtrees and thus we can apply the induction hypothesis together with third and fourth rules to infer the order. Also, notice that the four rules are mutually exclusive of each others and thus consistent. □

Consequently, we are now ready to state the main result of this section:

Theorem 6.2.1. The tree share structure $\mathcal{M} = \langle \mathbb{T}, \sqcap, \sqcup, \bar{\cdot}, \circ, \bullet \rangle$ is a CABA model. ◁

Proof. Follow directly from the fact that $\mathcal{M}$ is BA and Lemmas 6.2.1, 6.2.2. □

Using the above result, we obtain the lower bound complexity for first-order theory and existential theory of $\mathcal{M}$:

Corollary 6.2.1. The first-order theory of $\mathcal{M}$ is $\leq_{\log}$-hard for $\mathsf{STA}(*, 2^{n^{O(1)}}, n)$. On the other hand, its existential theory is $\leq_{\log}$-hard for NP. ◁

Proof. From Prop. 6.1.2, we imply that $\mathcal{M}$ is the unique model for CABA. Thus the lower bound complexity is direct from Prop. 6.1.3 and 6.1.4. □

The two bounds above are unnecessarily upper bounds because formulas in $\mathcal{M}$ contain tree share constants (which are infinitely many) while BA formulas only have two constants $\mathbf{0}$ and $\mathbf{1}$. As a result, we will derive the upper bounds for theories of $\mathcal{M}$ in the next section.

6.3 Upper bound for first-order theory of $\langle \mathbb{T}, \sqcap, \sqcup, \bar{\cdot}, \circ, \bullet \rangle$

As mentioned in the previous section, formulae of $\mathcal{M} = \langle \mathbb{T}, \sqcap, \sqcup, \bar{\cdot}, \circ, \bullet \rangle$ contain tree constants which do not belong to the Boolean Algebra (BA) language. Fortunately, by analyzing the semantics of $\langle \mathbb{T}, \sqcap, \sqcup, \bar{\cdot}, \circ, \bullet \rangle$, we can transform a tree formula Φ into an equivalent tree formula Φ' in polynomial time such that Φ' contains only $\bullet$ and $\circ$ as constants. As a result, we can interpret $\bullet$ as $\mathbf{1}$ and $\circ$ as $\mathbf{0}$ in the BA language and thus Φ' can be decided by Turing machine for atomless BAs. More precisely, we will show the following complexity results of $\mathcal{M}$:

Theorem 6.3.1. The first-order theory $\mathsf{Th}(\mathcal{M})$ is in $\mathsf{STA}(*, 2^{O(n^2)}, n)$. ◁

Together with Corollary 6.2.1, the following bounds are immediate:

Corollary 6.3.1. The first-order theory of $\mathsf{Th}(\mathcal{M})$ is $\leq_{\log}$-complete for $\mathsf{STA}(*, 2^{n^{O(1)}}, n)$. ◁

6.3.1 Definitions and notations

Here we introduce several definitions together with their notations that will be used in subsequent sub-sections. To begin with, we introduce *tree shape*, which is basically the tree skeleton without leaves:

Definition 6.3.1. The shape of a tree τ, denoted by $\langle\tau\rangle$, is obtained by replacing its leaves with $*$:

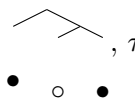
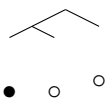
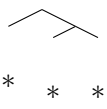
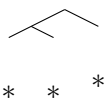
$$\langle e\rangle \stackrel{\text{def}}{=} *,\ e \in \{\bullet, \circ\} \qquad \langle\ \tau_1\ \ \tau_2\ \rangle \stackrel{\text{def}}{=} \langle\tau_1\rangle\ \ \langle\tau_2\rangle\ .$$

We denote the set of tree shapes as $\mathbb{S}$. Let s_1, s_2 be tree shapes then their combined shape, denoted by $s_1 \sqcup s_2$, is defined as*

$$* \sqcup s_2 \stackrel{\text{def}}{=} s_2 \qquad s_1 \sqcup * \stackrel{\text{def}}{=} s_1 \qquad s_1^l\ s_1^r \sqcup s_2^l\ s_2^r \stackrel{\text{def}}{=} s_1^l \sqcup s_2^l \quad s_1^r \sqcup s_2^r\ .$$

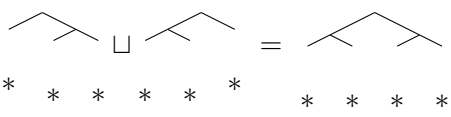
Furthermore, we say s_1 is included in s_2, denoted by $s_1 \sqsubseteq s_2$, if $s_2 = s_1 \sqcup s_2$. We override the shape function over tree formulae by combining all tree shapes in the formula, *i.e.* if $T = \{\tau_1, \ldots, \tau_n\}$ is the set of tree constants in Φ then:

$$\langle\Phi\rangle \stackrel{\text{def}}{=} * \text{ if } T = \emptyset \text{ and } \langle\Phi\rangle \stackrel{\text{def}}{=} \bigsqcup_{i=1}^{n} \langle\tau_i\rangle \text{ otherwise.}$$

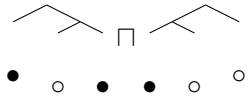
Example 6.3.1. Let $\tau_1 =$  , $\tau_2 =$  then $\langle\tau_1\rangle =$  and $\langle\tau_2\rangle =$  . ◁

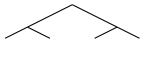
Also:

$$\langle\tau_1\rangle \sqcup \langle\tau_2\rangle = \quad \sqcup \quad = \quad .$$



*Note that shapes are not folded into any canonical form (if they were then the only one would be $*$).

Let $\Phi \stackrel{\text{def}}{=} \exists x.$  $= x$ then:

$$\langle \Phi \rangle \;=\; \langle \tau_1 \rangle \sqcup \langle \tau_2 \rangle \;=\;$$  .

$\triangleleft$

Next, we use tree shape as the unit to measure the *size* of trees, from which we will use to compute precise size for tree formulae:

Definition 6.3.2. The size of a tree shape s, denoted by $\|s\|$, is the number of its leaves:

$$\| * \| \stackrel{\text{def}}{=} 1 \qquad \| \overset{s_1 \quad s_2}{\frown} \| \stackrel{\text{def}}{=} \|s_1\| + \|s_2\|.$$

Meanwhile, we override the size function over trees as the number of its leaves*:

$$\| \bullet \| = \| \circ \| \stackrel{\text{def}}{=} 1 \qquad \| \overset{\tau_1 \quad \tau_2}{\frown} \| \stackrel{\text{def}}{=} \|\tau_1\| + \|\tau_2\|.$$

We override the size function over formulae recursively as follow†:

$$\|v\| \stackrel{\text{def}}{=} 1 \qquad \|\tau_1 = \tau_2\| \stackrel{\text{def}}{=} \|\tau_1\| + \|\tau_2\|$$

$$\|\bar{\Phi}\| = \|\neg\Phi\| = \|Qv.\ \Phi\| \stackrel{\text{def}}{=} \|\Phi\| + 1, Q \in \{\forall, \exists\}$$

$$\|\tau_1 \star \tau_2 = \tau_3\| \stackrel{\text{def}}{=} \|\tau_1\| + \|\tau_2\| + \|\tau_3\|, \star \in \{\sqcup, \sqcap\} \qquad \|\Phi_1 \star \Phi_2\| \stackrel{\text{def}}{=} \|\Phi_1\| + \|\Phi_2\| + 1, \star \in \{\wedge, \vee, \to\}.$$

$\triangleleft$

*We skip internal nodes as they serve no purpose for the computation and there are only linearly $n-1$ of them in a tree with n leaves.

†Note that the size of a tree $\|\tau\|$ is not the same as the height of a tree $|\tau|$. Also, $\|\tau\|$ is not the exact number of bits to represent τ but rather some linear approximation which will not change the overall complexity.

Example 6.3.2. We have $\| \overset{\bullet}{\underset{\circ\ \bullet}{\wedge}} \| = \| \overset{*}{\underset{*\ *}{\wedge}} \| = 3$ and $\|\forall x \exists y.\ \bar{y} \sqcup \overset{\bullet}{\underset{\bullet\ \circ}{\wedge}} = x\| = 7$. ◁

Lastly, the height of a formula Φ is defined to be the height of the highest trees in Φ:

Definition 6.3.3. Let $|\tau|$ be the height of tree τ, starting from zero. We override the height of tree formula Φ as $|\Phi|$ such that $|\Phi| \overset{\text{def}}{=} 0$ if Φ has no constants. Otherwise, let $\tau_1, \ldots, \tau_n$ be tree constants in Φ then $|\Phi| \overset{\text{def}}{=} \max\left(|\tau_i| \mid i = 1 \ldots n\right)$. ◁

Example 6.3.3. $|\overset{}{\underset{\bullet\ \circ\ \bullet}{\wedge}}| = 2$ and $|\forall x \exists y.\ x \sqcup y = \overset{}{\underset{\bullet\ \circ}{\wedge}}| = 1$. ◁

6.3.2 Decision procedure for flattening tree formulas

We propose a decision procedure in Algorithm 9 to transform a tree formula into an equivalent formula of height zero. The heart of our transformation is the function FLATTEN which takes a tree formula Φ as input and computes an equivalent tree formula Φ' such that Φ' only has $\bullet$ and $\circ$ as constants. Hence the complexity of CABA in Prop. 6.1.3 and 6.1.4 can be applied to Φ'. Our key function calls the subroutine SHAPE_DECOMPOSE that helps decompose a single variable or tree constant. In short, the input of SHAPE_DECOMPOSE is a pair of tree τ (or a variable v) and a shape s; whereas the output is a list of subtrees (or copies of v with additional suffix) computed from the decomposition of τ (or v) according to the shape s. Here $\mathsf{concat}(l_1, l_2,)$ concatenates two lists l_1 and l_2.

Example 6.3.4. Let $\tau = \overset{}{\underset{\bullet\ \circ\ \bullet}{\wedge}}$ and $s = \overset{}{\underset{*\ *\ *}{\wedge}}$ then:

$$\text{SHAPE\_DECOMPOSE}(\tau, s) = [\bullet, \bullet, \overset{}{\underset{\circ\ \bullet}{\wedge}}] \text{ and } \text{SHAPE\_DECOMPOSE}(v, s) = [v_{00}, v_{01}, v_1].$$

◁

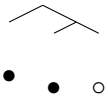
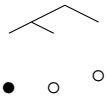
We now explain in detail how to decompose a formula Φ of size $n = \|\Phi\|$ using the procedure FLATTEN. First on line 4, we compute the formula shape $s = \langle\Phi\rangle$ by collectively combining all the tree shapes in Φ. Next between lines 5-9, for each atomic sub-formula $t_1 = t_2$ or $t_1 \star t_2 = t_3$ where $\star \in \{\sqcup, \sqcap\}$, we replace it with the conjunction $\bigwedge_{i=1}^{\|s\|} t_1^j = t_2^j$ or $\bigwedge_{i=1}^{\|s\|} t_1^j \star t_2^j = t_3^j$ in which $[t_i^1, \ldots, t_i^{\|s\|}] = \text{SHAPE\_DECOMPOSE}(t_i, s)$ is the decomposition list

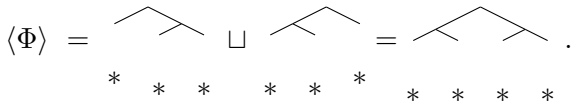
Algorithm 9 Flatten a formula into an equivalent formula of height zero

function FLATTEN(Φ)
Require: Φ is a sentence
Ensure: Return an equivalent sentence Φ' s.t. $\circ$ and $\bullet$ are the only constants.
  if $|\Phi| = 0$ **then return** Φ
  else
    $s \leftarrow \langle \Phi \rangle$
    for each atomic sub-formula $\Psi : t_1 = t_2$ or $t_1 \star t_2 = t_3$, $\star \in \{\sqcup, \sqcap\}$ in Φ **do**
      $[t_i^1, \ldots t_i^{\|s\|}] \leftarrow$ SHAPE_DECOMPOSE(t_i, s)
      $\Psi' \leftarrow \bigwedge_{j=1}^{\|s\|} \Psi_j$ where $\Psi_j \leftarrow t_1^j = t_2^j$ or $t_1^j \star t_2^j = t_3^j$
      $\Phi \leftarrow$ replace Ψ with Ψ'
    end for
    for each quantifier Qv in Φ **do**
      $[v_1, \ldots, v_n] \leftarrow$ SHAPE_DECOMPOSE(v, s)
      $\Phi \leftarrow$ replace Qv in Φ with $Qv_1 \ldots Qv_n$
    end for
    return Φ
  end if
end function

function SHAPE_DECOMPOSE(t, s)
Require: t is either a variable or a tree constant and s is a shape
Ensure: Return a list of subtrees of t by decomposing t according to shape s
  if $s = *$ **then return** $[t]$
  else let $s = \overset{s_1 \quad s_2}{\wedge}$ in
    if t is a variable (v or $\bar{v}$) **then**
      return concat(SHAPE_DECOMPOSE(t_0, s_1), SHAPE_DECOMPOSE(t_1, s_2))
    else if t is $\bullet$ or $\circ$ **then**
      return concat(SHAPE_DECOMPOSE(t, s_1), SHAPE_DECOMPOSE(t, s_2))
    else let $t = \overset{t_1 \quad t_2}{\wedge}$ in
      return concat(SHAPE_DECOMPOSE(t_1, s_1), SHAPE_DECOMPOSE(t_2, s_2))
    end if
  end if
end function

of t_i using subroutine SHAPE_DECOMPOSE. On lines $10-13$, we replace quantifier variables with their corresponding decomposed counterparts and return the modified formula as result.

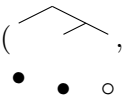
Example 6.3.5. Let $\Phi \stackrel{\text{def}}{=} \forall a \exists b.\ a \sqcup b =$  $\vee\ \neg(a =$  $)$. Then:

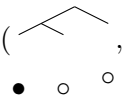


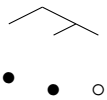
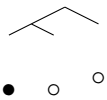
We decompose each variables and constants in Φ as:

$$\text{SHAPE\_DECOMPOSE}(a, \langle\Phi\rangle) = [a_{00}, a_{01}, a_{10}, a_{11}].$$

$$\text{SHAPE\_DECOMPOSE}(b, \langle\Phi\rangle) = [b_{00}, b_{01}, b_{10}, b_{11}].$$

$$\text{SHAPE\_DECOMPOSE}($$  $$, \langle\Phi\rangle) = [\bullet, \bullet, \bullet, \circ].$$

$$\text{SHAPE\_DECOMPOSE}($$  $$, \langle\Phi\rangle) = [\bullet, \circ, \circ, \circ].$$

There are two atomic sub-formulas, $\Psi_1 : a \sqcup b =$  and $\Psi_2 : a =$  . Thus:

$$\begin{aligned} \Psi_1' &= a_{00} \sqcup b_{00} = \bullet \wedge a_{01} \sqcup b_{01} = \bullet \wedge a_{10} \sqcup b_{10} = \bullet \wedge a_{11} \sqcup b_{11} = \circ. \\ \Psi_2' &= a_{00} = \bullet \wedge a_{01} = \circ \wedge a_{10} = \circ \wedge a_{11} = \circ. \end{aligned}$$

As a result, the transformed formula of height zero is computed as:

$$\Phi' = \forall a_{00} \forall a_{01} \forall a_{10} \forall a_{11} \exists b_{00} \exists b_{01} \exists b_{10} \exists b_{11}.\ \Psi_1'\ \vee\ \neg(\Psi_2').$$

$\triangleleft$

6.3.3 Analyzing the upper bound complexity

It remains to analyse the complexity of FLATTEN itself in Algorithm 9. In particular, given a formula Φ of size n, it suffices to show that FLATTEN has time complexity $O(n^2)$ while preserves the number of alternations so that the transformed formula Φ' has size $O(n^2)$ with the same quantifier alternations. Before proving the main result, we will provide several intermediate lemmas. First, we show that the size of the combined shape is limited by the sum of individual sizes:

Lemma 6.3.1. Let $s_1, \ldots, s_n$ be tree shapes then the size of their combined shape is at most the sum of their sizes:

1. $\|s_1 \sqcup s_2\| \leq \|s_1\| + \|s_2\|$.
2. $\|\bigsqcup_{i=1}^{n} s_i\| \leq \sum_{i=1}^{n} \|s_i\|$.

◁

Proof. 1. We prove by structural induction over s_1. For the base case $s_1 = *$, we have:

$$\| * \sqcup s_2\| = \|s_2\| < 1 + \|s_2\| = \| * \| + \|s_2\|.$$

On the other hand, if $s_2 = *$ then the result also follows by symmetric argument. Thus we consider the case $s_1 = \overset{}{\underset{s_1^l \quad s_1^r}{\wedge}}$ and $s_2 = \overset{}{\underset{s_2^l \quad s_2^r}{\wedge}}$. Then by our induction hypothesis:

$$\| \underset{s_1^l \quad s_1^r}{\wedge} \sqcup \underset{s_2^l \quad s_2^r}{\wedge} \| = \| \underset{s_1^l \sqcup s_2^l \quad s_1^r \sqcup s_2^r}{\diagup\!\diagdown} \| = \|s_1^l \sqcup s_2^l\| + \|s_1^r \sqcup s_2^r\|$$

$$\leq \|s_1^l\| + \|s_1^r\| + \|s_2^l\| + \|s_2^r\| = \| \underset{s_1^l \quad s_1^r}{\wedge} \| + \| \underset{s_2^l \quad s_2^r}{\wedge} \|.$$

2. We prove by induction over n. The base case $n = 1$ is trivial. Assume it holds for $n = k$. Consider the case $n = k + 1$ then by Prop. 1:

$$\| \bigsqcup_{i=1}^{k+1} s_i\| = \| \bigsqcup_{i=1}^{k} s_i \sqcup s_{k+1}\| \leq \| \bigsqcup_{i=1}^{k} s_i\| + \|s_{k+1}\|.$$

By our induction hypothesis, we have $\|\bigsqcup_{i=1}^{k} s_i\| \leq \sum_{i=1}^{k} \|s_i\|$ and thus the result follows. □

We generalize the above result to obtain the upper bound for tree formulae:

Lemma 6.3.2. Let Φ be a formula then the size of its shape is at most its size, *i.e.*:

$$\|\langle\Phi\rangle\| \;\leq\; \|\Phi\|.$$

◁

Proof. If Φ does not contain any constant then $\|\langle\Phi\rangle\| = \|*\| = 1$ and thus the inequality trivially holds given the fact that $\|\Phi\| \geq 2$. Otherwise, let $T = \{\tau_1, \ldots, \tau_n\}$ be the set of tree constants in Φ then by Lemma 6.3.1, we have:

$$\|\langle\Phi\rangle\| \;=\; \|\bigsqcup_{i=1}^{n}\langle\tau_i\rangle\| \;\leq\; \sum_{i=1}^{n}\|\langle\tau_i\rangle\| \;\leq\; \|\Phi\|.$$

in which the second inequality follows from the fact that tree constants are parts of the formula. Thus this completes the proof. □

We recall that the co-domain of SHAPE_DECOMPOSE is the n-dimensional list in $\mathbb{T}^n$. Accordingly, we extend the model $\mathcal{M}$ into the n-dimensional model $\mathcal{M}^n = \langle \mathbb{T}^n, \sqcup_n, \sqcap_n, \dot{-}_n\rangle$ in which $\sqcup_n, \sqcap_n, \dot{-}_n$ are defined by applying $\sqcup, \sqcap, \dot{-}$ component-wise. It is straightforward to verify that $\mathcal{M}^n$ is also a CABA. Thus by Proposition 6.1.2 about the uniqueness of isomorphism, two CABA models $\mathcal{M}$ and $\mathcal{M}^n$ are isomorphic. Additionally, we can construct an effective isomorphism between them using the procedure SHAPE_DECOMPOSE:

Lemma 6.3.3. For a shape s such that $\|s\| = n$, the function

$$\textsc{Shape\_decompose}(\_, s) \;\stackrel{\text{def}}{=}\; \lambda\tau.\ \textsc{Shape\_decompose}(\tau, s).$$

is an isomorphism from $\mathcal{M}$ to $\mathcal{M}^n$. ◁

Proof. For convenience, we use the shortcut SD instead of SHAPE_DECOMPOSE. Intuitively, $\mathrm{SD}(\tau, s)$ destructs the tree τ into sub-trees according to the shape s. As Boolean-like operators $\sqcup, \sqcap, \dot{-}$ are defined locally leaf-wise, these operators are still correct for the corresponding

subtrees. For example, assume $\mathrm{SD}(\tau_i, s) = [\tau_i^1, \ldots, \tau_i^m]$ for $i = 1, 2, 3$ then:

$$\tau_1 \sqcup \tau_2 = \tau_3 \quad \text{iff} \quad \bigwedge_{i=1}^{m} \tau_1^i \sqcup \tau_2^i = \tau_3^i \quad \text{iff} \quad \mathrm{SD}(\tau_1, s) \sqcup \mathrm{SD}(\tau_2, s) = \mathrm{SD}(\tau_3, s).$$

Thus the result follows.

□

Since the result list of $l = \textsc{Shape\_decompose}(\tau, s)$ contains subtrees of τ, their heights are strictly smaller than $|\tau|$ if $|\tau| > 0$ and $s \neq *$. Moreover, if we choose s sufficiently large then l will contain only subtrees of height zero:

Lemma 6.3.4. Let τ be a tree and s a shape such that $\langle \tau \rangle \sqsubseteq s$ then all trees in the output of $\textsc{Shape\_decompose}(\tau, s)$ have height zero. ◁

Proof. By induction on the structure of s. For convenience, we use the shortcut SD instead of Shape_decompose. The base case $s = *$ is simple as $\tau \in \{\bullet, \circ\}$. Consider the case $s = \overset{s_l \quad s_r}{\frown}$. If $|\tau| = 0$ then we are done. Otherwise, let $\tau = \overset{\tau_l \quad \tau_r}{\frown}$. Then $\mathrm{SD}(\overset{\tau_l \quad \tau_r}{\frown}, \overset{s_l \quad s_r}{\frown}) = \mathsf{concat}(\mathrm{SD}(\tau_l, s_l), \mathrm{SD}(\tau_r, s_r))$. By our induction hypothesis, both lists $\mathrm{SD}(\tau_l, s_l)$ and $\mathrm{SD}(\tau_r, s_r)$ only have trees of height zero and thus the result follows. □

Finally, the soundness of Flatten follows from Lemmas 6.3.2, 6.3.3 and 6.3.4:

Lemma 6.3.5. Let Φ be a tree formula then

1. $\Phi' = \textsc{Flatten}(\Phi)$ has height zero and is equivalent to Φ.
2. Furthermore, let $n = \|\Phi\|$ be the size of Φ then the time complexity of Flatten is $O(n^2)$ and it preserves the number of quantifier alternations in Φ.

◁

Proof. Prop. 1 follows from Lemmas 6.3.3 and 6.3.4. For Prop. 2, note that the time complexity of Flatten only differs by a constant factor compared to the size of Φ'. Also

by implementation, FLATTEN do no increase the number of quantifier alternations. Thus it remains to prove that the size of Φ' is $O(n^2)$.

Let $s = \langle\Phi\rangle$ be the shape of Φ and $n = \|s\|$ its size. Then for each variable v, $\mathrm{SD}(v, s)$ is a list of n variables. For atomic formula Ψ, FLATTEN decomposes it into $\Psi' = \bigwedge_{i=1}^{n} \Psi_i$ such that $\|\Psi_i\| = O(\|\Psi\|)$ and thus $\|\Psi'\| = O(\Sigma_{i=1}^{n}\|\Psi_i\|) = O(n\|\Psi\|)$, *i.e.*, their sizes differ by a factor of $O(n)$. Generally, two formulae Φ and Φ' also have sizes that differ by a factor of $O(n)$, *i.e.*, $\|\Phi'\| = O(n\|\Phi\|)$. By Lemma 6.3.2, we have $n = \|\langle\Phi\rangle\| \leq \|\Phi\|$ and hence $\|\Phi'\| = O(\|\Phi\|^2)$. □

Corollary 6.3.2. The existential theory of $\mathcal{M}$, *i.e.* $\Sigma_1 \cap \mathsf{Th}(\mathcal{M})$, is NP-complete. ◁

Proof. Notice that FLATTEN does not increase the number of quantifier alternations in the formula and thus by Prop. 6.1.3, $\Sigma_1 \cap \mathsf{Th}(\mathcal{M})$ is NP-complete. □

We are now ready to justify the correctness of Theorem 6.3.1:

Proof of Theorem 6.3.1. Using Lemma 6.3.5, we can transform, in log-space, a tree formula Φ into an equivalent formula Φ' of size $O(n^2)$ that only contains $\bullet, \circ$ as constants and has the same number of quantifier alternations. By Lemma 6.2.1, the formula Φ' is an atomless BA formula. By Proposition 6.1.4, a formula of size $O(n^2)$ with n alternations can be decided in $\mathsf{STA}(*, 2^{O(n^2)}, n)$ and hence the result follows.

6.4 Conclusion

In this chapter, we have demonstrated the decidability and complexity of a first-order theory $\mathcal{M} = \langle\mathbb{T}, \sqcap, \sqcup, \bar{\cdot}, \circ, \bullet\rangle$ over the Boolean logic of tree shares by pinpointing the connection to countable atomless Boolean algebras. From there, we were able to derive the precise complexity $\mathsf{STA}(*, 2^{n^{O(1)}}, n)$ for the first-order theory of $\mathcal{M}$.

Chapter 7

Fragments of $\bowtie$ and their complexity

Izzi: *... They planted a seed over his grave. The seed became a tree. Moses said his father became a part of that tree. He grew into the wood, into the bloom. And when a sparrow ate the tree's fruit, his father flew with the birds. He said... death was his father's road to awe. That's what he called it. The road to awe. Now, I've been trying to write the last chapter and I haven't been able to get that out of my head!*

Tom Creo: *Why are you telling me this?*

Izzi: *I'm not afraid anymore, Tommy.*

The Fountain (2006).

In Chapter 6, we studied the first-order theory complexity of the Boolean-like substructure $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot} \rangle$ in tree shares which is $\mathsf{STA}(*, 2^{n^{O(1)}}, n)$-complete. However, this Boolean-like substructure is not sufficiently expressive to model the scaling permissions in Chapter 3 as we need the operator $\bowtie$ as well. It turned out that bowtie is more complicated than join $\oplus$ because it can increase the tree height. This characteristic is shared with string concatenation that combines strings to create a longer one. In fact, we will show that the two operators $\bowtie$ and string concatenation are similar to each other in the sense that there exists an isomorphism between them. Consequently, we are able to derive the complexity

of first-order theory of $\langle \mathbb{T}, \bowtie \rangle$ which is between NP and PSPACE. As the first-order theory of string structure is undecidable, it follows that the first-order theory of $\langle \mathbb{T}, \bowtie \rangle$ and its extensions are also undecidable. To recover decidability, we need to restrict the form of $\bowtie$ in the formulae. In particular, we will show two fragments of $\bowtie$ with decidable first-order theory together with their complexity. In the first fragment, we show that if we restricted $\bowtie$ with constants either on the left or right then the fragment $\langle \mathbb{T}, \bowtie_\tau, _\tau\bowtie \rangle$ is first-order decidable with complexity $\mathsf{STA}(*, 2^{O(n)}, n)$-complete. In the second fragment, we prove that if we restricted $\bowtie$ with constants on the right then the combined fragment $\langle \mathbb{T}, \bowtie_\tau, \sqcup, \sqcap, \bar{\cdot} \rangle$ is also first-order decidable. This result is especially important as the scaling permission model in Chapter 3 requires both $\oplus$ and $\bowtie$ with constants on the right hand side (as normal recursive functions only need to split resources a finite number of times before passing them to other functions or their children). As a result, it is possible to develop complete decision procedures to handle scaling permission constraints as we did in Chapter 4 and 5. Interestingly, its complexity is non-elementary, *i.e.*, it is not bounded above by any k-exponential time class kEXP mentioned in 6.1.2.

The content of this chapter is structured as follows*:

1. In §7.1, we provide necessary backgrounds in word equations and automatic structures that are useful to derive our results.
2. In §7.2, we establish the isomorphism between the bowtie structure $\langle \mathbb{T}, \bowtie \rangle$ and string structure with concatenation $\langle \mathbb{S}, \cdot \rangle$ and accordingly derive its decidability and complexity.
3. In §7.3, we derive the decidability and complexity of $\langle \mathbb{T}, \bowtie_\tau, _\tau\bowtie \rangle$.
4. In §7.4, we derive the decidability and complexity of $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot}, \bowtie_\tau \rangle$.
5. In §7.5, we draw our conclusion.

*The materials in this chapter are taken from two papers "Decidability and Complexity of Tree Shares Formulas" [LHL16] and "Complexity Analysis of Tree Share Operations" [LHL17], joint work with my supervisor Aquinas Hobor and my mentor Anthony W. Lin.

7.1 Preliminaries

Here we introduce the word equations problem and automatic structures (in particular, tree automatic structures) which we will use to make the connections to bowtie fragments*. Informally, word equations are equational constraints about string concatenation whereas a structure is *tree automatic* if its domain, functions and predicates can be computed using tree automata.

7.1.1 Word equation

Let $A = \{a_1, a_2, \ldots\}$ be a finite set of letters, $\bullet$ be the string concatenation and A^* be the Kleene closure of A using $\bullet$. Furthermore, let $V = \{v_1, v_2, \ldots\}$ be set of variables, and $w \in \mathbb{W} \stackrel{\text{def}}{=} (A \cup V)^*$ a finitely generated word that includes both letters and variables then a *word equation* E is a pair of words $(w_1, w_2) \in \mathbb{W} \times \mathbb{W}$ (we will represent as $w_1 = w_2$). We override *word context* $\rho : V \to A^*$ to the domain $A \cup V$ by mapping constants to themselves, and override ρ to $\mathbb{W}$ by replacing each letter within a word with its value in ρ. Then ρ is a solution of word equation $w_1 = w_2$ if $\rho(w_1) = \rho(w_2)$.

Example 7.1.1. Let $A = \{0, 1\}$ be the alphabet then $x1y = 1x001$ is a word equation over A and $\rho = \{x \mapsto 11, y \mapsto 001\}$ is a solution. ◁

The satisfiability (**SAT**) of word equation asks whether a word equation $w_1 = w_2$ has a solution ρ. Makanin proposed a complete treatment to this problem in a series of papers [Mak77, Mak83, Mak85] but his method was highly intractable (quadruple-exponential non-deterministic time [KP96]). Substantial research since has improved this bound, *e.g.* [AP89, Jaf90]. Also, by a result in [BS90], the existential theory over word equations is known to be PTIME-reducible to **SAT** of a single word equation and thus it is sufficient to analyze solutions for a single word equation. Here we state the best known complexity bounds for word equations:

Proposition 7.1.1 ([Pla04, Pla06, Mar82, Kus06])**.** The **SAT** problem of word equation has lower bound NP-hard and upper bound PSPACE. Furthermore, the first-order theory of

*For fundamental definitions and results in logics and complexity please refer to §6.1.2.

word equation is undecidable. $\triangleleft$

7.1.2 Bottom-up tree automaton

In detail, a bottom-up tree automaton is represented as $\mathcal{A} = \langle Q, F, Q_f, \delta \rangle$ such that:

1. Q is the set of states.
2. $F = \{f_1^{r_1}, \ldots f_m^{r_m}\}$ is the ranked alphabet in which f_i is the alphabet symbol and $r_i \in \mathbb{N}$ is its associated arity.
3. $Q_f \subseteq Q$ is the set of accepting states.
4. δ is the set of transition rules of the form:

$$g(q_1(t_1), \ldots, q_n(t_n)) \quad \mapsto \quad q_{n+1}(g(t_1, \ldots, t_n)).$$

in which $g^n \in F$ is an n-ary symbol, g_i is a state and t_i is a subtree variable. Informally, these rules allows us to infer the state of the parent node from the states of its children.

We call $\mathcal{A}$ *deterministic* if any two rules in δ have different left hand sides, otherwise $\mathcal{A}$ is *nondeterministic*. Let t be a tree term constructed from F then $\mathcal{A}$ runs on t by first applying δ at each leaf of t spontaneously and then proceeding upward. We say $\mathcal{A}$ accepts t if the state associated with the root of t is an accepting state in Q_f.

Example 7.1.2. Let $F = \{\bullet^0, \circ^0, \mathsf{node}^2\}$ be the ranked alphabet for Boolean binary trees. We will construct a tree automaton that only accepts canonical trees. In particular:

1. $Q = \{q_\circ, q_\bullet, q_n\}$ is the set of states.
2. $Q_f = \{q_n\}$ contains a single accepting state.
3. The transition relation δ consists of the following rules:

$$\bullet \mapsto q_\bullet(\bullet) \qquad \circ \mapsto q_\circ(\circ) \qquad \mathsf{node}(q_1(t_1), q_2(t_2)) \mapsto q_n(\mathsf{node}(t_1, t_2)), \{q_n\} \subseteq \{q_1, q_2\}$$

$$\mathsf{node}(q_\circ(t_1), q_\bullet(t_2)) \mapsto q_n(\mathsf{node}(t_1, t_2)) \qquad \mathsf{node}(q_\bullet(t_1), q_\circ(t_2)) \mapsto q_n(\mathsf{node}(t_1, t_2)).$$

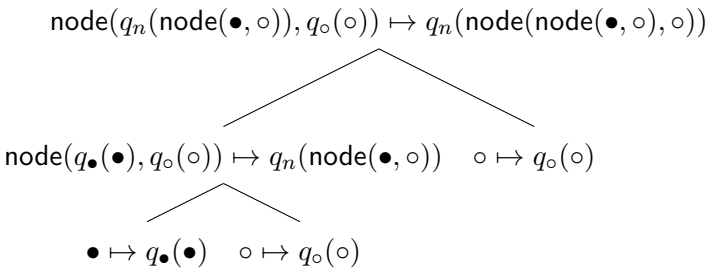
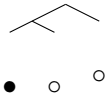


Figure 7.1: An accepting run of tree automaton in Ex. 7.1.2 over $\mathsf{node}(\mathsf{node}(\bullet,\circ),\circ)$.

In short, the automaton will get stuck if it encounters two child leaves with the same value and thus any accepted tree is guaranteed to be in canonical form. We demonstrate in Figure 7.1 an accepting run over the tree  which is explicitly written as $\mathsf{node}(\mathsf{node}(\bullet,\circ),\circ)$. $\triangleleft$

Remark. The counterpart of bottom-up tree automaton (BTTA) is its *top-down* version (TDTA) in which a run over tree term t is executed from root downward leaves and t is accepted if all its leaves are in accepting states. It is well-known (*e.g.* [CDG+07]) that nondeterministic BTTAs have the same expressive power as nondeterministic TDTAs in the sense that one can simulate the other. Interestingly, deterministic BTTAs are more expressive than deterministic TDTAs because informally speaking, the state of the parent node has no information of its children's states. As it is sufficient to construct tree automatic structures from BTTAs, we skip the formal definition of TDTAs.

7.1.3 Tree automatic structures

Informally, a structure $\mathcal{A} = \langle \mathcal{U}, g_1, \dots, g_k \rangle$ is *tree automatic* if its domain $\mathcal{U}$ and function-s/predicates g_i can be computed using tree automata. First, we explain how to construct tree automaton for a predicate P (function is a special predicate in which the last argument is the output). Let $t_1, \dots, t_n$ be tree terms then the *convolution* of tuple $(t_1, \dots, t_n)$ is computed by aligning all $t_1, \dots, t_n$ from left to right together at their roots. As it is common that $t_1, \dots, t_n$ have different shapes, we fill in missing spots using a special symbol $\diamond$.

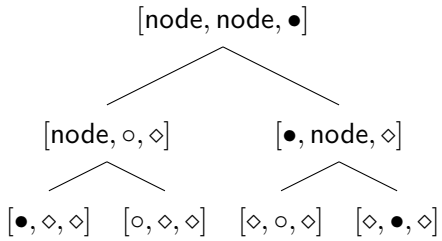
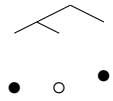
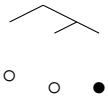
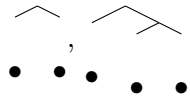


Figure 7.2: The convolution of (t_1, t_2, t_3) in Ex. 7.1.3.

Example 7.1.3. Let $t_1 =$  , $t_2 =$  and $t_3 = \bullet^*$ then the convolution of the triple (t_1, t_2, t_3) is the tree in Fig. 7.2. ◁

Generally, the convolution of predicate P is the convolution set of all its member tuples. We say predicate P is accepted by tree automaton $\mathcal{R}$ if $\mathcal{R}$ accepts the convolution of P.

Example 7.1.4. Here we will construct a tree automaton $\mathcal{R}$ that accepts the tree union $\sqcup$. To make it simple, we temporarily skip the canonical form of input and output trees, *e.g.*, our tree can have trees like  as input or output. Notice that from $\mathcal{R}$, it is fairly straightforward (but tedious due to many combinations) to construct a modified tree automaton $\mathcal{R}'$ in which the canonical form is enforced as done in Ex. 7.1.2. In detail:

1. $A = \{[a, b, c]^r \mid [a, b, c] \in \{\mathsf{node}, \bullet, \circ, \diamond\}^3\}$ is the ranked alphabet in which $[a, b, c]$ has rank 0 if it contains no node and 2 otherwise.
2. $Q = \{q_{(a,b,c)} \mid (a, b, c) \in \{n, \bullet, \circ\}^3\}$ is the set of states.
3. $Q_f = Q$ is the set of accepting states.
4. The transition δ has 'leaf' rules $[e_1, e_2, e_3] \mapsto q_{(e'_1, e'_2, e'_3)}$ that satisfies three conditions:

$$(1)\ (e'_1, e'_2, e'_3) \in \{\circ, \bullet\}^3 \qquad (2)\ e'_1 \sqcup e'_2 = e'_3 \qquad (3)\ \text{if } e_i \in \{\bullet, \circ\} \text{ then } e'_i = e_i.$$

In short, the leaf rules help guess the missing values in some tree components. For example, $[\diamond, \diamond, \bullet] \mapsto [\circ, \bullet, \bullet]$ and $[\diamond, \diamond, \bullet] \mapsto [\bullet, \circ, \bullet]$.

*We remove internal letters node to make the representation more pleasant.

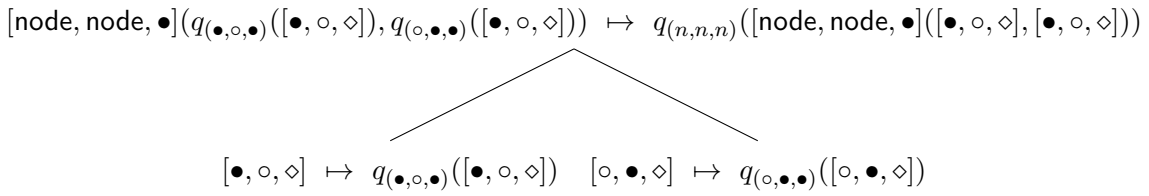
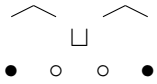


Figure 7.3: An accepting run of $\mathcal{R}$ in Ex. 7.1.4.

Furthermore, δ also contains the 'node' rules:

$$[e_1, e_2, e_3](q_{(a_1^1,a_2^1,a_3^1)}(t_1), q_{(a_1^2,a_2^2,a_3^2)}(t_2)) \mapsto q_{(n,n,n)}([e_1, e_2, e_3](t_1, t_2)).$$

that satisfies the condition: if $e_i \in \{\bullet, \circ\}$ then $a_i^1 = a_i^2 = e_i$. In short, the node rules ensure that the guessing value is indeed consistent with the observed value. For example, $[\mathsf{node}, \mathsf{node}, \bullet](q_{(\bullet,\circ,\bullet)}(t_1), q_{(\circ,\bullet,\bullet)}(t_2)) \mapsto q_{n,n,n}([\mathsf{node}, \mathsf{node}, \bullet](t_1, t_2))$.

We demonstrate an accepting run of $\mathcal{R}$ over the instance  $= \bullet$ in Fig. 7.3. ◁

Finally, we restate a well-known result about the decidability of tree automatic structures:

Proposition 7.1.2 (*e.g.* [BG04, Blu99, KM07, CDG+07])**.** The first-order theory of tree automatic structures is decidable. Furthermore, there exist tree automatic structures whose first-order complexity is non-elementary. ◁

7.2 Decidability of general multiplication $\bowtie$ over tree shares

In this section, we will prove the following results about $\mathcal{M} = \langle \mathbb{T}, \bowtie \rangle$:

Theorem 7.2.1. Let $\mathcal{M} = \langle \mathbb{T}, \bowtie \rangle$ then:

1. The existential theory of $\mathcal{M}$ is in PSPACE.
2. The existential theory of $\mathcal{M}$ is NP-hard.
3. The general first-order theory over $\mathcal{M}$ is undecidable.

◁

The proof of Theorem 7.2.1 largely rests on the identical conclusions for the key subtheory $\mathcal{M}^+ = \langle \mathbb{T}^+, \bowtie \rangle$, where $\mathbb{T}^+ \stackrel{\text{def}}{=} \mathbb{T} \setminus \{\circ\}$ are the "positive trees" obtained by removing the "zero element" $\circ$ from $\mathbb{T}$:

Lemma 7.2.1. Let $\mathcal{M}^+ = \langle \mathbb{T}^+, \bowtie \rangle$ then:

1. The existential theory of $\mathcal{M}^+$ is in PSPACE.
2. The existential theory of $\mathcal{M}^+$ is NP-hard.
3. The general first-order theory over $\mathcal{M}^+$ is undecidable.

$\triangleleft$

We will prove Lemma 7.2.1 shortly, but first let us use it to polish off Theorem 7.2.1:

Proof of Theorem 7.2.1. We take each part in turn as follows:

1. Represent the set of variables $V = \{x_1, \ldots, x_n\}$ in a given formula F of $\mathcal{M}$ as a n-length bitvector. We can enumerate through all possibilities $P_1, \ldots, P_{2^n}$ for this vector using linear space and binary addition. For each possibility P_j, variable x_i's bit is 0 to indicate that x_i must be $\circ$ and 1 when x_i must be non-$\circ$. For each x_k that is marked as $\circ$, we substitute $\circ$ for x_k in F to reach F_j and simplify using the rules

$$\frac{\pi_2 = \circ}{\pi_1 \bowtie \circ = \pi_2} \qquad \frac{\pi_2 = \circ}{\circ \bowtie \pi_1 = \pi_2} \qquad \frac{\pi_1 = \circ \vee \pi_2 = \circ}{\pi_1 \bowtie \pi_2 = \circ}$$

$$\frac{\pi_2 \neq \circ}{\pi_1 \bowtie \circ \neq \pi_2} \qquad \frac{\pi_2 \neq \circ}{\circ \bowtie \pi_1 \neq \pi_2} \qquad \frac{\pi_1 \neq \circ \wedge \pi_2 \neq \circ}{\pi_1 \bowtie \pi_2 \neq \circ}.$$

   We can then just check to make sure that the resulting "fresh" (in)equalities are consistent with the current value of the bitvector P_j. If not, we have reached a contradiction and can proceed to the next bitvector P_{j+1}. If so, then after removing the trivial equalities (*e.g.* $\circ = \circ$) from F_j we are left with an equivalent formula F_j^+ which is in $\mathcal{M}^+$, so by Lemma 7.2.1.1 we can check if F_j is satisfiable in PSPACE. If so, we know that F_j is satisfiable, and thus that F is satisfiable. If not, we proceed to the next bitvector P_{j+1}; if all F_j are unsatisfiable then F is unsatisfiable.

2. By Lemma 7.2.1.2 it is sufficient to reduce a formula F^+ in $\mathcal{M}^+$ to $\mathcal{M}$. Let V be the set of variables in F^+ and define $F \triangleq F^+ \wedge \left(\bigwedge_{x \in V} x \neq \circ \right)$; note that we construct F in linear time from $|F^+|$. F is satisfiable in $\mathcal{M}$ if and only if F^+ is satisfiable in $\mathcal{M}^+$, so we are done.

3. Any extension of an undecidable theory is also undecidable and thus we are done.

□

7.2.1 Infinite alphabets

To define our isomorphism from $\mathbb{T}^+$ to A^* it will be convenient if the alphabet A can be countably infinite. Accordingly, we must reduce word equations over an infinite alphabet to the standard finite case. Let σ be the function that extracts the set of alphabet letters from a word w, *e.g.* $\sigma(v_1 a_1 a_3 v_2) = \{a_1, a_3\}$, we override σ to word equation $w_1 = w_2$ as $\sigma(w_1 = w_2) \stackrel{\text{def}}{=} \sigma(w_1) \cup \sigma(w_2)$. Furthermore, let ϕ be the projection function that takes a word w and a set of letters $B \subseteq A$ and removes all letters in w that are not in B, *e.g.*, $\phi(v_1 a_1 a_3 v_2, \{a_1, a_2\}) = v_1 a_1 v_2$. It follows that ϕ with fixed B is homomorphism over A, *i.e.*, $w_1 \cdot w_2 = w_3$ iff $\phi(w_1, B) \cdot \phi(w_2, B) = \phi(w_3, B)$. Now we are ready to state and prove the extension to infinite alphabets:

Lemma 7.2.2. Let A be infinite alphabet and $e \stackrel{\text{def}}{=} w_1 = w_2$ a word equation over A. Then e is satisfiable over A iff e is satisfiable over the finite alphabet $\sigma(e)$. ◁

Proof. $\Leftarrow$ is trivial. Let ρ be a solution of e over A, we define $\rho' \stackrel{\text{def}}{=} \lambda v.\ \phi(\rho(v), \sigma(e))$ to be the restriction of ρ over the finite alphabet $\sigma(e)$. Notice that ρ' preserves all the letters in e and $\rho(w_1) = \rho(w_2)$ implies $\rho'(w_1) = \rho'(w_2)$. Thus ρ' is a new solution of e that only contains letters from finite alphabet $\sigma(e)$. □

On the other hand, we also need to deal with disequations of the form $w_1 \neq w_2$. Notice that the result in Lemma 7.2.2 is not applicable for disequations, *e.g.*, $a1 \neq 1a$ is not satisfiable for any $a \in \{1\}^*$ although it is satisfiable with $a = 2$. Fortunately, this problem is fairly easy to fix by allowing the new finite alphabet to contain some extra letters to mark the

difference between two words in a disequation. In particular, it is sufficient to add n extra letters for n disequations:

Lemma 7.2.3. Let A be infinite alphabet and $S = \{e_1, \dots e_{m+n}\}$ be a system of m word equations and n word disequations over A. Then S is satisfiable over A iff S is satisfiable over $\sigma(S) \cup \{a_1, \dots, a_n\}$ where a_i is some extra letter in A but not in $\sigma(S)$. ◁

Proof. For a disequation $w_1 \neq w_2$ to hold, it suffices that w_1, w_2 are different at least one place. Let ρ be a solution of S then for each disequation $w_1 \neq w_2$, we preserve letters in the first position that w_1, w_2 differ while changing all other letters not in $\sigma(S)$ with a fixed letter in $\sigma(S)$. For equation $w_1 = w_2$, we use the same trick as in Lemma 7.2.2 but instead remove letters not in $\sigma(S)$, we replace them with some fixed letter in $\sigma(S)$ so that the word's length is preserved. As a result, we only need at most one extra letter for each disequation and thus effectively reduce the infinite alphabet to finite one. □

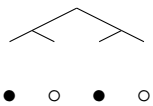
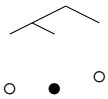
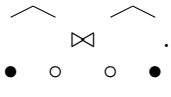
7.2.2 Finding an infinite alphabet inside $\mathbb{T}^+$

Since $\bowtie$ is a kind of multiplication operation, and the fundamental building blocks of $\langle \mathbb{N}, \times \rangle$ are prime numbers, it is natural to wonder whether there is an analogue on trees. There is:

Definition 7.2.1. A tree $\tau \neq \bullet$ is *prime* if it cannot be factorized into smaller trees, *i.e.*:

$$\forall \tau_1, \tau_2.\ \tau = \tau_1 \bowtie \tau_2 \Rightarrow (\tau_1 = \bullet \vee \tau_2 = \bullet).$$

Furthermore, let $\mathsf{Prime}(\tau)$ indicate τ is prime and $\mathbb{P}$ be the set of all prime trees. ◁

Example 7.2.1. Examples of tree primes are  and  . On the other hand, the tree  is not prime since it can be factored as  . ◁

To begin with, we need the following technical lemma which allows us to split an application of bowtie $\tau_2 \bowtie \tau_3$ to children of τ_2:

Lemma 7.2.4. Let $\tau_1, \tau_2, \tau_3, \tau_1^l, \tau_1^r \in \mathbb{T}^+$ and $\tau_1 = \tau_2 \bowtie \tau_3 \wedge \tau_1 =$  then either one of

the following properties holds:

1. $\tau_2 = \bullet$ and $\tau_1 = \tau_3$, or
2. There exist τ_2^l, τ_2^r such that $\tau_2 =$  and $\tau_1^l = \tau_2^l \bowtie \tau_3$ and $\tau_1^r = \tau_2^r \bowtie \tau_3$.

◁

Proof. The case $\tau_2 = \bullet$ is trivial. Otherwise, there exist $\tau_2^l, \tau_2^r \in \mathbb{T}$ such that $\tau_2 =$  .

By definition of $\bowtie$, $\tau_1 = \tau_2 \bowtie \tau_3$ is computed by replacing each leaf $\bullet$ in τ_2 with τ_3, which is equivalent to replace each leaf $\bullet$ in τ_2^l and τ_2^r with τ_3.

Thus, $\tau_1^l = \tau_2^l \bowtie \tau_3$ and $\tau_1^r = \tau_2^r \bowtie \tau_3$. □

Now we show that prime trees are finitely many and have similar characteristics as prime numbers in which they can be used as the basis to represent the entire domain:

Lemma 7.2.5. Prime trees have the following properties:

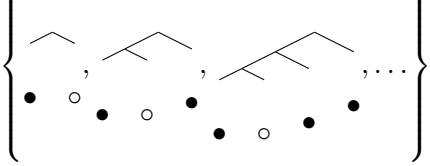
1. There are countably infinitely many prime trees.
2. Let $\tau_1, \tau_1', \tau_2, \tau_2' \in \mathbb{P}$, $\tau_1 \bowtie \tau_2 = \tau_1' \bowtie \tau_2'$ iff $\tau_1 = \tau_1'$ and $\tau_2 = \tau_2'$.
3. Two prime tree sequences $\{\tau_1^i\}_{i=1}^{k_1}$ and $\{\tau_2^i\}_{i=1}^{k_2}$ are equal iff their $\bowtie$ products are equal:

$$\bowtie_{i=1}^{k_1} \tau_1^i = \bowtie_{i=1}^{k_2} \tau_2^i \Leftrightarrow (k_1 = k_2 \bigwedge_{i=1}^{k_1} \tau_1^i = \tau_2^i).$$

◁

Proof of Lemma 7.2.5. We let $\mathbb{T}^n$ be the set of trees with height at most n.

1. We construct an infinite sequence S of prime trees: let $p_1 \stackrel{\text{def}}{=}$  , $p_j \stackrel{\text{def}}{=}$  , *i.e.*:

$$S \stackrel{\text{def}}{=} \left\{ \text{  } \right\}.$$

It is immediate that p_1 is prime. To prove that p_i is prime for $i > 1$, we proceed as follows. Suppose $p_i = \tau_1 \bowtie \tau_2$ and neither τ_1 nor τ_2 is $\bullet$. The right subtree of each p_i is just $\bullet$ and by the definition of $\bowtie$ must contain a copy of τ_2, *i.e.* $\tau_2 = \bullet$, so we have a contradiction and p_i is prime.

2. We prove by induction on the height of τ_1, τ_1'. The base case $\mathbb{T}^0$ is easy to verify. Assume it holds for $\mathbb{T}^k$ and $\tau_1, \tau_1' \in \mathbb{T}^{k+1}$. Let $\tau_1 = \overset{\wedge}{\tau_1^l \quad \tau_1^r}$ and $\tau_1' = \overset{\wedge}{\tau_1^{l'} \quad \tau_1^{r'}}$ then by Lemma 7.2.4, we derive:

$$\tau_1^l \bowtie \tau_2 = \tau_1^{l'} \bowtie \tau_2' \quad \text{and} \quad \tau_1^r \bowtie \tau_2 = \tau_1^{r'} \bowtie \tau_2'.$$

The induction hypothesis yields $\tau_1^l = \tau_1^{l'}$, $\tau_1^r = \tau_1^{r'}$ and $\tau_2 = \tau_2'$. Consequently, $\tau_1 = \tau_1'$.

3. This is a simple generalization of property 2.

□

Of course the real fun with prime numbers is the the unique factorization theorem. Since $\bowtie$ is not commutative we get a stronger version of the traditional theorem:

Lemma 7.2.6. For each $\tau \in \mathbb{T}^+ \backslash \{\bullet\}$, there exists a unique prime sequence $\tau_1, ..., \tau_n \in \mathbb{P}$ such that $\tau = \bowtie_{i=1}^n \tau_i$. ◁

Proof. We prove by induction on the height of τ. The base case $\mathbb{T}^1$ is trivial. Assume it holds for $\mathbb{T}^k$ and let $\tau \in \mathbb{T}^{k+1}$. If τ is prime then we are done. Otherwise, let $\tau^1, \tau^2 \in \mathbb{T}^k \backslash \{\bullet\}$ and $\tau = \tau^1 \bowtie \tau^2$. By our induction hypothesis, there are 2 sequences $\tau_1^1, ..., \tau_{k_1}^1 \in \mathbb{T}_p$ and $\tau_1^2, ..., \tau_{k_2}^2 \in \mathbb{T}_p$ such that $\tau^1 = \bowtie_{i=1}^{k_1} \tau_i^1$ and $\tau^2 = \bowtie_{i=1}^{k_2} \tau_i^2$ and thus $\tau = (\bowtie_{i=1}^{k_1} \tau_i^1) \bowtie (\bowtie_{i=1}^{k_2} \tau_i^2)$. The uniqueness is a consequence of property 3 from Lemma 7.2.5. □

Corollary 7.2.1. The prime set $\mathbb{P} \cup \{\bullet\}$ is a basis of $\mathcal{M}^+$, *i.e.* the closure of $\mathbb{P}$ over $\bowtie$ together with $\bullet$ is $\mathbb{T}^+$. Furthermore, it is the smallest basis: if B is a basis of $\mathcal{M}^+$ then $\mathbb{T}_p \cup \{\bullet\} \subseteq B$. ◁

Accordingly, we will use $\mathbb{P}$ as our "infinite alphabet" in our isomorphism.

7.2.3 Connecting tree shares to word equations

Recall that $\mathbb{P}$ is the infinite alphabet of prime trees and $\langle \mathbb{P}^*, \bullet \rangle$ is the corresponding string structure with concatenation. We are ready to make the central connection needed for Lemma 7.2.1:

Lemma 7.2.7. The structure $\langle \mathbb{T}^+, \bowtie \rangle$ is isomorphic to $\langle \mathbb{P}^*, \bullet \rangle$. ◁

Proof. Let $f : \mathbb{T}^+ \to \mathbb{P}^*$ be defined as follows. First, map the identity element $\bullet$ to the empty word ϵ and then for each prime tree $\tau_{\mathsf{p}} \in \mathbb{T}^+$ map τ_{p} to itself. Finally, for each composite $\tau \in \mathbb{T}^+$ map τ to exactly the concatenation of its (unique) prime factors.

We now wish to prove that for any τ_1 and τ_2, $f(\tau_1 \bowtie \tau_2) = f(\tau_1) \bullet f(\tau_1)$. Let us consider the easy cases first. If $\tau_1 = \bullet$ then $f(\tau_1 \bowtie \tau_2) = f(\tau_2) = \epsilon \bullet f(\tau_2) = f(\tau_1) \bullet f(\tau_2)$. The situation is symmetric when $\tau_2 = \bullet$. Now let us consider the case when neither τ_1 nor τ_2 is $\bullet$. Let $p_1, \ldots, p_i$ be the unique prime factors of τ_1 and $p'_1, \ldots, p'_j$ be the unique prime factors of τ_2. By Lemma 7.2.6, $p_1, \ldots, p_i, p'_1, \ldots, p'_j$ are exactly the unique prime factors of $\tau_1 \bowtie \tau_2$, so:

$$f(\tau_1 \bowtie \tau_2) \;=\; f(p_1 \bowtie \cdots \bowtie p_i \bowtie p'_1 \bowtie \cdots \bowtie p'_j) \;=\; p_1 \bullet \cdots \bullet p_i \bullet p'_1 \bullet \cdots \bullet p'_j \;=$$

$$(p_1 \bullet \cdots \bullet p_i) \bullet (p'_1 \bullet \cdots \bullet p'_j) \;=\; f(\tau_1) \bullet f(\tau_2).$$

To prove f is surjective, let $w \in \mathbb{P}^*$ be the concatenation of primes $p_1 \bullet \cdots \bullet p_i$; then by the definition $f(p_1 \bowtie \cdots \bowtie p_i) = w$. To prove f is injective, suppose $f(\tau_1) = f(\tau_2)$. Let $p_1, \ldots p_i$ be the prime factors of τ_1 and $p'_1, \ldots p'_j$ be the prime factors of τ_2. Accordingly we know that $p_1 \bullet \cdots \bullet p_i = p'_1 \bullet \cdots \bullet p'_j$, and since equality over words can only occur if the words have the same length and have the same letters, we know $i = j$ and $p_k = p'_k$ for all k. □

Corollary 7.2.2. Similar to word equations, tree equations e over $\langle \mathbb{T}^+, \bowtie \rangle$ contain tree constants and variables; we can map these to word equations e' over $\langle \mathbb{P}^*, \bullet \rangle$ by mapping variables to themselves, constants to the concatenation of their prime factors, and multiplication $\bowtie$ to concatenation $\bullet$. The resulting system is equivalent, *i.e.* if $\rho : V \to \mathbb{T}^+$ satisfies e then $f \circ \rho$ satisfies e', where $\circ$ in this case means functional composition and f is the isomorphism constructed in Lemma 7.2.7. ◁

We are now ready to start tackling Lemma 7.2.1. We start with the simplest:

Proof of Lemma 7.2.1.3. As mentioned in Prop. 7.1.1, the first order theory over word equations is known to be undecidable. By Lemma 7.2.7 we know that this theory is isomorphic to the first order theory over tree shares with $\bowtie$, which accordingly must be undecidable. □

To show Lemma 7.2.1.1 we need to know that tree factorization can be done within PSPACE. In fact we can do much better:

Lemma 7.2.8. Factoring an arbitrary positive tree share τ is in PTIME. ◁

Proof. Let $\mathbb{S}(\tau)$ be the set of all subtrees of τ and $\mathbb{S}_n(\tau) \subset \mathbb{S}(\tau)$ be the set of all subtrees of τ with height exactly n. $\mathbb{S}(\tau)$ can be computed recursively:

$$\mathbb{S}(\circ) = \{\circ\} \qquad \mathbb{S}(\bullet) = \{\bullet\} \qquad \mathbb{S}(\overset{\frown}{\underset{\tau_1 \quad \tau_2}{\quad}}) = \mathbb{S}(\tau_1) \cup \mathbb{S}(\tau_2) \cup \{\overset{\frown}{\underset{\tau_1 \quad \tau_2}{\quad}}\}.$$

If $\tau = \tau_1 \bowtie \tau_2$ ($\{\tau_1, \tau_2\} \subset \mathbb{T}^+ \backslash \{\bullet\}$), then there exists $n \in \mathbb{N}$ such that $\mathbb{S}_n(\tau) = \{\tau_2\}$, that is, $\mathbb{S}_{|\tau_2|}(\tau)$ is exactly the singleton set $\{\tau_2\}$. Additionally, $S(\tau) = \bigcup_{i=0}^{|\tau|} S_i(\tau)$.

Thus we can find all the prime factors of τ (which is inspired from the well-known sieve of Eratosthenes) as follows: first we compute $\mathbb{S}(\tau)$ and partition it into $\mathbb{S}_0(\tau), \ldots, \mathbb{S}_{|\tau|}(\tau)$. Let $i \in \mathbb{N}$ be the smallest number such that $\mathbb{S}_i(\tau)$ is the singleton set $\{\tau_1\}$ for some $\tau_1 \in \mathbb{T}$ (note that i must be larger than 0 since $\mathbb{S}_0(\tau) = \{\circ, \bullet\}$). If $i = |\tau|$ then τ itself is a prime, otherwise, we replace all subtrees τ_1 of τ with $\bullet$ and call the new tree τ'. If all the "old" $\bullet$ leaves of τ are replaced and τ' is in canonical form then $\tau = \tau' \bowtie \tau_1$, τ_1 is a prime factor of τ, and we can repeat the process with τ' to find the next prime factor. Otherwise, we consider the next singleton set $\mathbb{S}_j(\tau)$.

If τ has n leaves than its description requires $O(n)$ bits and the time to compute $\mathbb{S}(\tau)$ is $O(n)$. Note that $|\mathbb{S}_k(\tau)| \leq \frac{n}{k+1}$ because there are at least $k+1$ leaves in a tree of height k. Therefore, the number of subtrees from height 1 to n is at most $\Sigma_{i=1}^{n} \frac{n}{i+1} \leq n^2$. Computing the height of a subtree τ' of τ requires $O(n)$, thus the time to partition $\mathbb{S}(\tau)$ is $O(n^3)$. The number of times we need to restart the process is $O(n^2)$. Consequently, the time for tree

factorization is $O(n^5)$, polynomial in the description of τ (more efficient solutions exist). □

Tree factorization is fundamentally simpler than integer factorization since the representation of a tree already contains the descriptions of all of its tree factors. In contrast, the connection between the representation of a number and the representation of its prime factors is vague: *e.g.* among the 24 factors of 74,611,647 are 333 (which does not appear at all in the representation of the original) and 8,290,183 (which only shares a single 1 with the original).

Proof of Lemma 7.2.1.1. We take the tree shares and factor them using Lemma 7.2.8 and then construct the isomorphic system of word equations using the calculated prime factors as the alphabet using Corollary 7.2.2. As mentioned in Prop. 7.1.1, the best known complexity bound for the existential word equation problem is PSPACE. □

For Lemma 7.2.1.2 we need one final fact:

Lemma 7.2.9. For any n (represented in unary) we can find a length-n sequence of tree primes S in polynomial time of n. ◁

Proof. Consider the sequence S from Lemma 7.2.5: the description of p_i is only a constant size larger than the description of p_{i-1} so the description of S is quadratic in n. □

Proof of Lemma 7.2.1.2. Suppose we have an arbitrary problem Q in NP. We can reduce Q to word equations in polynomial time [Pla04, Pla06]. We then use Lemma 7.2.9 to construct a set of primes the size of the number of alphabet letters that appear in the equations and map each letter in the word alphabet to a distinct prime, creating a set of word equations over $\mathbb{P}^*$. Since the representation of the constants does not affect the computational properties of the theory, we can conclude that $\mathbb{T}^+$ is NP-hard. □

7.3 Fragment $\langle \mathbb{T}, \bowtie_\tau, _\tau\bowtie \rangle$

The multiplication operator $\bowtie$ is a complicated operator compared to the Boolean operators because it can increase the tree height. Informally, the operator $\bowtie$ is analogous to the string concatenation operator as both can be used to create a longer structure. In fact, $\bowtie$ can be

reduced to string concatenation as proved in §7.2. As a result, the first-order theory over $\langle \mathbb{T}, \bowtie \rangle$ is undecidable, although its existential theory is decidable in PSPACE. Accordingly, we are interested in a restriction of that theory that will recover decidability for first-order reasoning. We restrict $\bowtie$ to take only constants as one operand, obtaining the two families of unary operators indexed by constants τ:

$$ {}_{\tau}\bowtie(x) \stackrel{\text{def}}{=} \tau \bowtie x \quad \text{and} \quad \bowtie_{\tau}(x) \stackrel{\text{def}}{=} x \bowtie \tau. $$

In §7.3.1, we report the decidability and complexity result together with the main proof. In §7.3.2, we prove the key lemma that is essential to derive our main results.

7.3.1 Decidability and complexity result

Here we will show that the first-order theory of $\mathcal{R} = \langle \mathbb{T}, \bowtie_{\tau}, {}_{\tau}\bowtie \rangle$ is elementary with complexity $\leq_{\text{log-lin}}$-complete for $\mathsf{STA}(*, 2^{O(n)}, n)$*:

Theorem 7.3.1. The complexity of $\mathsf{Th}(\mathcal{R})$ is $\leq_{\text{log-lin}}$-complete for $\mathsf{STA}(*, 2^{O(n)}, n)$. ◁

We prove Theorem 7.3.1 by solving a similar problem in which $\bullet, \circ$ are excluded from the domain $\mathbb{T}$. That is, let $\mathbb{T}^+ = \mathbb{T} \backslash \{\bullet, \circ\}$ and $\mathcal{R}^+ = \langle \mathbb{T}^+, {}_{\tau}\bowtie, \bowtie_{\tau} \rangle$, we want:

Lemma 7.3.1. The complexity of $\mathsf{Th}(\mathcal{R}^+)$ is $\leq_{\text{log-lin}}$-complete for $\mathsf{STA}(*, 2^{O(n)}, n)$. ◁

The proof of Theorem 7.3.1 from Lemma 7.3.1. The hardness proof is direct from the fact that $\tau \in \mathcal{R}^+$ can be expressed as $\tau \in \mathcal{R} \wedge \tau \neq \circ \wedge \tau \neq \bullet$. The proof for upper bound is obtained by 'guessing' the values of variables. In particular, we partition the domain $\mathbb{T}$ into three disjoint sets $S_0 = \{\circ\}, S_1 = \{\bullet\}$ and $S_2 = \mathbb{T}^+$. We then use a ternary vector of length n to 'guess' the partition domain of n variables in the input formula, *e.g.*, $i \mapsto S_i$ for $i = 0, 1, 2$. If a variable v is 'guessed' to be in S_0 or S_1, we substitute v with either $\circ$ or $\bullet$ respectively. Next, each term $\bowtie_{\tau}(a)$ or ${}_{\tau}\bowtie(a)$ that contains $\bullet$ or $\circ$ is simplified using the following identities:

$$ \tau \bowtie \bullet \;=\; \bullet \bowtie \tau \;=\; \tau \qquad \tau \bowtie \circ \;=\; \circ \bowtie \tau \;=\; \circ\,. $$

*Please refer to §6.1.2 for definition of alternating Turing machines and their complexity class $\mathsf{STA}(s(n), t(n), a(n))$.

After this transformation, all the atomic sub-formulas that contain $\circ$ or $\bullet$ are either $v_1 = v_2$ or trivial equalities that can be replaced by either $\top$ or $\bot$. As a result, the new equivalent formula is free of $\bullet$ and $\circ$ and all variables are restricted to $\mathbb{T}^+$. Hence the formula can be solved by the Turing machine that decides $\mathsf{Th}(\mathcal{R}^+)$. The whole guessing process does not increase the formula's size or number of quantifier alternations. Thus adding this extra guessing process will not increase the total complexity of $\mathsf{Th}(\mathcal{R})$. □

7.3.2 Connection to string structure with successors

In this subsection, we will devote to the proof of Lemma 7.3.1. To prove the complexity $\mathsf{Th}(\mathcal{R}^+)$, we construct a log-space isomorphism from $\mathcal{R}^+$ to the structure of ternary strings with prefix and suffix successors. Furthermore, the transformed formula has linear size $O(n)$ where n is the size of the original formula. Here we recall a result from [RV03]:

Proposition 7.3.1 ([RV03])**.** Let $\mathcal{S} = \langle \{0,1\}^*, P_0, P_1, S_0, S_1 \rangle$ be the structure of binary trees with prefix successors P_0, P_1 and suffix successors S_0, S_1 such that:

$$P_0(s) = 0 \cdot s \qquad P_1(s) = 1 \cdot s \qquad S_0(s) = s \cdot 0 \qquad S_1(s) = s \cdot 1.$$

Then the first-order theory of $\mathcal{S}$ is $\leq_{\text{log-lin}}$-complete for $\mathsf{STA}(*, 2^{O(n)}, n)$. ◁

The above result can be generalized to successors P_s and S_s with the same complexity:

Lemma 7.3.2. Let Σ be a finite alphabet of size $k \geq 2$ and $\mathcal{S}' = \langle \Sigma^*, P_s, S_s \rangle$ the structure of binary trees with infinitely many prefix successors P_s and suffix successors S_s for $s \in \Sigma^*$ such that:

$$P_s(s') = s \cdot s' \qquad S_s(s') = s' \cdot s.$$

Then the first-order theory of $\mathcal{S}'$ is $\leq_{\text{log-lin}}$-complete for $\mathsf{STA}(*, 2^{O(n)}, n)$. ◁

Proof. The proof in [RV03] can be naturally generalized to finite alphabet Σ of size $k \geq 2$ with k prefix and suffix successors. For a string $s \in \Sigma^*$ such that $s = a_1 \dots a_n$ and $a_i \in \Sigma$,

the successors P_s and S_s can be defined in linear size from successors in $\mathcal{S}$ as below:

$$P_s \stackrel{\text{def}}{=} \lambda s'.\ P_{a_1}(\ldots P_{a_n}(s')) \qquad S_s \stackrel{\text{def}}{=} \lambda s'.\ S_{a_n}(\ldots S_{a_1}(s')).$$

As these definitions require no additional quantifier variable, the result follows. □

Next, we recall some key results from §7.2.2 about the construction of an isomorphism between trees and strings in word equation:

Definition 7.3.1. A tree τ in $\mathbb{T}\backslash\{\circ, \bullet\}$ is prime if $\tau = \tau_1 \bowtie \tau_2$ implies $\tau_1 = \bullet$ or $\tau_2 = \bullet$. ◁

Proposition 7.3.2. Each tree in $\mathbb{T}\backslash\{\circ, \bullet\}$ is uniquely represented as a sequence of prime trees $\{\tau_i\}_{i=1}^{n}$ s.t. $\tau = \tau_1 \bowtie \cdots \bowtie \tau_n$.

As a result, each tree in $\mathbb{T}\backslash\{\circ, \bullet\}$ can be treated as a string in a word equation in which the alphabet is $\mathbb{P}$, the countably infinite set of prime trees, and $\bowtie$ is the string concatenation. ◁

We show how to encode trees using ternary strings from $\{0, 1, 2\}^*$. Since $\mathbb{P}$ is countably infinite, we can construct a bijective *encoding function* $I : \mathbb{P} \mapsto \{0, 1\}^*$ that encodes each prime tree as a binary string including the empty string. The mapping $\hat{I}$ from $\mathbb{T}^+$ to ternary strings in $\{0, 1, 2\}^*$ is constructed from I by using 2 as delimiter between two consecutive prime trees:

Lemma 7.3.3. Let $\hat{I} : \mathbb{T}^+ \mapsto \{0, 1, 2\}^*$ be a mapping s.t. $\hat{I}(\tau) = I(\tau)$ for $\tau \in \mathbb{P}$. Otherwise, let $\tau \in \mathbb{T}^+$ s.t. $\tau = \tau_1 \bowtie \ldots \bowtie \tau_n$, $\tau_i \in \mathbb{P}$ then $\hat{I}(\tau) = I(\tau_1) \cdot 2 \ldots 2 \cdot I(\tau_n)$.

By Prop. 7.3.2, $\hat{I}$ is bijective and $\hat{I}(\tau_1 \bowtie \tau_2) = \hat{I}(\tau_1) \cdot 2 \cdot \hat{I}(\tau_2)$. ◁

We are now ready to make the connection between Lemma 7.3.2 and 7.3.3 by constructing the isomorphism from $\bowtie$-structure $\mathcal{R}^+$ to ternary string structure $\mathcal{S}'$:

Lemma 7.3.4. Let $f : \langle \mathbb{T}^+, {}_\tau\bowtie, \bowtie_\tau \rangle \mapsto \langle \{0, 1, 2\}^*, P_s, S_s \rangle$ s.t.:

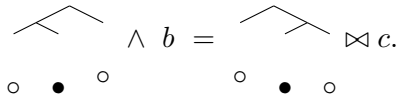
1. For each tree $\tau \in \mathbb{T}^+$, we let $f(\tau) \stackrel{\text{def}}{=} \hat{I}(\tau)$.
2. For each function ${}_\tau\bowtie$, we let $f({}_\tau\bowtie) \stackrel{\text{def}}{=} P_{\hat{I}(\tau)}$.
3. For each function $\bowtie_\tau$, we let $f(\bowtie_\tau) \stackrel{\text{def}}{=} S_{\hat{I}(\tau)}$.

Then f is an isomorphism from $\mathcal{R}^+$ to $\mathcal{S}'$. ◁

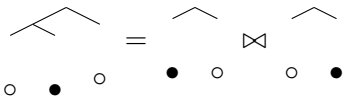
Proof of Lemma 7.3.1. We use the function f in Lemma 7.3.4 to reduce formulae in $\mathcal{R}^+$ to formulae in ternary string structure $\mathcal{S}'$. It remains to ensure for a tree $\tau \in \mathbb{T}^+$ of size n, its ternary string $f(\tau)$ only has linear size $O(n)$. This can be done by constructing the encoding function I *after observing the input formula.* To be precise, given a formula Φ of $\mathcal{R}$, we first factorize all its tree constants into prime trees, which is in log-space as shown in Lemma 7.2.8. Suppose the formula contains n prime trees $\{\tau_i\}_{i=1}^n$ in ascending order of size then we use the most efficient binary encoding by letting $I(\tau_i) = s_i$ where s_i is the i^{th} string in lexicographic order of $\{0,1\}^*$. Thus the size of τ_i and the length of s_i only differ by a constant factor. Since a tree $\tau_1 \bowtie \ldots \bowtie \tau_n$ only needs $O(\sum_{i=1}^n \hat{I}(\tau_i))$ to represent, its size and the length of $\hat{I}(\tau)$ also differ by a constant factor. Hence, the result follows. □

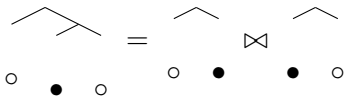
Example 7.3.1. Consider the formula:

$$\Phi \stackrel{\text{def}}{=} \forall a \exists b \exists c.\ a \ = \ b \bowtie \ \wedge \ b \ = \ \bowtie c.$$



First we factorize all tree constants in Φ:

$c_1 =$ 

$c_2 =$ 

Let $I($  $) = \epsilon$ and $I($  $) = 0$ then $c_1 = 20$, $c_2 = 02$ and the equivalent formula in $\mathcal{S}'$ is:

$$\forall a \exists b \exists c.\ a \ = \ S_{20}(b) \ \wedge \ b \ = \ P_{02}(c).$$

◁

7.4 Fragment $\langle \mathbb{T}, \sqcup, \sqcap, \dot{-}, \bowtie_\tau \rangle$

Theorem 7.2.1 shows that the first-order theory (FO) over $\mathcal{M}$ is undecidable, so of course any extension of $\mathcal{M}$—*e.g.* with $(\sqcap, \sqcup, \dot{-})$—also has an undecidable FO. However, if we restrict

the form of $\bowtie$-equations to be $\pi_1 \bowtie \tau = \pi_2$ where $\tau \in \mathbb{T}$, then the FO of $\mathcal{M}$ is decidable because the relation is tree-automatic. This type of restriction is inspired by Jain *et al.*'s concept of *semi-automatic structures* [JKS+14], in which relations are restricted so that all input arguments are fixed constants except for one argument which is a variable. As a result, certain relations become automatic, *e.g.* multiplication in unary language.

We will show $\mathcal{K} = \langle \mathbb{T}, \sqcap, \sqcup, \bar{\cdot}, \bowtie_\tau \rangle$, where $\bowtie_\tau$ denotes the family of all right-restricted forms of $\bowtie$ indexed by tree constants, is tree-automatic by constructing bottom-up tree automata that recognize the domain $\mathbb{T}$ and the relations in $\mathcal{K}$:

Theorem 7.4.1. The structure $\mathcal{K}$ is tree-automatic and thus the FO of $\mathcal{K}$ is decidable. $\triangleleft$

One major application of $\bowtie$ is to define the predicate multiplication in Chapter 3 that asserts the split/join ownership of predicates in separation logic. Here the splitting/joining occurs on the right-hand side of the $\bowtie$, *e.g.*, the binary string permission model in §3.6.3. Moreover, many functions need to divide their ownership only a finite number of times before *e.g.* calling other functions or indeed themselves recursively. This is because the program text of functions is finite. Accordingly, we believe that $\mathcal{K}$ is worthy of attention.

7.4.1 Tree automata construction

The automaton construction for domain $\mathbb{T}$ and union $\sqcup$ are provided in Example 7.1.3 and Example 7.1.4. The automaton construction for intersection $\sqcap$ is therefore similar. Hence we will discuss the automaton construction for the remaining two operators complement $\bar{\cdot}$ and bowtie $\bowtie_\tau$.

Construction of $\mathcal{A}_c$. The automaton $\mathcal{A}_c$ for complement $\bar{\cdot}$ is straightforward: we need to verify the opposite values leaf-wise between two trees*. To be precise, we have:

1. $Q = \{q\}$ is the set of state.
2. $F = \{\bullet, \circ, \mathsf{node}, \diamond\}^2$ is the ranked alphabet in which a letter has rank 2 if it contains node, otherwise it has rank 0.

*We simplify the construction by assuming that input trees are already in canonical form.

3. $Q_f = \{q\}$ contains a single accepting state.

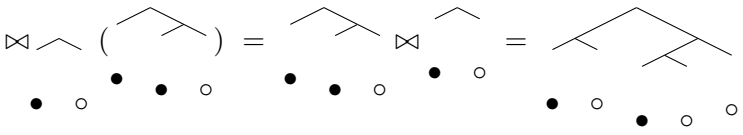
4. The transition δ contains the following rules:

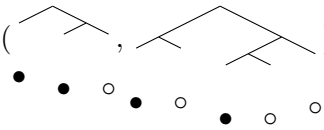
$$[\bullet, \circ] \mapsto q([\bullet, \circ]) \qquad [\circ, \bullet] \mapsto q([\circ, \bullet])$$

$$[\mathsf{node}, \mathsf{node}](q(v_1), q(v_2)) \quad \mapsto \quad q([\mathsf{node}, \mathsf{node}](v_1, v_2)).$$

Construction of $\mathcal{A}_{\bowtie_\tau}$. Next, we give the description of bottom-up tree automaton $\mathcal{A}_{\bowtie_\tau}$ that recognizes the predicate $\bowtie_\tau$. As the detail construction is significantly complicated, we will only focus in the high level description. Basically, the representation of the appended tree τ is finite and thus can be remembered by the automaton. Given a pair (τ_1, τ_2) s.t. $\tau_2 = \tau_1 \bowtie \tau$, we traverse upward and use the automaton states to record the observed subtrees in τ_2. The key point here is that we only need to remember subtrees of τ which require finite amount of memory. Once it is certain that all the subtrees in τ_2 are indeed τ, it is fairly easy for the automaton to ensure the remaining of τ_2 matches with τ_1.

Example 7.4.1. We illustrate in Figure 7.5 an accepting run over the instance:



The input is $(\tau_1, \tau_2) =$ () whose convolution is given in Figure 7.4*.

◁

Remark. As we are about to prove, the other relation ${}_\tau\bowtie$ is not tree-automatic in this representation.

Lemma 7.4.1. In the current representation of tree shares, there exists infinitely many τ such that ${}_\tau\bowtie$ is not tree-automatic. ◁

First, we recall the Pumping Lemma for tree automata:

*We simplify node to a single letter n.

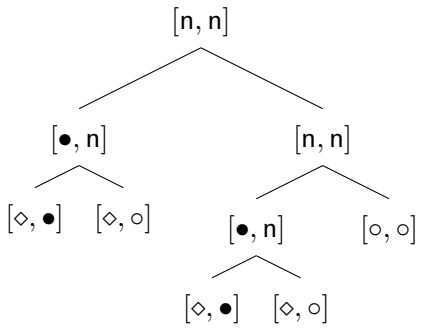


Figure 7.4: Convolution of (τ_1, τ_2) in Example 7.4.1.

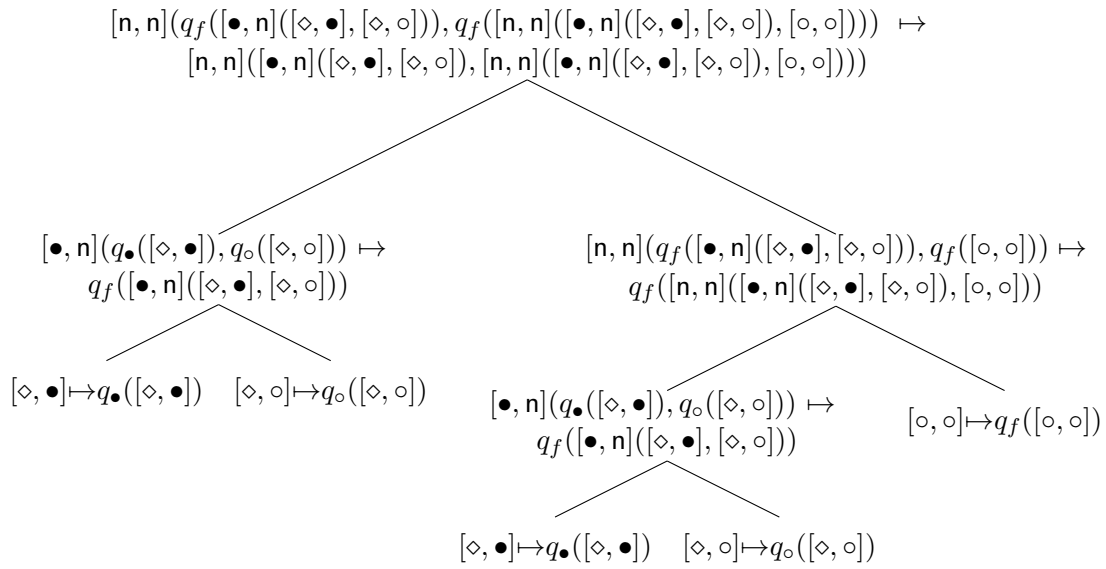
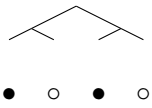
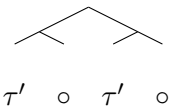


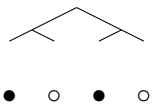
Figure 7.5: An accepting run over tree automaton for predicate $\bowtie_\tau$ in Example 7.4.1.

Definition 7.4.1 (Term, context and substitution [CDG+07])**.** Let $\mathcal{A} = (\mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta)$ be a tree automaton and V the set of variables. We define $T(\mathcal{F}, V)$ the set of all tree terms derived from $\mathcal{F} \cup V$ and $T(\mathcal{F}, \emptyset)$ is the set of ground terms. A term t is linear if each variable appears at most once in t. A context C is a linear term of $T(\mathcal{F}, V)$ and $\mathcal{C}(\mathcal{F})$ denotes the set of all contexts with single variable. A context is trivial if it is reduced to a variable. Let $C[t]$ be the substitution of $C \in \mathcal{C}(\mathcal{F})$ by replacing the variable in C with the term t. We define $C^0[t] = v$ where v is the variable in C, $C^1[t] = C[t]$ and $C^{n+1}[t] = C[C^n[t]]$. ◁

Proposition 7.4.1 (Pumping Lemma for Tree Automata [CDG+07])**.** Let $\mathcal{L}$ be the set of all ground terms recognizable by a tree automaton. There is a constant $k \in \mathbb{N}^+$ satisfying the following condition: for all ground term $t \in \mathcal{L}$ and $|t| > k$, there exists a context $C \in \mathcal{C}(\mathcal{F})$, a nontrivial context $C' \in \mathcal{C}(\mathcal{F})$ and a ground term t' such that $t = C[C'[t']]$ and $\forall n \in \mathbb{N}.\ C[C'^n[t']] \in \mathcal{L}$. ◁

Proof of Lemma 7.4.1. Let ${}_\tau\bowtie$ where $\tau =$ 

. For an input tree $\tau' \in \mathbb{T}^+$, the result tree is 

in which the left and right subtree are identical. If ${}_\tau\bowtie$ is automatic then it satisfies the Pumping Lemma. However, the Pumping Lemma only allows us to pump either the left or the right subtree and thus they will be different after pumping, which is a contradiction. Now consider the following sequence: $\tau_1 =$ 

, $\tau_{n+1} =$ 

then each of the ${}_{\tau_i}\bowtie$ is not automatic. □

7.4.2 Non-elementary lower bound

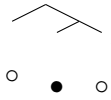
In §7.4.1, we showed that first-order theory of $\mathcal{K} = \langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot}, \bowtie_\tau \rangle$ is decidable by constructing a tree automatic representation for $\mathcal{K}$. As the complexity of tree automatic structures are non-elementary in general, no useful upper bound for $\mathcal{K}$ was derived from that decidability result. In this subsection, we will show that the complexity of $\mathsf{Th}(\mathcal{K})$ is non-elementary by reducing the binary tree structure with prefix relation [CH90] into $\mathcal{K}$ which is well-known to be non-elementary:

Theorem 7.4.2. The complexity of $\mathsf{Th}(\mathcal{K})$ is non-elementary. ◁

We recall the definition and complexity result of binary trees with prefix relation:

Proposition 7.4.2 ([CH90, Sto74])**.** Let $\mathcal{B} = \langle \{0,1\}^*, S_0, S_1, \preceq \rangle$ be the binary tree structure in which $\{0,1\}^*$ is the set of binary strings, S_i is the successor function s.t. $S_i(s) = s \cdot i$, and $\preceq$ is the binary prefix relation s.t. $x \preceq y$ iff there exists z satisfies $x \cdot z = y$.

Then the first-order theory of $\mathcal{B}$ is non-elementary. ◁

We proceed to construct a reduction from $\mathcal{B}$ to $\mathcal{K}$. For **convenience**, we use the symbol $\mathcal{L}$ to represent the left tree  and $\mathcal{R}$ for right tree  . In detail, we map the set of strings $\{0,1\}^*$ into the set of *unary trees* $\mathcal{U}(\mathbb{T})$that have exactly one black leaf, *e.g.*, $\bullet$ and  :

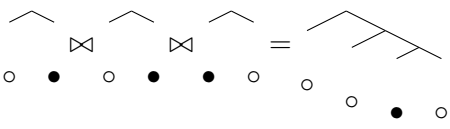
Lemma 7.4.2. Let $g : \langle \{0,1\}^*, S_0, S_1, \preceq \rangle \mapsto \langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot}, \bowtie_\tau \rangle$ s.t.:

$$g(\epsilon) \stackrel{\text{def}}{=} \bullet \qquad g(0) \stackrel{\text{def}}{=} \mathcal{L} \qquad g(1) \stackrel{\text{def}}{=} \mathcal{R}$$

$$g(b_1 \dots b_n) \stackrel{\text{def}}{=} g(b_1) \bowtie \dots \bowtie g(b_n), b_i \in \{0,1\}$$

$$g(S_0) \stackrel{\text{def}}{=} \lambda s.\ \bowtie_{\mathcal{L}} (g(s)) \qquad g(S_1) \stackrel{\text{def}}{=} \lambda s.\ \bowtie_{\mathcal{R}} (g(s)) \qquad g(x \preceq y) \stackrel{\text{def}}{=} g(y) \sqsubseteq g(x).$$

Then g is a bijection from $\{0,1\}^*$ to $\mathcal{U}(\mathbb{T})$, the set of unary trees. Furthermore, we have $x \preceq y$ iff $g(y) \sqsubseteq g(x)$*. ◁

Proof. The proof that g is bijective is done by proving g is both injective and surjective. Intuitively, the string s represents the path to the single black leaf in the tree $g(s)$ in which the letter 0 means 'left' and 1 means 'right'. For example, the tree $g(110) = \mathcal{R} \bowtie \mathcal{R} \bowtie \mathcal{L} =$  represents the path 'right, right, left'.

In detail, assume $g(a_1 \dots a_k) = g(b_1 \dots b_h)$ where $a_i, b_j \in \{0,1\}$. It follows that

$$g(a_1) \bowtie \dots \bowtie g(a_k) = g(b_1) \bowtie \dots \bowtie g(b_h).$$

*Recall from §6.1.3 that $\tau_1 \sqsubset \tau_2 \stackrel{\text{def}}{=} \tau_1 \sqcap \tau_2 = \tau_1 \wedge \tau_1 \sqcap \tau_2 \neq \tau_2$.

By uniqueness representation of tree shares (Lemma 7.2.6) and the fact that $g(a_i), g(b_j) \in \{\mathcal{L}, \mathcal{R}\}$ are tree primes, we conclude that $h = k$ and $g(a_i) = g(b_i)$ or $a_i = b_i$. Hence g is injective. To see why g is surjective, consider $\tau \in \mathcal{U}(\mathbb{T})$. If $\tau \in \{\mathcal{L}, \mathcal{R}\}$ then we are done. Otherwise, we can find $\tau_1 \in \mathcal{U}(\mathbb{T})$ and $\tau_2 \in \{\mathcal{L}, \mathcal{R}\}$ s.t. $\tau = \tau_1 \bowtie \tau_2$. By inductive argument, there is a binary string s_1 s.t. $g(s_1) = \tau_1$ and $s_2 \in \{0, 1\}$ s.t. $g(s_2) = \tau_2$. Thus

$$g(s_1 \cdot s_2) = g(s_1) \bowtie g(s_2) = \tau_1 \bowtie \tau_2 = \tau.$$

For the last claim, let $\tau_1, \tau_2 \in \mathcal{U}(\mathbb{T})$ be unary trees then we have $\tau_1 \sqsubseteq \tau_2$ iff there exists a unary tree $\tau_3 \in \mathcal{U}(\mathbb{T})$ s.t. $\tau_2 \bowtie \tau_3 = \tau_1$ (as $a \bowtie b$ is a subtree of a). Thus $x \preceq y$ iff $\exists z.\ xz = y$ iff $\exists z.\ g(x) \bowtie g(z) = g(y)$ iff $g(y) \sqsubseteq g(x)$. □

One essential criterion of the reduction is to express the type of $\mathcal{U}(\mathbb{T})$ using the signature from $\mathcal{K}$. We show that $\mathcal{U}(\mathbb{T})$ is expressible using $\bowtie_\tau$ and $\sqsubset$. Formally:

Lemma 7.4.3. The type $\mathcal{U}(\mathbb{T})$ is expressible in $\mathcal{K}$ using a $\forall$-formula:

$$\tau \in \mathcal{U}(\mathbb{T}) \quad \text{iff} \quad \tau \neq \circ \ \wedge\ \big(\forall \tau'.\ \bowtie_{\mathcal{L}} (\tau') \sqsubset \tau \ \leftrightarrow\ \bowtie_{\mathcal{R}} (\tau') \sqsubset \tau\big).$$

where $\tau_1 \sqsubset \tau_2 \stackrel{\text{def}}{=} (\tau_1 \sqcap \tau_2 = \tau_1) \wedge (\tau_1 \sqcap \tau_2 \neq \tau_2)$. ◁

Proof. We prove $\Rightarrow$ by induction on the height of τ. The base case $|\tau| = 0$, *e.g.* $\tau = \bullet$, is simple to check. Thus it remains to prove the case $|\tau| = n + 1$.

Let $\tau_l = \ \bowtie_{\mathcal{L}} (\tau') = \tau' \bowtie \mathcal{L}$ and $\tau_r = \ \bowtie_{\mathcal{R}} (\tau') = \tau' \bowtie \mathcal{R}$ where τ' is arbitrary. By symmetric argument, it suffices to prove $\tau_l \sqsubset \tau$ implies $\tau_r \sqsubset \tau$. As $|\tau| > 0$, either $\tau =$  or $\tau =$ 

for some unary trees τ_1. W.l.o.g. let $\tau =$  $= \mathcal{L} \bowtie \tau_1$. From $\tau' \bowtie \mathcal{L} \sqsubset \tau$, we infer

$\tau' =$  $= \mathcal{L} \bowtie \tau_1'$ for some τ_1'.

Thus $\tau_1 = \mathcal{L} \bowtie \tau_1' \bowtie \mathcal{L} \ \sqsubset\ \mathcal{L} \bowtie \tau_1 = \tau$ and therefore $\tau_1' \bowtie \mathcal{L} \sqsubset \tau_1$. As τ_1 is unary and

$|\tau_1| < |\tau|$, the induction hypothesis gives us $\tau_1' \bowtie \mathcal{R} \sqsubset \tau_1$. Thus:

$$\tau_r \;=\; \tau' \bowtie \mathcal{R} \;=\; \mathcal{L} \bowtie \tau_1' \bowtie \mathcal{R} \;\sqsubset\; \mathcal{L} \bowtie \tau_1 \;=\; \tau.$$

For $\Leftarrow$, assume $\tau \notin \mathcal{U}(\mathbb{T})$. As $\tau \neq \circ$, it follows that τ contains at least two black leaves in its representation. Consequently, we can find $\tau_1 \in \mathcal{U}(\mathbb{T})$ s.t. $\tau_1 \sqsubset \tau$ and for any $\tau_2 \in \mathcal{U}(\mathbb{T})$, if $\tau_1 \sqsubset \tau_2$ then $\tau_2 \not\sqsubseteq \tau$. By properties of unary trees, we can represent τ_1 as either $\tau_1' \bowtie \mathcal{L}$ or $\tau_1' \bowtie \mathcal{R}$ for some $\tau_1' \in \mathcal{U}(\mathbb{T})$. Consequently, we have $\tau_1 \sqsubset \tau_1'$ and thus $\tau_1' \not\sqsubseteq \tau$. The premise gives us $\tau_1' \bowtie \mathcal{L} \sqsubset \tau$ and $\tau_1' \bowtie \mathcal{R} \sqsubset \tau$. Hence $\tau_1' = \tau_1' \bowtie (\mathcal{L} \sqcup \mathcal{R}) \sqsubseteq \tau$ which is a contradiction. □

Proof of Theorem 7.4.2. Our technique is similar to the one in [Grä90] in which we interpret formulas of $\mathcal{B}$ using the signature of $\mathcal{K}$. The interpretation of constants and symbols is mentioned in Lemma 7.4.2. Next we replace sub-formula $\exists x.\ \Phi$ with $\exists x.\ x \in \mathcal{U}(\mathbb{T}) \wedge \Phi$ and $\forall x.\ \Phi$ with $\forall x.\ x \in \mathcal{U}(\mathbb{T}) \rightarrow \Phi$. Thus by Lemmas 7.4.2 and 7.4.3, a string formula in $\mathcal{B}$ can be transformed into an equivalent tree formula in $\mathcal{K}$. Hence the first-order complexity of $\mathcal{K}$ is bounded below by the first-order complexity of $\mathcal{B}$. By Prop. 7.4.2, the first-order complexity of $\mathcal{K}$ is non-elementary. □

7.5 Conclusion

We have developed a more precise understanding of the complexity of the tree share model. We have provided the first serious look at the complexity of the tree multiplication $\bowtie$ operator and by way of an isomorphism to word equations prove that the existential theory is in PSPACE and NP-hard while the general first-order theory is undecidable. To recover decidability, we have found that by restricting multiplication to be by a constant (on both the left ${}_\tau\!\bowtie$ and right $\bowtie_\tau$ sides) we obtain a subtheory $\mathcal{R}$ whose first-order theory is $\mathsf{STA}(*, 2^{O(n)}, n)$-complete. Recall from Corollary 6.3.1 that as a Boolean algebra, their first-order theory is $\mathsf{STA}(*, 2^{n^{O(1)}}, n)$-complete, even when arbitrary constants are allowed in the formulae. Accordingly, we have two theories whose first-order theory is elementarily decidable. Unfortunately, their combined theory is at best non-elementary, even if we only

allow multiplication by a constant on the right side $\bowtie\!\lhd_\tau$. Despite these remaining questions, our understanding of the structure has improved meaningfully, allowing us to contemplate using it inside practical verification tools. We have already incorporated tree shares and their Boolean structure into the HIP/SLEEK verification toolchain [NDQC07, LNHC17] and are actively exploring how to incorporate their multiplicative structure as well. With our improved understanding of the combined structure $\mathcal{K}$ we believe that we can take advantage of powerful heuristics for automata such as antichain and simulation [ACH+10].

Chapter 8

Conclusion and Future work

Alice: *Would you tell me, please, which way I ought to go from here?*
The Cheshire Cat: *That depends a good deal on where you want to get to.*
Alice: *I don't much care where.*
The Cheshire Cat: *Then it doesn't much matter which way you go.*
Alice: *...So long as I get somewhere.*
The Cheshire Cat: *Oh, you're sure to do that, if only you walk long enough.*

Alice in Wonderland.

In this thesis, we conducted a comprehensive study of the tree share structure proposed by Dockins *et al.* [DHA09]. Our contributions can be classified into three categories: applications, systems and theory:

1. For applications, we used tree shares to reason about fractional permissions in program verification. In §4, we showed how to embed tree shares into assertion language as well as extract their constraints to solve independently. We also made another major contribution in §3 by developing a general modal logic framework that allows permission reasoning over arbitrary predicates. Our approach can handle sophisticated verification tasks such as bi-abduction inference, induction over inductive predicates and precision reasoning.

2. For systems, we integrated tree shares into the HIP/SLEEK tool [NDQC07] to verify practical programs. Our first contribution in §4 is the two complete decision procedures

SAT for satisfiability and IMP for entailment checking over tree share constraints. Our second contribution in §5 is the two certified procedures GSAT and GIMP that can additionally handle disequations of the form $\neg(a \oplus b = c)$. Furthermore, our new procedures can be run entirely inside the Coq native environment with reasonable performance.

3. For theory, we established several decidability and complexity results over the tree share structure. In §6, we proved that the Boolean-like structure $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot} \rangle$ is first-order decidable and its complexity is $\mathsf{STA}(*, 2^{n^{O(1)}}, n)$-complete. In §7, we showed that the bowtie structure $\langle \mathbb{T}, \bowtie \rangle$ is almost isomorphic to string structure with concatenation $\langle \Sigma, \cdot \rangle$. As a result, its existential theory is NP-hard and in PSPACE whereas its first-order theory is undecidable. To recover decidability for bowtie, we need to restrict the form of bowtie in the formulae. As a result, we were able to prove two first-order decidable fragments of bowtie. The first fragment is $\langle \mathbb{T}, _\tau \bowtie, \bowtie_\tau \rangle$ in which we require at least one operant must be constant. This fragment has first-order complexity $\mathsf{STA}(*, 2^{O(n)}, n)$-complete. The second fragment is the combined structure $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot}, \bowtie_\tau \rangle$ that includes all Boolean-like operators together with restricted bowtie whose second operant is constant. Interestingly, although its first-order theory is decidable, we showed that the complexity is non-elementary.

Lastly, we are left with several directions to explore in the future:

1. For applications, we would like to use tree shares to model different permission types. At the moment, we mainly use the structure for two fundamental permissions: read and write. In practice, it is convenient to have other permission types for deallocation, lock acquire (the permission to acquire a lock) or "counter-deallocation" (the permission that prevents other threads from deallocating a given resource, even though it may be weaker than a read permission). As a result, we believe this problem is worth investigating in the future.

2. For systems, we would like to integrate tree shares as a concrete model for scaling separation algebra into verification tools for practical purposes. This requires a specialized decision procedure to handle tree share constraints in $\langle \mathbb{T}, \oplus, \bowtie \rangle$. One piece

of bad news is that the first-order theory of $\langle \mathbb{T}, \oplus, \bowtie \rangle$ is undecidable and thus reasoning about a complete procedure is impossible. There are two potential solutions for this problem: we can either develop a semi-decision procedure, or carefully modify the logic inference so that the constraints are in the decidable fragment $\langle \mathbb{T}, \oplus, \bowtie_\tau \rangle$. Both approaches are worth considering as long as the procedure can handle a reasonable range of common tree share constraints.

3. For theory, we are interested in answering several unknown decidability and complexity queries about tree shares. For instance, it is not yet known whether the structure $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot}, {}_\tau\bowtie, \bowtie_\tau \rangle$ is first-order decidable or any structure that contains both left-bowtie ${}_\tau\bowtie$ and other Boolean-like operators.

References

[AB07] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step cminor. In *TPHOLs*, pages 5–21, 2007.

[ACH+10] Parosh Aziz Abdulla, Yu-Fang Chen, Lukás Holík, Richard Mayr, and Tomás Vojnar. When simulation meets antichains. In *TACAS*, pages 158–174, 2010.

[ADH09] Andrew W. Appel, Robert Dockins, and Aquinas Hobor. Mechanized semantic library, 2009.

[ADH+14] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[AP89] H. Abdulrab and J.P Pechuchet. Solving word equations. In *Journal of Symbolic Computation*, pages 499–521, 1989.

[App11a] Andrew W. Appel. Efficient verified red-black trees, 2011.

[App11b] Andrew W. Appel. Verified software toolchain. In *ESOP*, 2011.

[BCO06] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2006.

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter O'H, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.

[BDD+11] Matko Botinčan, Dino Distefano, Mike Dodds, Radu Grigore, Daiva Naudžiūnienė, and Matthew J. Parkinson. coreStar: the core of jStar. In K. Rustan M. Leino and Michał Moskal, editors, *BOOGIE 2011*, pages 65–77, 2011.

[BG04] A. Blumensath and E. Grade. Finite presentations of infinite structures: automata and interpretations. In *Theory of Computer Systems*, pages 641–674, 2004.

[BGK17] James Brotherston, Nikos Gorogiannis, and Max Kanovich. Biabduction (and related problems) in array separation logic. In *CADE*, 2017.

[Blu99] A. Blumensath. *Automatic Structures*. PhD thesis, RWTH Aachen, 1999.

[BMSS14] John Tang Boyland, Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Constraint semantics for abstract read permissions. In *FTfJP*, pages 2:1–2:6, 2014.

[Boy03] John Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72, 2003.

[Boy10] John Tang Boyland. Semantics of fractional permissions with nesting. *ACM Trans. Program. Lang. Syst.*, 32(6):22:1–22:33, August 2010.

[Bro06] Stephen Brookes. Variables as resource for shared-memory programs: Semantics and soundness. *Electron. Notes Theor. Comput. Sci.*, 158:123–150, 2006.

[Bro07a] Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, pages 227–270, 2007.

[Bro07b] James Brotherston. Formalised inductive reasoning in the logic of bunched implications. In *SAS*, pages 87–103, 2007.

[BS90] J Richard Büchi and Steven Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. pages 671–683, 1990.

[CDD+15] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NFM*, pages 3–11, 2015.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.

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

[CDOY09] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.

[CDV09] Cristiano Calcagno, Dino Distefano, and Viktor Vafeiadis. Bi-abductive resource invariant synthesis. In *APLAS*, pages 259–274, 2009.

[CH90] Kevin J. Compton and C. Ward Henson. A uniform method for proving lower bounds on the computational complexity of logical theories. *Annals of Pure and Applied Logic*, 48(1):1–79, 1990.

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers;. *Commun. ACM*, 14(10):667–668, October 1971.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *Journal of Symbolic Logic*, 1(2):73–74, 1936.

[CKS81] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, January 1981.

[CLQ17] Wei Ngan Chin, Ton Chanh Le, and Shengchao Qin. Automated verification of countdownlatch, 2017.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS*, pages 366–378, 2007.

[CP17] Arthur Charguéraud and François Pottier. Temporary read-only permissions for separation logic. In *ESOP*, pages 260–286, 2017.

[CPV07] Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS*, pages 233–248, 2007.

[Dev] http://www.comp.nus.edu.sg/~lxbach/tools/share_infer/.

[DHA09] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, pages 161–177, 2009.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[DPJ08] Dino Distefano and Matthew J. Parkinson J. jstar: Towards practical verification for Java. In *OOPSLA*, pages 213–226, 2008.

[dRPDYG14] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In *ECOOP*, pages 207–231, 2014.

[DYBG+13] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *POPL*, pages 287–300, 2013.

[DYdAB17] Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. Caper: Automatic verification for fine-grained concurrency. In *ESOP*, pages 420–447, 2017.

[DYDG+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.

[ES03] N. Een and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–508, 2003.

[FLLV15] Jan Fiedor, Zdeněk Letko, João Lourenço, and Tomáš Vojnar. Dynamic validation of contracts in concurrent code. In *EUROCAST*, pages 555–564, 2015.

[Flo67] Robert W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Proceedings of a Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–31, Providence, 1967. American Mathematical Society.

[GBC11] Alexey Gotsman, Josh Berdine, and Byron Cook. Precision and the conjunction rule in concurrent separation logic. *Electron. Notes Theor. Comput. Sci.*, pages 171–190, 2011.

[Ghe12] Cristian A. Gherghina. *Efficiently Verifying Programs with Rich Control Flows*. PhD thesis, National University of Singapore, 2012.

[Göd29] K Gödel. *The first proof of the completeness theorem*. PhD thesis, University Of Vienna, 1929.

[Grä90] Erich Grädel. Simple interpretations among complicated theories. *Information Processing Letters*, 35(5):235–238, 1990.

[GVA07] Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *PLDI*, pages 256–265, 2007.

[Hal74] Paul R. Halmos. *Lectures on Boolean Algebras*. Springer, 1974.

[HAZ08] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.

[HG11] Hobor and Cristian Gherghina. Barriers in concurrent separation logic. In *ESOP*, pages 276–296, 2011.

[HG12] Aquinas Hobor and Cristian Gherghina. Barriers in concurrent separation logic: Now with tool support! *Logical Methods in Computer Science*, 8(2), 2012.

[HHH08] Christian Haack, Marieke Huisman, and Clément Hurlin. Reasoning about Java's reentrant locks. In *APLAS*, pages 171–187, 2008.

[HLMS11] Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. Fractional permissions without the fractions. In *FTfJP*, 2011.

[HM15] Marieke Huisman and Wojciech Mostowski. A symbolic approach to permission accounting for concurrent reasoning. In *ISPDC*, pages 165–174, 2015.

[HMP17] Jochen Hoenicke, Rupak Majumdar, and Andreas Podelski. Thread modularity at many levels: A pearl in compositional verification. In *POPL*, pages 473–485, 2017.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[Hob08] Aquinas Hobor. *Oracle Semantics*. PhD thesis, Princeton University, 2008.

[HV13] Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. In *POPL*, pages 523–536, 2013.

[HW06] J. Hayman and G. Winskel. Independence and concurrent separation logic. In *LICS*, pages 147–156, 2006.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001.

[Jaf90] Joxan Jaffar. Minimal and complete word unification. *J. ACM*, 37(1):47–85, 1990.

[JK03] Erik J. Johnson and Aaron R. Kunze. *Ixp2400-2800 Programming: The Complete Microengine Coding Guide*. Intel Press, 2003.

[JKS+14] Sanjay Jain, Bakhadyr Khoussainov, Frank Stephan, Dan Teng, and Siyuan Zou. Semiautomatic structures. In *CSR*, pages 204–217, 2014.

[Jon83] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP*, pages 321–332, 1983.

[JP11] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282, 2011.

[JSP10] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In *APLAS*, 2010.

[JSS+15] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650, 2015.

[KLVU10] Bohuslav Křena, Zdeněk Letko, Tomáš Vojnar, and Shmuel Ur. A platform for search-based testing of concurrent software. In *PADTAD*, pages 48–58, 2010.

[KM07] Bakhadyr Khoussainov and Mia Minnes. Three lectures on automatic structures. In *Logic Colloquium*, pages 132–176, 2007.

[Koz80] Dexter Kozen. Complexity of boolean algebras. In *Theoretical Computer Science*, pages 221–247, 1980.

[Koz06] Dexter C. Kozen. *Theory of Computation*. Springer, 2006.

[KP96] Antoni Koscielski and Leszek Pacholski. Complexity of Makanin's algorithm. *J. ACM*, 43(4):670–684, 1996.

[Kus06] D. Kuske. Theories of orders on the set of words. In *Theoretical Informatics and Applications*, pages 53–74, 2006.

[LCT15] Duy-Khanh Le, Wei-Ngan Chin, and Yong Meng Teo. Threads as resource for concurrency verification. In *PEPM*, pages 73–84, 2015.

[LGH12] Xuan-Bach Le, Cristian Gherghina, and Aquinas Hobor. Decision procedures over sophisticated fractional permissions. In *APLAS*, 2012.

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

[LH17] Xuan-Bach Le and Aquinas Hobor. Logical reasoning over disjoint fractional permissions. 2017. Under submission.

[LHL16] Xuan-Bach Le, Aquinas Hobor, and Anthony W. Lin. Decidability and complexity of tree shares formulas. In *FSTTCS*, 2016.

[LHL17] Xuan-Bach Le, Aquinas Hobor, and Anthony W. Lin. Complexity analysis of tree share operations. 2017. Under submission.

[LM09] K. Rustan Leino and Peter Müller. A basis for verifying multi-threaded programs. In *ESOP*, pages 378–393, 2009.

[LNHC17] Xuan-Bach Le, Thanh Toan Nguyen, Aquinas Hobor, and Wei Ngan Chin. A certified decision procedure for tree shares. In *ICFEM*, 2017.

[Mak77] G. S Makanin. The problem of solvability of equations in a free semigroup. In *Mat. Sbornik*, pages 147–236, 1977.

[Mak83] G. S Makanin. Equations in a free group. In *Izvestiya AN SSSR*, pages 1199–1273, 1982-1983.

[Mak85] G.S Makanin. Decidability of the universal and positive theories of a free group. In *Izvestiya AN SSSR*, pages 735–749, 1984-1985.

[Mar82] S. Marchenkov. Unsolvability of positive ∀∃-theory of free semi-group. In *Sibirsky mathmatichesky jurnal*, pages 196–198, 1982.

[MHWL12] Wenrui Meng, Fei He, Bow-Yaw Wang, and Qiang Liu. Thread-modular model checking with iterative refinement. In *NFM*, pages 237–251, 2012.

[MO96] Kim Marriott and Martin Odersky. Negative boolean constraints. In *Theoretical Computer Science*, pages 365–380, 1996.

[MSS16] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, pages 41–62, 2016.

[NDQC07] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, pages 251–266, 2007.

[NLWSD14] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, 2014.

[OHe07] Peter W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, April 2007.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, 2001.

[Pap03] Christos H. Papadimitriou. *Computational Complexity.* John Wiley and Sons Ltd., 2003.

[Par05] Matthew Parkinson. *Local Reasoning for Java.* PhD thesis, University of Cambridge, 2005.

[PBC06] Matthew Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in Hoare logics. In *LICS*, pages 137–146, 2006.

[PH09] Steven Givant Paul Halmos. *Introduction to Boolean Algebras.* Springer, 2009.

[Pip96] Nicholas Pippenger. Pure versus impure LISP. In *POPL*, pages 104–109, 1996.

[Pla04] W. Plandowski. Satisfiability of word equations with constants is in PSPACE. In *Journal of the Association for Computing Machinery*, pages 483–496, 2004.

[Pla06] W. Plandowski. An efficient algorithm for solving word equations. In *STOC*, pages 467–476, 2006.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[Ric53] H.G Rice. Classes of recursively enumerable sets and their decision problems. In *Transactions of the American Mathematical Society*, pages 358–366, 1953.

[RV03] Tatiana Rybina and Andrei Voronkov. Upper bounds for a theory of queues. In *ICALP*, pages 714–724, 2003.

[SB14] Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *ETAPS*, pages 149–168, 2014.

[SNB15] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87, 2015.

[Sol] http://www.comp.nus.edu.sg/~lxbach/certtool/.

[Sri13] Shashi Mohan Srivastava. *A Course on Mathematical Logic*. 2nd edition, 2013.

[Sto74] L. Stockmeyer. *The complexity of decision problems in automata theory and logic*. PhD thesis, M.I.T., 1974.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[TDB13] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390, 2013.

[Vaf07] Vafeiadis. *Fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.

[Vaf09] Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI*, pages 335–348, 2009.

[Vaf11] Viktor Vafeiadis. Concurrent separation logic and operational semantics. *Electron. Notes Theor. Comput. Sci.*, pages 335–351, 2011.

[Vil11] Jules Villard. *Heaps and Hops*. PhD thesis, Laboratoire Spécification et Vérification, École Normale Supérieure de Cachan, France, 2011.

[VLC10] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Tracking heaps that hop with Heap-Hop. In *TACAS*, pages 275–279, 2010.

[VP07] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.

[Whi61] John Eldon Whitesitt. *Boolean Algebra and Its Applications*. Addison-Wesley, 1961.

[YLB+08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008.

Appendix A

Additional proofs for Chapter 3

Here §A.1 we will show the necessary conditions for the underlying fractional permission model in order to satisfy the scaling rules. In §A.2, we prove why we cannot have disjointness axiom together with two distributivity axioms for fractional permissions as well as why we cannot have the axiom for inverse element of $\otimes$.

A.1 Necessary conditions for scaling rules

Using fractional heap semantics in §3.5.1, we can derive the following scaling rules from Fig. 3.2:

Theorem A.1.1. The following rules automatically hold in the fractional heap model:

1. DotPure: $\pi \cdot (|P| \wedge \mathsf{emp}) \dashv\vdash |P| \wedge \mathsf{emp}$.
2. DotPos: $P \vdash Q \;\Rightarrow\; \pi \cdot P \vdash \pi \cdot Q$.
3. DotDisj: $\pi \cdot (P \vee Q) \dashv\vdash \pi \cdot P \vee \pi \cdot Q$.
4. DotConj $\vdash$: $\pi \cdot (P \wedge Q) \vdash \pi \cdot P \wedge \pi \cdot Q$.
5. DotUniv: $\pi \cdot (\forall x : \tau.\ P(x)) \dashv\vdash \forall x : \tau.\ \pi \cdot P(x)$ for $\tau \neq \emptyset$.
6. DotExis: $\pi \cdot (\exists x : \tau.\ P(x)) \dashv\vdash \exists x : \tau.\ \pi \cdot P(x)$.

◁

Proof. For DotPure, let h_0 be the empty heap, *i.e.* $h_0 \models \mathsf{emp}$ then $h_0 = \pi \otimes h_0$ for all π and thus the result follows. For DotPos, let $h \models \pi \cdot P$ then there exists h' s.t. $h = \pi \otimes h'$

and $h' \models P$. By the premise $P \vdash Q$, we have $h' \models Q$ and thus $h \models \pi \cdot Q$.

For DotDisj $\Rightarrow$, let $h \models \pi \cdot (P \vee Q)$ then there exists h' s.t. $h = \pi \otimes h'$ and $h' \models P \vee Q$. W.l.o.g. assume $h' \models P$ then $h \models \pi \cdot P$ by definition and thus $h \models \pi \cdot P \vee \pi \cdot Q$. The other direction is similar. The proof for DotConj $\vdash$ is also similar to the disjunction case.

Both direction in DotExis and the left-to-right of DotUniv are simple. For the other direction of DotUniv, let $h \models \forall x : \tau.\ \pi \cdot P(x)$ then the condition $\tau \neq \emptyset$ implies the existence of h' s.t. $h = \pi \otimes h'$. The rest of the proof is straightforward. □

Other scaling rules need certain properties from the permission model. For example, left-to-right direction of DotPlus needs the right distributivity of $\otimes$ over $\oplus$. While the need for precision condition in DotPlus is standard as mentioned in [Boy10], it is quite complicated to explain the reason we com up with uniformity condition for DotStar. Briefly speaking, without that condition, it is impossible to construct a reasonable model to justify the soundness of our rules. In particular, we will show in §A.2 that without the uniformity condition then permission models only have at most two elements. For now, let us replace DotStar with Dotstar$'$ in which the uniformity precondition is omitted. Then we can extract the following properties from other scaling rules:

Theorem A.1.2. In the fractional heap model, we need the following properties in the fractional model $\mathcal{M} = \langle \mathcal{U}, \oplus, \otimes, \mathcal{F}, \mathcal{E} \rangle$ for scaling rules in Fig. 3.2 over positive permissions:

1. Identity of $\otimes$ ($I_\otimes$): $\pi \otimes \mathcal{F} = \mathcal{F} \otimes \pi = \pi$.
2. Left distributivity ($\overleftarrow{D}$): $\pi \otimes (\pi_1 \oplus \pi_2) = (\pi \otimes \pi_1) \oplus (\pi \otimes \pi_2)$.
3. Right distributivity ($\overrightarrow{D}$): $(\pi_1 \oplus \pi_2) \otimes \pi = (\pi_1 \otimes \pi) \oplus (\pi_2 \otimes \pi)$.
4. Associativity of $\otimes$ ($A_\otimes$): $\pi_1 \otimes (\pi_2 \otimes \pi_3) = (\pi_1 \otimes \pi_2) \otimes \pi_3$.
5. Right cancellativity of $\otimes$ ($\overrightarrow{E_\otimes}$): $(\pi \otimes \pi_1 = \pi \otimes \pi_2) \Rightarrow \pi_1 = \pi_2$.

◁

We divide the proof into several lemmas A.1.1-A.1.6 that establish the connection between scaling rules in Fig. 3.2 and fractional permission properties. Also, we assume that all permissions π are positive, *i.e.*, $\pi \neq \circ$.

The right-to-left of DotConj needs $\oplus$ to be right-cancellative as iff condition:

Lemma A.1.1. The following properties are equivalent (π is fixed while other variables are universal):

1. Predicate (DotConj $\dashv$): $\pi \cdot P \wedge \pi \cdot Q \vdash \pi \cdot (P \wedge Q)$.
2. Mapping: $\pi \cdot x \overset{\pi_1}{\longmapsto} v \wedge \pi \cdot x \overset{\pi_2}{\longmapsto} v \vdash \pi \cdot (x \overset{\pi_1}{\longmapsto} v \wedge x \overset{\pi_2}{\longmapsto} v)$.
3. Heap: $\pi \otimes h_1 = \pi \otimes h_2 \Rightarrow h_1 = h_2$.
4. Permission ($\overrightarrow{E_\otimes}$): $(\pi \otimes \pi_1 = \pi \otimes \pi_2) \Rightarrow \pi_1 = \pi_2$.

◁

Proof. We will show that (3) $\Leftrightarrow$ (4) $\Leftrightarrow$ (2) and (1) $\Leftrightarrow$ (4).

For (4) $\Rightarrow$ (3), let $a \in \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2)$ s.t. $h_1(a) = (v_1, \pi_1)$ and $h_2(a) = (v_2, \pi_2)$. We will show that $h_1(a) = h_2(a)$ or $v_1 = v_2$ and $\pi_1 = \pi_2$. By definition, we have $\pi \otimes h_1(a) = (v_1, \pi \otimes \pi_1)$ and $\pi \otimes h_2(a) = (v_2, \pi \otimes \pi_2)$ and thus the premise gives us $v_1 = v_2$ and $\pi \otimes \pi_1 = \pi \otimes \pi_2$. By (5), we arrive $\pi_1 = \pi_2$. For (3) $\Rightarrow$ (4), assume $\pi \otimes \pi_1 = \pi \otimes \pi_2$, we pick h_1, h_2 s.t. $\mathsf{dom}(h_i) = \{a\}$ and $h_i(a) = (v, \pi_i)$. Then $\pi \otimes h_1 = \pi \otimes h_2$ and thus $h_1 = h_2$ or $\pi_1 = \pi_2$. The case (3) $\Leftrightarrow$ (4) is similar.

For (4) $\Rightarrow$ (1), let $h, \rho \models \pi \cdot P \wedge \pi \cdot Q$ then $h, \rho \models \pi \cdot P$ and $h, \rho \models \pi \cdot Q$. Thus there exist h_1, h_2 s.t. $h = \pi \otimes h_1 = \pi \otimes h_2$, $h_1, \rho \models P$ and $h_2, \rho \models Q$. By (5), we can prove (4) and thus $h_1 = h_2$. As a result, $h_1, \rho \models P \wedge Q$ and thus $h, \rho \models \pi \cdot (P \wedge Q)$. For (1) $\Rightarrow$ (4), assume $\pi \otimes \pi_1 = \pi \otimes \pi_2$, we choose $P = x \overset{\pi_1}{\longmapsto} v$ and $Q = x \overset{\pi_2}{\longmapsto} v$ then (5) becomes (3). As (3) implies (5), the result follows. □

Similarly, $\overrightarrow{E_\oplus}$ is also the corresponding property for DotImpl and DotNeg. On the other hand, DotFull requires $\mathcal{F}$ to be the left identity of $\oplus$:

Lemma A.1.2. The following properties are equivalent (variables are universal):

1. Predicate (DotFull): $\mathcal{F} \cdot P \dashv\vdash P$.
2. Mapping: $\mathcal{F} \cdot (x \overset{\pi}{\mapsto} v) \dashv\vdash x \overset{\pi}{\mapsto} v$.
3. Heap: $\mathcal{F} \otimes h = h$.

4. Permission ($\overleftarrow{I_\otimes}$): $\mathcal{F} \otimes \pi = \pi$.

◁

Proof. We will show (3) ⇔ (4) ⇔ (2) and (1) ⇔ (4).

For (4) ⇒ (3), the case of empty heap is trivial. Otherwise let $a \in \mathsf{dom}(h)$ s.t. $h(a) = (v, \pi)$ then $\mathcal{F} \cdot h(a) = (v, \mathcal{F} \otimes \pi) = (v, \pi)$. Thus the two heaps are the same. For (3) ⇒ (4), we pick h s.t. $\mathsf{dom}(h) = \{a\}$ and $h(a) = (v, \pi)$. Then $\pi \cdot h(a) = (v, \mathcal{F} \otimes \pi)$ and thus $\pi = \mathcal{F} \otimes \pi$. The case (4) ⇔ (2) is similar.

For (1) ⇒ (4), pick $P = x \overset{\pi}{\mapsto}$ then (1) becomes (2). Since (2) ⇒ (4), the result follows. For (4) ⇒ (1), first consider left-to-right direction and assume $h, \rho \models \mathcal{F} \cdot P$. By definition, there exists h' s.t. $h = \mathcal{F} \otimes h'$ and $h', \rho \models P$. As (1) ⇒ (3), we have $h = h'$ and thus $h, \rho \models P$. The right-to-left direction is similar. □

The fact that $\mathcal{F}$ is the right identity of $\oplus$ follows from DotMap:

Lemma A.1.3. The following properties are equivalent (variables are universal):

1. Mapping (DotMap): $\pi \cdot x \mapsto v \dashv\vdash x \overset{\pi}{\mapsto} v$.
2. Permission ($\overrightarrow{I_\otimes}$): $\pi \otimes \mathcal{F} = \pi$.

where $x \mapsto v$ is the shortcut for $x \overset{\mathcal{F}}{\mapsto} v$. ◁

Proof. For (1) ⇒ (2), let $h, \rho \models \pi \cdot x \mapsto v$ then there exists h' s.t. $h = \pi \otimes h'$ and $h', \rho \models x \mapsto v$. By definition of map, we have $\mathsf{dom}(h') = \{x\}$ and $h'(x) = (v, \mathcal{F})$. As a result, $h(x) = (v, \pi \otimes \mathcal{F})$. As $h, \rho \models x \overset{\pi}{\mapsto} v$, we derive $\pi \otimes \mathcal{F} = \pi$. The direction (2) ⇒ (1) is similar. □

For the rule DotDot, the corresponding permission property is the associativity of $\otimes$ $A_\otimes$:

Lemma A.1.4. The following properties are equivalent (variables are universal):

1. Predicate (DotDot): $\pi_1 \cdot (\pi_2 \cdot P) \dashv\vdash (\pi_1 \otimes \pi_2) \cdot P$.
2. Mapping: $\pi_1 \cdot (\pi_2 \cdot x \overset{\pi}{\mapsto} v) \dashv\vdash (\pi_1 \otimes \pi_2) \cdot x \overset{\pi}{\mapsto} v$.
3. Heap: $\pi_1 \otimes (\pi_2 \otimes h) = (\pi_1 \otimes \pi_2) \otimes h$.

4. Permission ($A_{\otimes}$): $\pi_1 \otimes (\pi_2 \otimes \pi_3) = (\pi_1 \otimes \pi_2) \otimes \pi_3$.

◁

Proof. We will show (3) ⇔ (4) ⇔ (2) and (1) ⇔ (4).

For (4) ⇒ (3), the case empty heap is trivial. Otherwise, let $a \in \mathsf{dom}(h)$ s.t. $h(a) = (v, \pi)$. Then $\pi_1 \cdot (\pi_2 \cdot h)(a) = (v, \pi_1 \otimes (\pi_2 \otimes \pi))$ and $h(\pi_1 \otimes \pi_2) \cdot h(a) = (v, (\pi_1 \otimes \pi_2) \otimes \pi)$. By (4), we infer that two heaps are the same. For (4) ⇒ (3), choose h s.t. $\mathsf{dom}(h) = \{a\}$ and $h(a) = (v, \pi)$ and proceed in a similar manner as above. Also, (4) ⇔ (2) is similar.

For (4) ⇒ (1) left-to-right, let $h, \rho \models \pi_1 \cdot (\pi_2 \cdot P)$ then there exist h' s.t. $h = \pi_1 \otimes (\pi_2 \otimes h)$ and $h', \rho \models P$. As (4) ⇒ (3), we have $h = (\pi_1 \otimes \pi_2) \otimes h'$ and thus $h, \rho \models (\pi_1 \otimes \pi_2) \cdot P$. The right-to-left direction is similar. For (1) ⇒ (4), we pick $P = x \overset{\rho}{\mapsto} v$ then (1) becomes (1). As (2) ⇒ (4), the result follows. □

Last but not least, the two rules DotPlus and DotStar′ (*i.e.* without uniformity condition) require the left and right distributivity $\overleftarrow{D}$ and $\overrightarrow{D}$ of $\otimes$ over $\oplus$. However, both $\overleftarrow{D}$ and $\overrightarrow{D}$ only ensure the left-to-right direction for their akin scaling rules:

Lemma A.1.5. The following properties are equivalent (π_1, π_2 are fixed, other variables are universal):

1. Predicate (DotPlus): $(\pi_1 \oplus \pi_2) \cdot P \vdash (\pi_1 \cdot P) * (\pi_2 \cdot P)$.
2. Mapping: $(\pi_1 \oplus \pi_2) \cdot x \overset{\pi}{\mapsto} v \dashv\vdash (\pi_1 \cdot x \overset{\pi}{\mapsto} v) * (\pi_2 \cdot x \overset{\pi}{\mapsto} v)$.
3. Heap: $(\pi_1 \oplus \pi_2) \otimes h = (\pi_1 \otimes h) \oplus (\pi_2 \otimes h)$.
4. Permission ($\overrightarrow{D}$): $(\pi_1 \oplus \pi_2) \otimes \pi = (\pi_1 \otimes \pi) \oplus (\pi_2 \otimes \pi)$.

Furthermore, there exists fractional model satisfies ($\overrightarrow{D}$) but the right-to-left at predicate level fails. ◁

Proof. The proof is similar to other proofs above. For the counterexample, choose the rational model $\mathcal{Q} = \langle [0;1], +, \otimes \rangle$ then $\mathcal{Q}$ satisfies $\overleftarrow{D}$. We let $P = x \overset{0.6}{\longmapsto} v \vee y \overset{0.6}{\longmapsto} v$ and define a heap h as $\mathsf{dom}(h) = \{x, y\}$ and $h(x) = (v, 0.24), h(y) = (v, 0.24)$. Then $h \models 0.4 \cdot P * 0.4 \cdot P$ but $h \not\models 0.8 \cdot P$. As a result, $0.4 \cdot P * 0.4 \cdot P \not\vdash 0.8 \cdot P$. □

Lemma A.1.6. The following properties are equivalent (π is fixed, other variables are universal):

1. Predicate (DotStar′ $\vdash$): $\pi \cdot (P * Q) \vdash (\pi \cdot P) * (\pi \cdot Q)$.
2. Mapping: $\pi \cdot (x \overset{\pi_1}{\longmapsto} v * x \overset{\pi_2}{\longmapsto} v) \dashv\vdash (\pi \cdot x \overset{\pi_1}{\longmapsto} v) * (\pi \cdot x \overset{\pi_2}{\longmapsto} v)$.
3. Heap: $\pi \otimes (h_1 \oplus h_2) = (\pi \otimes h_1) \oplus (\pi \otimes h_2)$.
4. Permission ($\overleftarrow{D}$): $\pi \otimes (\pi_1 \oplus \pi_2) = (\pi \otimes \pi_1) \oplus (\pi \otimes \pi_2)$.

Furthermore, there exists fractional model satisfies $\overrightarrow{D}$ but the right-to-left at predicate level fails. ◁

Proof. Similar as above proofs. For the counterexample, we choose the rational model $\mathcal{Q} = \langle [0;1], +, \times \rangle$ then $\mathcal{Q}$ satisfies $\overrightarrow{D}$. We let $P = Q = x \overset{0.6}{\longmapsto} v$ and define h as $\mathsf{dom}(h) = \{x\}$ and $h(x) = (v, 0.6)$. Then $h \models 0.5 \cdot P * 0.5 \cdot Q$ but $h \not\models 0.5(P * Q)$ because $P * Q$ is not satisfiable. As a result, $(0.5 \cdot P) * (0.5 \cdot Q) \not\vdash 0.5 \cdot (P * Q)$. □

A.2 On essential axioms for fractional permissions

From Theorem A.1.2, we collectively discovered five necessary properties for permission model $\mathcal{M}$. In addition, as suggested in [COY07], the structure $\langle \mathcal{U}, \oplus \rangle$ needs to be a cancellative semi-group to force $*$ behave normally, *e.g.*, $*$ is commutative and associative. In detail, $\langle \mathcal{U}, \oplus \rangle$ satisfies:

$$\text{Commutativity of } \oplus\,(C_\oplus) : \pi \oplus \pi' = \pi' \oplus \pi$$

$$\text{Associativity of } \oplus\,(A_\oplus) : \pi_1 \oplus (\pi_2 \oplus \pi_3) = (\pi_1 \oplus \pi_2) \oplus \pi_3$$

$$\text{Cancellativity of } \oplus\,(E_\oplus) : \pi \oplus \pi_1 = \pi \oplus \pi_2 \Rightarrow \pi_1 = \pi_2$$

In our interpretation, the full permission $\mathcal{F}$ is the largest permission in which all other permissions are contained. One the other hand, we have $\mathcal{E}$ as the empty permission, which is essentially the identity for $\oplus$. These two elements can be expressed by the following

properties:

$$\text{Max } (M) : \forall\pi\exists\bar{\pi}.\ \bar{\pi} \oplus \pi = \mathcal{F} \qquad \text{Identity of } \oplus\,(I_{\oplus}) : \pi \oplus \mathcal{E} = \mathcal{E} \oplus \pi = \pi$$

Also, $\mathcal{M}$ needs the disjointness property which states only $\mathcal{E}$ can be joined with itself:

$$\text{Disjointness } (D) : \pi \oplus \pi = \pi' \Rightarrow \pi = \pi' = \mathcal{E}$$

We recall that a permission π is positive if $\pi \neq \mathcal{E}$. To prevent positive permissions from suddenly dropping to empty when scaling, we require positivity to be closed under $\otimes$, *i.e.*:

$$\text{Positivity } (P) : (\pi \neq \mathcal{E} \land \pi' \neq \mathcal{E}) \Rightarrow \pi \otimes \pi' \neq \mathcal{E}$$

We call 12 properties mentioned above and in Theorem A.1.2 the *Essential Scaling Permission Algebra* (ESPA). Optimistically, ESPA is consistent as it is satisfied by the following models:

Lemma A.2.1. Let $\mathcal{M}_1$ contain one element in which $\mathcal{F} = \mathcal{E}$ and $\mathcal{M}_2$ contain two elements $\{\mathcal{F}, \mathcal{E}\}$ s.t.:

1. $\mathcal{F} \oplus \mathcal{E} \ = \ \mathcal{E} \oplus \mathcal{F} \ \stackrel{\text{def}}{=} \ \mathcal{F}$.
2. $\mathcal{F} \oplus \mathcal{F}$ is not defined.
3. $\mathcal{E} \oplus \mathcal{E} \ \stackrel{\text{def}}{=} \ \mathcal{E}$.
4. $\mathcal{F} \otimes a \ = \ a \otimes \mathcal{F} \ \stackrel{\text{def}}{=} \ a$.
5. $\mathcal{E} \otimes a \ = \ a \otimes \mathcal{E} \ \stackrel{\text{def}}{=} \ \mathcal{E}$.

for $a \in \{\mathcal{F}, \mathcal{E}\}$ then $\mathcal{M}_1$, $\mathcal{M}_2$ satisfy ESPA. ◁

Surprisingly, $\mathcal{M}_1$, $\mathcal{M}_2$ are the only ESPA models as justified by the following result:

Theorem A.2.1. If $\mathcal{M}$ satisfies ESPA then it only has at most two elements. ◁

Proof. Suppose there exists $\mathcal{M}$ of at least three elements. Then $\mathcal{F} \neq \mathcal{E}$* and there exists a

*if $\mathcal{E} = \mathcal{F}$ then for any π, we have $\pi \stackrel{I_{\oplus}}{=} \mathcal{E} \oplus \pi = \mathcal{F} \oplus \pi \stackrel{M}{=} (\bar{\pi} \oplus \pi) \oplus \pi \stackrel{A_{\oplus}}{=} \bar{\pi} \oplus (\pi \oplus \pi) \stackrel{D}{\Rightarrow} \pi = \mathcal{E}$.

s.t. $a \neq \mathcal{F}$ and $a \neq \mathcal{E}$. By M, let $\bar{a}$ s.t. $\bar{a} \oplus a = \mathcal{F}$ then:

$$a \overset{I_\otimes}{=} \mathcal{F} \otimes a \overset{M}{=} (\bar{a} \oplus a) \otimes a \overset{\overrightarrow{D}}{=} (\bar{a} \otimes a) \oplus (a \otimes a)$$

By a different transformation, we have:

$$a \overset{I_\otimes}{=} a \otimes \mathcal{F} \overset{M}{=} a \otimes (\bar{a} \oplus a) \overset{\overleftarrow{D}}{=} (a \otimes \bar{a}) \oplus (a \otimes a)$$

Thus:

$$(\bar{a} \otimes a) \oplus (a \otimes a) = (a \otimes \bar{a}) \oplus (a \otimes a) \overset{E_\oplus}{\Rightarrow} \bar{a} \otimes a = a \otimes \bar{a} \quad (F_1)$$

On the other hand:

$$\begin{aligned}
\mathcal{F} \overset{I_\otimes}{=} \mathcal{F} \otimes \mathcal{F} &\overset{M}{=} (a \oplus \bar{a}) \otimes (a \oplus \bar{a}) \\
&\overset{\overrightarrow{D},\overleftarrow{D}}{=} (a \otimes a) \oplus (a \otimes \bar{a}) \oplus (\bar{a} \otimes a) \oplus (\bar{a} \otimes \bar{a}) \\
&\overset{A_\oplus}{=} (a \otimes a) \oplus [(a \otimes \bar{a}) \oplus (\bar{a} \otimes a)] \oplus (\bar{a} \otimes \bar{a}) \\
&\overset{F_1}{=} (a \otimes a) \oplus [(a \otimes \bar{a}) \oplus (a \otimes \bar{a})] \oplus (\bar{a} \otimes \bar{a}) \\
&\overset{D}{\Rightarrow} a \otimes \bar{a} = \mathcal{E} \quad (F_2)
\end{aligned}$$

Also, we have $\bar{a} \neq \mathcal{E}$ (otherwise $a \overset{I_\oplus}{=} a \oplus \mathcal{E} = a \oplus \bar{a} \overset{M}{=} \mathcal{F}$ contradicts to our assumption). However, this gives use $a \otimes \bar{a} \neq \mathcal{E}$ by positivity P and thus contradicts to F_2. Hence $\mathcal{M}$ cannot exist. □

Theorem A.2.1 potentially puts an end to the construction of useful fractional permission models. Fortunately, we overcome this shortcoming by sacrificing the right distributivity axiom $\overrightarrow{D}$ in ESPA. To be precise, let *Scaling Permission Algebra* (SPA) be the set of axioms in ESPA except $\overrightarrow{D}$. Then together with the precision and uniformity conditions, we can show that fractional heaps constructed from SPA models are SSA model in Fig. 3.16 and thus satisfy the scaling rules in Fig. 3.2:

Theorem A.2.2. Let $\mathcal{M} = \langle \mathcal{U}, \oplus, \otimes, \mathcal{F}, \mathcal{E} \rangle$ be a SPA model then its fractional heaps are SSA model and thus satisfy all scaling rules. Furthermore, there exists a SPA model with infinitely many elements. ◁

Proof. Let $\mathsf{identity}(a) \stackrel{\text{def}}{=} a = \mathcal{E}$, $\mathsf{force}(\pi, \pi') \stackrel{\text{def}}{=} \pi$, and $\mathsf{mul}(\pi, \pi') \stackrel{\text{def}}{=} \pi \otimes \pi'$. Then $\mathcal{M}' = \langle \mathcal{U}^+, \oplus, \otimes, \mathcal{F}, \mathsf{identity}, \mathsf{force}, \mathsf{mul} \rangle$ is a SSA. By a result from §3.4.4, the fractional heap model $\mathtt{Addr} \rightharpoonup \mathtt{Val} \times \mathcal{U}^+$ is also SSA. □

As suggested in §3.5.1, it is desirable that the left inverse of $\otimes$ exists, *i.e.*, for a permission π, there exists π' s.t. $\pi' \otimes \pi = \mathcal{F}$. Unfortunately, only the left inverse of $\mathcal{F}$ exists which is itself:

Lemma A.2.2. If $\mathcal{M}$ satisfies SPA then it also satisfies $\pi \otimes \pi' = \mathcal{F} \Rightarrow \pi = \pi' = \mathcal{F}$. ◁

Proof. Suppose $\pi' \otimes \pi = \mathcal{F}$. Then:

$$\pi' \stackrel{I_\otimes}{=} \pi' \otimes \mathcal{F} \stackrel{M}{=} \pi' \otimes (\pi \oplus \bar{\pi}) \stackrel{\overleftarrow{D}}{=} (\pi' \otimes \pi) \oplus (\pi' \otimes \bar{\pi}) = \mathcal{F} \oplus (\pi' \otimes \bar{\pi})$$

Also let $a = \pi' \otimes \bar{\pi}$ then:

$$\pi' = \mathcal{F} \oplus a \stackrel{M}{=} (\bar{a} \oplus a) \oplus a \stackrel{A_\oplus}{=} \bar{a} \oplus (a \oplus a) \stackrel{D}{\Rightarrow} a = \mathcal{E}$$

Thus $\pi' = \mathcal{F} \oplus \mathcal{E} \stackrel{I_\oplus}{=} \mathcal{F}$ and also $\mathcal{F} = \pi' \otimes \pi = \mathcal{F} \otimes \pi \stackrel{I_\otimes}{=} \pi$. □