A Certified Decision Procedure for Sophisticated Fractional Permissions

Le Xuan Bach[†] Aquinas Hobor[‡,†]

[†]School of Computing [‡]Yale-NUS College
National University of Singapore

Abstract. We develop a certified decision procedure for reasoning about systems of equations over the "tree share" fractional permission model of Dockins et al. Fractional permissions are used to track and reason about shared ownership of resources between multiple parties, *e.g.* in a concurrent program. We show how to extend the theory over systems of equations of tree shares to handle both positive and negative clauses using a new proof technique we dub "separate, cut and glue". In addition to being certified, our new procedure enjoys better performance than previous work and incorporates a better treatment nonzero variables.

1 Introduction

Fractional shares enable reasoning about shared ownership of resources between multiple parties, *e.g.* in a concurrent program [3]. The obvious model for fractional shares is rational numbers in $[0, 1]$, with 0 representing no ownership, 1 representing full ownership, and $0 < x < 1$ representing partial ownership. A verification system then allows programs different actions depending on the amount of ownership of some resource such as a memory cell, *e.g.* with 1 allowing both reading and writing, $0 < x < 1$ allowing only reading, and 0 allowing nothing. The map between permission levels and actions can be considered a kind of policy and is distinct from the model used to represent the shares.

Unfortunately, rational numbers are not an ideal model for shares. Consider the following recursive predicate definition for fractionally-owned binary trees:

$$\mathsf{tree}(\ell, \pi) \quad \triangleq \quad (\ell = \mathsf{null} \ \wedge \mathsf{emp}) \ \vee \ (\ell \overset{\pi}{\mapsto} (\ell_l, \ell_r) \star \mathsf{tree}(\ell_l, \pi) \star \mathsf{tree}(\ell_r, \pi)) \quad (1)$$

Here we write $a \overset{\pi}{\mapsto} b$ to indicate that memory location a contains b and is owned with (nonempty) share π. This tree predicate is obtained directly from the standard recursive predicate for binary trees in separation logic by asserting only π ownership of the root and recursively doing the same for the left and right substructures, and so at first glance looks obviously correct. The problem is that when $\pi \in (0,]$, then tree can describe some non-tree directed acyclic graphs.

Parkinson proposed a model based on sets of natural numbers that solved this issue but introduced others [14], and then Dockins *et al.* [7] proposed the following "tree share" model, which fixes all of the aforementioned issues. A tree share τ is inductively defined as a binary tree with boolean leaves: $\tau \triangleq \circ \mid \bullet \mid \widehat{\tau\ \tau}$.

Here $\circ$ denotes an "empty" leaf while $\bullet$ a "full" leaf. The tree $\circ$ is thus the empty share, and $\bullet$ the full share. There are two "half" shares:  and  , and four "quarter" shares, beginning with  . It is a feature that the two half shares are distinct from each other—indeed, this is directly related to why the tree predicate in (1) describes exactly fractional trees in separation logic when $\pi \in \tau$.

Notice that we presented the first quarter share as  instead of *e.g.*  . This is deliberate: the second choice is not a valid share because the tree is not in *canonical form*. A tree is in canonical form when it is in its most compact representation under the inductively-defined equivalence relation $\cong$:

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

Maintaining canonical form is a headache in Coq but does not introduce any fundamental difficulty. Accordingly, for this presentation we will simply fold and unfold trees to/from canonical form when required by the narrative.

Due to their good metatheorical properties, a number of program logics [9,11] and verification tools incorporate tree shares [16,12,1]. Historically, program verifiers have had more difficulty using tree shares because they must develop algorithms that can handle them. Program verifiers care about entailment between predicates. In previous work [12] we showed how to divide an entailment between separation logic formulae incorporating fractional ownership into 1) a fraction-free separation logic entailment, and 2) an entailment between systems of share equations. For example, in classical separation logic the entailment $x \overset{\pi_1}{\mapsto} a * x \overset{\pi_2}{\mapsto} b \vdash \exists \pi.\ x \overset{\pi}{\mapsto} c$ divides into the fraction-free $x \mapsto a \wedge a = b \vdash x \mapsto c$ and the share entailment $\pi_1 \oplus \pi_2 = \pi_3 \vdash \exists \pi.\ \pi = \pi_3$.

Here $\oplus$ is the "join" operation on shares; its formal definition is somewhat technical due to the necessity of managing the canonical forms but the core idea is quite straightforward. Simply unfold both trees under $\cong$ into the same shape and join them leafwise using the rules $\circ \oplus \circ = \circ$, $\circ \oplus \bullet = \bullet$, and $\bullet \oplus \circ = \bullet$; afterwards refold under $\cong$ back into canonical form. Here is an example:

Because $\bullet \oplus \bullet$ is undefined, the join relation on trees is a partial operation. Dockins *et al.* [7] prove that the join relation satisfies a number of useful axioms *e.g.* associativity and commutativity (§2.2 has the full list). One key axiom, not satisfied by $(\mathbb{Q}, +)$, is "disjointness": $x \oplus x = y \ \Rightarrow\ x = \circ$. Disjointness is the axiom that forces the tree predicate—equation 1—to behave properly.

In previous work we developed a tool to decide the tree share entailments [13]. The present paper improves on our previous work in several ways. From a practical point of view, our new tool is fully machine-checked in Coq, giving the highest level of assurance that both its underlying theory and implementation are rock solid. A trend in recent years has been to develop verification toolsets

within Coq [2,5,1]; since certified tools generally only depend on other certified tools, they have not been able to take advantage of our previous implementation[1]. Moreover, we have incorporated a number of heuristics that meaningfully improve performance without sacrificing soundness or completeness; some of these should be usable in other certified decision procedures in the future.

From a theoretical point of view, our major improvement on previous work is a better treatment of negations; negative clauses in logic are often significantly more difficult to handle than positive ones are. Our previous theory supported a very limited form of negation, where we could force variables to be nonempty (*i.e.* $\neg\pi = \circ$). Unfortunately, we were unable to mechanize this argument in Coq. Our new theory can handle arbitrary negative clauses (*i.e.* $\neg\pi_1 \oplus \pi_2 = \pi_3$), although to date we have only implemented the portion of it necessary to be compatible with existing clients into Coq. A second theoretical improvement is a more careful treatment of existential variables.

A final contribution of this paper—although in fact we cover it early—is that we show how to use tree shares to define a set of permissions, *i.e.* predicates over shares that specify a simple but useful policy. Our procedure can prove sound inference rules about the policy, enabling their easy use in verification systems.

The rest of this paper is structured as follows. In §2 we show how to use tree shares to define and reason about a set of permissions. In §3 we provide an overview of our decision procedure. In §4 we prove the main theoretical result. First we generalize our previous results into a technique we call "cut and glue", and then show how to handle negative clauses using a refined technique we dub "separate, cut, and glue". Our new technique is used to prove that the new procedures are sound even with negative clauses. In §5 we discuss our rather large (over 40k LOC) certified implementation and benchmark its performance. Finally, in §6 we discuss related & future work and conclude. We put formalities that muddle the main narrative, such as the definition of $\oplus$, in appendix A.

Our tool is completely machine-checked in Coq and available at:

`http://www.comp.nus.edu.sg/~lxbach/share_prover/`

2 Permission modeling and extraction

We begin by showing how to use tree shares in a more encapsulated way via permission policies, which can help verification tools isolate share-related reasoning from *e.g.* separation-logic reasoning. We provide a model of a nontrivial policy as an example of what can be reasoned about using our system. Policies are written in a restricted logic of share formulae, but we provide evidence that this format is able to express interesting facts about shares.

2.1 Share policies

A policy $\mathbb{P}$ is a set of predicates over shares together with associated inference rules. One simple policy is to allow, for each address x, either a single writer

[1] Verification tools that are not certified can still benefit from our certified decision procedure by running the OCaml code generated by the Coq extraction mechanism.

or multiple readers. We can model this policy using two predicates $\mathsf{READ}(\pi)$ and $\mathsf{WRITE}(\pi)$. To assert that a thread has *e.g.* permission to write to x at a program point one can then assert $x \overset{\pi}{\mapsto} v \wedge \mathsf{WRITE}(\pi)$.

For inference rules, consider the following:

$$\frac{\mathsf{WRITE}(\pi)}{\mathsf{READ}(\pi)}\ \textsc{WriteRead} \qquad \frac{}{\mathsf{WRITE}(\bullet)}\ \textsc{FullWrite} \qquad \frac{\mathsf{WRITE}(\pi)}{\pi = \bullet}\ \textsc{WriteFull}$$

$$\frac{\mathsf{READ}(\pi)}{\exists \pi_1, \pi_2.\ \pi_1 \oplus \pi_2 = \pi \wedge \mathsf{READ}(\pi_1) \wedge \mathsf{READ}(\pi_2)}\ \textsc{SplitRead}$$

That is, a thread that can write to location x can also read from it; the full permission enables writing; writing implies the full permission; and any read permission can be split into two read permissions. To prove that these rules are sound one would provide a model for the READ and WRITE predicates, such as $\mathsf{READ}(\pi) \triangleq \neg(\pi = \circ)$ and $\mathsf{WRITE}(\pi) \triangleq \pi = \bullet$. Given these definitions, the first three rules are trivial, although the proof of SplitRead may not be obvious.

A more interesting problem is to determine if a given set of inference rules is complete. The group above is not; for example it leaves out the following interesting rule, which implies that if a given thread can write to location x then none of the other threads can read from it:

$$\frac{\pi_1 \oplus \pi_2 = \pi_3 \wedge \mathsf{WRITE}(\pi_1) \wedge \mathsf{READ}(\pi_2)}{\bot}\ \textsc{SoloWrite}$$

A second interesting problem is in extending a set of permissions. For example, Parkinson has shown that sometimes a thread may have neither permission to read from nor write to a location x, but it would like to know that x is still allocated [15]. To extend our set of permissions to reason about this behavior, we can add to $\mathbb{P}$ the predicates SAFE, which allows neither reading nor writing but prevents deallocation; and FULL, which allows arbitrary access. Writing $\tau_1 \subseteq \tau_2$ to mean $\exists \tau'.\ \tau_1 \oplus \tau' = \tau_2$, the following model supports all four permissions:

$$\begin{array}{llll} \mathsf{WRITE}(\pi) & \triangleq & \overset{\frown}{\bullet\ \circ} \subseteq \pi & \mathsf{READ}(\pi) \triangleq \exists \pi'.\pi' \subseteq \pi \wedge \pi' \subseteq \overset{\frown}{\bullet\ \circ} \wedge \neg(\pi' = \circ) \\ \mathsf{FULL}(\pi) & \triangleq & \pi = \bullet & \mathsf{SAFE}(\pi) \triangleq \exists \pi'.\pi' \subseteq \pi \wedge \pi' \subseteq \overset{\frown}{\circ\ \bullet} \wedge \neg(\pi' = \circ) \end{array}$$

Some of our previous inference rules are no longer sound (*e.g.* FullWrite). Other rules are still sound (*e.g.* SplitRead), along with a number of new rules (*e.g.*, it is possible to write an analogous SplitSafe rule). Finding and proving the soundness of a complete set of inference rules is significantly more complex.

Although the policy model we provide above is useful in practice, rather than providing a fixed set of inference rules, it is more flexible to allow the user of a verification system to provide a model for the permission policy they wish to use and then have the system determine whether this model allows the entailments necessary to verify the program. To enable this flexibility we need to have a sound and complete decision procedure over fractional shares.

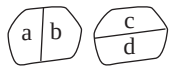
Functional:	$x \oplus y = z_1 \;\Rightarrow\; x \oplus y = z_2 \;\Rightarrow\; z_1 = z_2$
Commutative:	$x \oplus y \;=\; y \oplus x$
Associative:	$x \oplus (y \oplus z) \;=\; (x \oplus y) \oplus z$
Cancellative:	$x_1 \oplus y = z \;\Rightarrow\; x_2 \oplus y = z \;\Rightarrow\; x_1 = x_2$
Unit:	$\exists u.\, \forall x.\, x \oplus u = x$
Disjointness:	$x \oplus x = y \;\Rightarrow\; x = y$
Cross split:	$a \oplus b = z \wedge c \oplus d = z \Rightarrow \exists ac, ad, bc, bd.$ $ac \oplus ad = a \wedge bc \oplus bd = b \wedge ac \oplus bc = c \wedge ad \oplus bd = d$
	$\forall$  $\exists$ 
Infinite Splitability:	$x \neq \circ \;\Rightarrow\; \exists x_1, x_2.\, x_1 \neq \circ \;\wedge\; x_2 \neq \circ \;\wedge\; x_1 \oplus x_2 = x$

Fig. 1. Properties of tree shares

2.2 Share formulae

Our permission predicates above can be written as formulae as follows:

$$\pi \;\triangleq\; \tau \mid v \qquad \Delta \;\triangleq\; \pi_1 \oplus \pi_2 = \pi_3 \mid \neg(\pi_1 \oplus \pi_2 = \pi_3) \mid \Delta \wedge \Delta \qquad \Phi \;\triangleq\; \exists v.\Phi \mid \Delta$$

Shares π are either tree constants τ or variables v. Share clauses Δ are positive clauses, negative clauses, and conjunctions of clauses—that is, they are systems of positive and negative clauses. In the theory it is convenient to treat equalities $\pi_1 = \pi_2$ as macros for $\pi_1 \oplus \circ = \pi_2$, although our certified tool tracks equalities separately for optimization purposes. Share formulae Φ are a list of existential bindings followed by a system of clauses.

A context ρ is a mapping from variable names to tree shares. The semantics of forcing, written $\rho \models \Phi$, is straightforward:

$$\begin{array}{lcl} \rho \models \pi_1 \oplus \pi_2 = \pi_3 & \triangleq & \rho(\pi_1) \oplus \rho(\pi_2) = \rho(\pi_3) \\ \rho \models \neg(\pi_1 \oplus \pi_2 = \pi_3) & \triangleq & \neg(\rho \models \pi_1 \oplus \pi_2 = \pi_3) \\ \rho \models \Delta_1 \wedge \Delta_2 & \triangleq & (\rho \models \Delta_1) \wedge (\rho \models \Delta_2) \\ \rho \models \exists v.\, \Phi & \triangleq & \exists \tau.\, [v \mapsto \tau]\rho \models \Phi \end{array}$$

We write $\Phi \vdash \Phi' \triangleq \forall \rho.\, (\rho \models \Phi) \Rightarrow (\rho \models \Phi')$ to indicate the usual entailment between formulae. Accordingly, any variables not explicitly bound existentially are universally bound and shared between both sides of the entailment.

Eagle-eyed readers may notice that our permission inference rules above were not stated in exactly the right format due to the placement of quantifiers, *e.g.* the SplitRead rule actually has three levels of nested quantifiers. However, each of these rules can be converted into our format by pulling the existential to the outer level as long as we are careful to always choose fresh names for existential quantifiers and rename inner clauses accordingly.

Assuming the fresh names trick, one might wonder how general our format is. One interesting exercise is to examine the metatheoretical properties of tree shares described by Dockins et al. [7]; these are given in figure 1. Several of

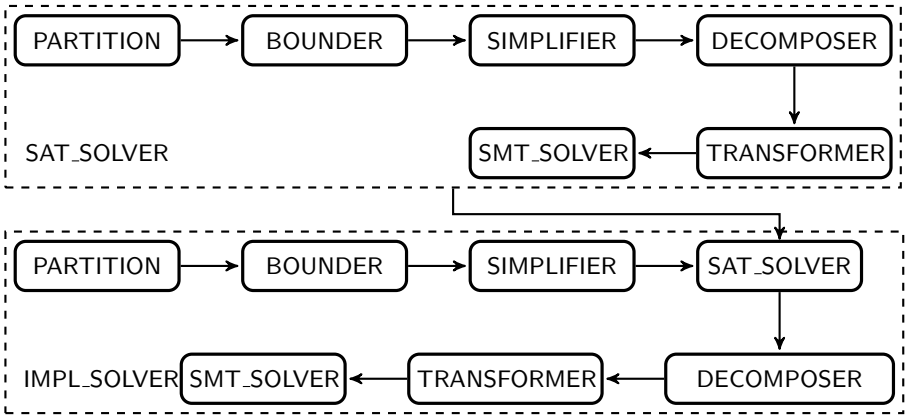


Fig. 2. SAT solver and IMPL solver

these are the standard properties of separation algebras [4], 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— to behave properly), Cross-split (used *e.g.* in settings involving overlapping data structures), and Infinite splitability (to verify divide-and-conquer algorithms). Very encouragingly, all of the axioms except for "Unit" are expressible as entailments in our format. Unit requires the order of quantifiers to swap; our format can express the weaker "Multiunit axiom" $\forall x.\ \exists u.\ x \oplus u = x$ as well as the more concrete unit axiom $\forall x.\ x \oplus \circ = x$.

The main purpose of our decision procedure is to decide entailments between systems of share equations. We present experimental results in §5, but as a teaser we are pleased to report that our certified implementation proved all nine axioms in less than one second, even running in interpreted Gallina within Coq.

Unfortunately, while the $\oplus$ operation has many useful properties for verifying programs, they are not strong enough for algebraic techniques like Gaussian elimination (which in any event is useless with negative clauses even in $\mathbb{Q}$), necessitating the theory and implementation in the remainder of this paper.

3 Overview of our decision procedures

We are now ready to outline our decision procedure, although here we focus on the overall structure, given in figure 2 and outlying components. The heart of the procedure will be deferred until §4. Our procedure is written entirely in Gallina and certified to be bug-free in Coq.

We have written two procedures to solve problems over share equation systems, satisfaction **SAT** and entailment **IMPL**. Generally speaking they share similar components; moreover, satisfaction checking is used as a subroutine in entailment checking because certain transformations we wish to apply are only sound if the antecedent is satisfiable. Here is a description of each component:

- PARTITION: the original system is partitioned into *independent subsystems*. This optimization improves the tool's performance significantly in practice. Our PARTITION module is reusable for other kinds of equation systems and is further detailed in §5.
- BOUNDER: the bounder gives each variable v an initial bound $\circ \subseteq v \subseteq \bullet$ and tries to narrow the bound 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 out of the system. In other cases we can find a contradiction (when the upper bound goes below the lower bound), allowing us to terminate the procedure early. The bounder is an improved version of a previous incomplete solver [12], and can improve performance by two orders of magnitude in practice.
- SIMPLIFIER: the combination of a substitution engine and several effective heuristics for reducing the overall difficulty via calculation (*e.g.* from $v \oplus \tau_1 = \tau_2$ we can compute an exact value for v using an inverse of $\oplus$) and finding a contradictions. The core idea was contained in our previous work [13], so our main contribution here is just the certified implementation.
- DECOMPOSER: the system is *decomposed* into an *equivalent* set of subsystems of lesser height. This component will be discussed in detail in §4.
- TRANSFORMER: further heuristics, plus a transformation from systems of equations of height zero into equivalent boolean sentences using the rules $\circ \mapsto \bot$, $\bullet \mapsto \top$, $\neg(v = \circ) \mapsto \neg v$, $\pi_1 = \pi_2 \mapsto (\pi_1 \wedge \pi_2) \vee (\neg\pi_1 \wedge \neg\pi_2)$, $\pi_1 \oplus \pi_2 = \pi_3 \mapsto (\pi_1 \wedge \neg\pi_2 \wedge \pi_3) \vee (\neg\pi_1 \wedge \pi_2 \wedge \pi_3) \vee (\neg\pi_1 \wedge \neg\pi_2 \wedge \neg\pi_3)$. Although this transformation is dead simple, its correctness is not and relies heavily Lemma 4 and 7, presented in §4.
- SMT_SOLVER: The solver uses simple quantifier elimination to check the validity of the boolean sentence. Our SMT solver is rather naïve, and thus forms the performance bottleneck of our tool, but we could not find a suitable Gallina alternative. Despite its naiveté, as discussed in §5, our overall performance seems acceptable in practice due to the above heuristics.

3.1 Language of the certified tool

Formally, our tool takes systems of equations Σ as input (one system for **SAT**, two for **IMPL**). A constraint system Σ is a 4-ary tuple of lists: $(l_{\texttt{nz}}, l_{\texttt{ex}}, l_{\texttt{eql}}, l_{\texttt{eqn}})$, where $l_{\texttt{nz}}$ is a list of *non-zero* variables, $l_{\texttt{ex}}$ is a list of *existential* variables, $l_{\texttt{eql}}$ is a list of *equalities* and $l_{\texttt{eqn}}$ is a list of *equations*. Equations are simply positive clauses as above, *i.e.* $\pi_1 \oplus \pi_2 = \pi_3$. Equalities are a special type of equation, *i.e.* $\pi_1 = \pi_2$ is equivalent to $\pi_1 \oplus \circ = \pi_2$. However, equalities are handled separately in the tool because they can be more easily substituted away.

The list of existential variables is exactly equivalent to the existentials in the share formulae discussed in §2. However, the list of nonzeros is unfortunately weaker, since it only allows negative clauses of the format $\neg(v = \circ)$. We had

already written a substantial amount of code & machine-checked proof before realizing that our proof technique to handle nonzeros could handle arbitrary negative clauses. We intend to update our tool to handle arbitrary negative clauses in the future but for now our implementation lagging our theory. One positive is that, as already discussed, even this limited form of negative clauses can express interesting policies for verification purposes.

The formal semantics of systems of equations is very similar to the share formulae given earlier. To handle lists of existentials, we define the notion of a *context override*, written $\rho[\rho' \Leftarrow l]$ and defined as:

$$\rho[\rho' \Leftarrow l] \quad \triangleq \quad \lambda x. \begin{cases} \rho'(v) & \text{when } x \in l \\ \rho(v) & \text{otherwise} \end{cases}$$

Given this, we can define the semantics of a system of equations as

Definition 1 (Semantics of a system of tree equations).

$$\begin{array}{lcl} \rho \models v & \triangleq & \neg(\rho(v) = \circ) \\ \rho \models [] & \triangleq & \top \\ \rho \models a :: l & \triangleq & \rho \models a \ \wedge \ \rho \models l \\ \rho \models (l_{nz}, l_{ex}, l_{eql}, l_{eqn}) & \triangleq & \exists \rho'. \rho[\rho' \Leftarrow l_{ex}] \ \models \ l_{nz} \ \wedge \ l_{eql} \ \wedge \ l_{eqn} \end{array}$$

We write $\rho \models v$ to mean that v is nonzero (by analogy to Boolean logic where one can say that a variable x is true by just writing the variable itself). We extend satisfaction to lists in the natural way via a series of conjunctions. A context ρ satisfies a system Σ when we can existentially find another context ρ' and override the original context at the existential variables l_{ex}, and then pass the resulting context to the list of nonzeros, equations, and equalities.

Formally our tool answers two queries:

- **SAT**(Σ): Is Σ satisfiable, *i.e.* $\exists \rho. \rho \models \Sigma$?
- **IMPL**(Σ_1, Σ_2): Does Σ_1 entail Σ_2, *i.e.* $\forall \rho. (\rho \models \Sigma_1 \Rightarrow \rho \models \Sigma_2)$?

Notice that $l_{\texttt{ex}}$ is unnecessary in **SAT**(Σ) and thus can be ignored. Although **SAT** can be considered a special case of **IMPL** by adding a trivial antecedent and making all variables existential, it is computationally easier to solve **SAT** and so we keep it separate. (Moreover, our **IMPL** uses **SAT** as a subroutine!)

4 Correctness of the decision procedures

4.1 Definitions and Previous Work

We provide some key results from our previous work [13] that will help build the algorithm as well as verify its correctness. Notice that our previous Constraint System Σ is simply a binary tuple $(l_{\texttt{eql}}, l_{\texttt{eqn}})$ without existential and non-zero variables. We defined four operators: *decomposition* $\phi(\tau)$, which takes a tree and "splits" it at the top; *left* $\lfloor \overleftarrow{\tau} \rfloor_n$ and *right* $\lfloor \overrightarrow{\tau} \rfloor_n$ *rounding*, which take trees and

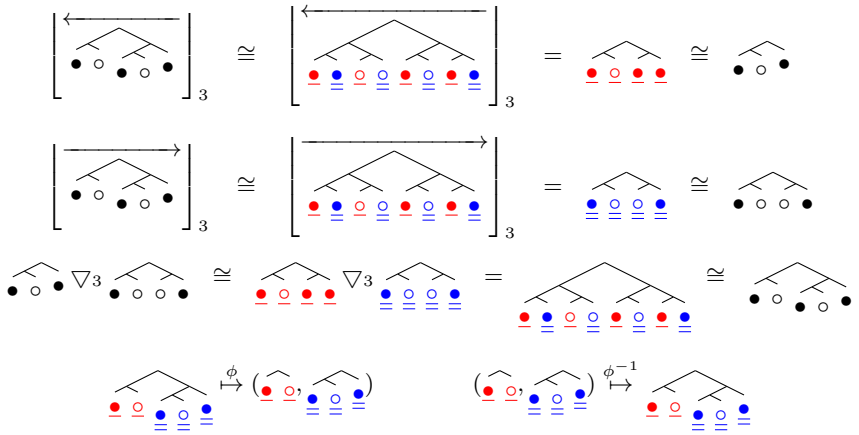
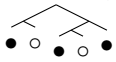


Fig. 3. Example of tree operators

split them at the bottom; and *averaging* $\tau_1 \bigtriangledown_n \tau_2$, which recombines trees "bottom up". A minor new addition to these functions is *recomposition* $\phi^{-1}(\tau_1, \tau_2)$, which recombines trees "top down". The bottom-focused functions take an additional parameter n which specifies the height at which the operation should take place. We illustrate these functions using Figure 3 to help the readers gain intuition; their formal definition is given in appendix A.2. Our example is the tree  at height 3, together with its derived trees. To visually track what is going on for $\lfloor \overleftarrow{\tau} \rfloor_n$, $\lfloor \overrightarrow{\tau} \rfloor_n$, 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 $\phi(\tau)$ and $\phi^{-1}(\tau_1, \tau_2)$, we color red for leaves from the left sub-tree and blue for leaves from the right sub-tree. Those functions work nicely with $\oplus$ as showed in Lemma 1, which was verified in Coq by induction on the tree height.

Lemma 1 ([13]). *Assume* $\lfloor \overleftarrow{\tau_i} \rfloor_n = \tau_i^l$, $\lfloor \overrightarrow{\tau_i} \rfloor_n = \tau_i^r$, $\phi(\tau_i) = (\tau_{i,l}, \tau_{i,r})$, $i \in \{1, 2, 3\}$.

- If $n > |\tau_i|$ then $\tau_i^l = \tau_i^r = \tau_i$ (2)
- If $n = |\tau_i|$ then $|\tau_i^l| < n \wedge |\tau_i^r| < n$ (3)
- If $n > |\tau_i|$ then $\tau_i \bigtriangledown_n \tau_i = \tau_i$ (4)
- $\tau_1 \oplus \tau_2 = \tau_3 \Leftrightarrow \tau_1^l \oplus \tau_2^l = \tau_3^l \wedge \tau_1^r \oplus \tau_2^r = \tau_3^r$ (5)
- $\tau^l \bigtriangledown_n \tau^r = \tau_i \Leftrightarrow \tau^l = \tau_i^l \wedge \tau^r = \tau_i^r$ (6)
- $\tau_1 \oplus \tau_2 = \tau_3 \Leftrightarrow \tau_{1,l} \oplus \tau_{2,l} = \tau_{3,l} \wedge \tau_{1,r} \oplus \tau_{2,r} = \tau_{3,r}$ (7)

Using $(\overleftarrow{\lfloor\rfloor}, \overrightarrow{\lfloor\rfloor}, \bigtriangledown)$, Le *et al.* [13] proved the sufficiency of finite search for **SAT** and **IMPL** on the domain $\mathbb{T}$:

Theorem 1 (Finite search for SAT and IMPL [13]). *Let* $|\rho|, |\Sigma|, |(\Sigma_1, \Sigma_2)|$ *denote the largest tree's height in* $\rho, \Sigma, (\Sigma_1, \Sigma_2)$ *respectively (0 if there is no tree*

*constant). To solve **SAT** and **IMPL**, it is sufficient to consider only solutions ρ whose height, $|\rho|$, is up to the height of the system ($|\Sigma|$ for **SAT** and $|(\Sigma_1, \Sigma_2)|$ for **IMPL**).*

The technique we used to prove Theorem 1 is called "*cut and glue*". First one defines some *cut functions* $f_i : \mathbb{T} \times D \to \mathbb{T}, i = 1 \ldots k$ and a glue function $g : \mathbb{T}^k \times D \to \mathbb{T}$ where D contains extra information. f_i and g can be extended over the domain of Constraint Systems and contexts. Let $\rho_j \models \Sigma_j, j = 1 \ldots n$, one uses f_i to *cut* ρ_j and Σ_j such that the $\models$ relation is still preserved: $f_i(\rho_j, d) \models f_i(\Sigma_j)$, or g to *glue* them together: $g(\rho_1, \ldots, \rho_n, d) \models g(\Sigma_1, \ldots, \Sigma_n)$. Here comes the magic: there are f_i and g such that, under the right settings, change ρ_j but not Σ_j. As a result, they *create* new solutions. A deeper result is they implicitly imply the solution space for the Constraint System can be constructed *via a set of basic solutions*. In addition, this technique was used to decompose the system into sub-systems which can be transformed into equivalent boolean formulas. In our case, the cut functions are $(\overleftarrow{\lfloor\rfloor}, \overrightarrow{\lfloor\rfloor}, \phi)$ while the glue functions are $(\bigtriangledown, \phi^{-1})$.

Here we will sketch the main idea. Property 2 and 3 tell us the rounding functions, which are undefined when $n < |\tau_i|$, reduce the tree's height when $n = |\tau_i|$ and leave it intact otherwise while property 5 states how the $\oplus$ relation can be *equivalently transferred* into the sub-trees. As a result, one can construct two new *smaller* solutions ρ', ρ'' from the original solution ρ by applying $\overleftarrow{\lfloor\rfloor}$ and $\overrightarrow{\lfloor\rfloor}$ respectively. This step can be repeated until $|\rho| \leq |\Sigma|$. This is the key result for **SAT**.

IMPL proof requires property 6 that shows the relationship between $\bigtriangledown$ and $(\overleftarrow{\lfloor\rfloor}, \overrightarrow{\lfloor\rfloor})$: briefly speaking one is the inverse of the other. Using property 5 and 6, we proved in Coq the relationship between $\bigtriangledown$ and $\oplus$:

$$\forall n, \tau_i, \tau_i', \tau_i''(i \in \{1,2,3\}). \bigwedge_{i=1}^{3} \tau_i \bigtriangledown_n \tau_i' = \tau_i'' \Rightarrow$$
$$\tau_1'' \oplus \tau_2'' = \tau_3'' \Leftrightarrow \tau_1 \oplus \tau_2 = \tau_3 \wedge \tau_1' \oplus \tau_2' = \tau_3' \tag{8}$$

This property provides the construction of a new solution ρ'' from two solutions ρ, ρ' using $\bigtriangledown$ where the parameter n needs to be set at least $\max(|\rho|, |\rho'|, |\Sigma|) + 1$. Thus $(\overleftarrow{\lfloor\rfloor}, \overrightarrow{\lfloor\rfloor}, \bigtriangledown)$ establishes a 2-way construction among solutions of height at least $|\Sigma|$. Consequently, all solutions of height greater than $|\Sigma|$ can be constructed from solutions of height at most $|\Sigma|$. This result verifies the correctness of **IMPL**.

Even we know the search space can be reduced to be finite, designing a practical algorithm to solve **SAT** and **IMPL** is a challenge because the search space is exponentially exploded. Let a_n be the number of canonical trees of height at most n then it satisfies the reoccurence relation: $a_0 = 2, a_n = a_{n-1}^2 = 2^{2^{n-1}}$. Thus the number of canonical trees increases exponentially as n increases. This is when ϕ comes to rescue. At the first glance, one may find ϕ is similar to $(\overleftarrow{\lfloor\rfloor}, \overrightarrow{\lfloor\rfloor})$ with respect to $\oplus$: property 7 and 5 are almost identical. The key difference is $(\overleftarrow{\lfloor\rfloor}, \overrightarrow{\lfloor\rfloor})$ change the tree's shape at bottom leaves while the change made by ϕ

is at the root. As a result, ϕ fails to preserve the constants in Σ and cannot be used to construct new solutions. However, ϕ can be utilized to construct *smaller sub-systems* whose answers are combined to derive the answer for the original system. In fact, this function is the heart of the DECOMPOSER component. Let $\phi(\Sigma) = (\Sigma_1, \Sigma_2)$ where Σ_1 (Σ_2) is constructed by keeping all variables in Σ while replacing all constant τ with its first (second) projection on the result of $\phi(\tau)$. In a similar fashion, one defines $\phi(\rho) = (\rho_1, \rho_2)$ for context ρ. Property 7 can be generalized for Constraint Systems: $\rho \models \Sigma \Leftrightarrow \rho_1 \models \Sigma_1 \wedge \rho_2 \models \Sigma_2$. Using ϕ, any Σ is reduced to a list of Σ_i such that $|\Sigma_i| = 0$. At height 0, Theorem 1 tells us each tree variable can be treated as a boolean variable, thus **SAT** and **IMPL** are transformed into equivalent boolean sentences whose validity is checked by SMT solvers.

4.2 Decision Procedure for SAT

Our system $\Sigma = (l_{\mathtt{nz}}, l_{\mathtt{ex}}, l_{\mathtt{eql}}, l_{\mathtt{eqn}})$ is extended from $(l_{\mathtt{eql}}, l_{\mathtt{eqn}})$ by allowing existential and non-zero variables. While the addition of $l_{\mathtt{ex}}$ is simple and straightforward as it does not break any desired properties of $(\overleftarrow{\lfloor\rfloor}, \overrightarrow{\lfloor\rfloor}, \bigtriangledown, \phi)$, it is not the case for $l_{\mathtt{nz}}$. In general, negative constraints follow the *disjunctive* pattern:

Lemma 2. *Assume* $\lfloor \overleftarrow{\tau_i} \rfloor_n = \tau_i^l$, $\lfloor \overrightarrow{\tau_i} \rfloor_n = \tau_i^r$, $\phi(\tau_i) = (\tau_{i,l}, \tau_{i,r})$, $i \in \{1,2,3\}$.

- $\tau^l \bigtriangledown_n \tau^r \neq \tau_i \Leftrightarrow \tau^l \neq \tau_i^l \vee \tau^r \neq \tau_i^r$ (9)
- $\tau_1 \oplus \tau_2 \neq \tau_3 \Leftrightarrow \tau_1^l \oplus \tau_2^l \neq \tau_3^l \vee \tau_1^r \oplus \tau_2^r \neq \tau_3^r$ (10)
- $\tau_1 \oplus \tau_2 \neq \tau_3 \Leftrightarrow \tau_{1,l} \oplus \tau_{2,l} \neq \tau_{3,l} \vee \tau_{1,r} \oplus \tau_{2,r} \neq \tau_{3,r}$ (11)
- $\tau_1 \neq \tau_2 \Leftrightarrow \tau_1^l \neq \tau_2^l \vee \tau_1^r \neq \tau_2^r$ (12)
- $\tau_1 \neq \tau_2 \Leftrightarrow \tau_{1,l} \neq \tau_{2,l} \vee \tau_{1,r} \neq \tau_{2,r}$ (13)
- $\forall n, \tau_i, \tau_i', \tau_i''(i \in \{1,2,3\}). \bigwedge_{i=1}^{3} \tau_i \bigtriangledown_n \tau_i' = \tau_i'' \Rightarrow$
  $\tau_1'' \oplus \tau_2'' \neq \tau_3'' \Leftrightarrow \tau_1 \oplus \tau_2 \neq \tau_3 \vee \tau_1' \oplus \tau_2' \neq \tau_3'$ (14)

Proof. A constructive proof by induction on the tree height was verified in Coq.

Here is the problem: $(l_{\mathtt{ex}}, l_{\mathtt{eql}}, l_{\mathtt{eqn}})$ follow conjunctive pattern with respect to $(\overleftarrow{\lfloor\rfloor}, \overrightarrow{\lfloor\rfloor}, \bigtriangledown, \phi)$. The outlier $l_{\mathtt{nz}}$ makes a finite search space proof like the one in Theorem 1 cannot be carried out uniformly as before. In fact, it is no longer true in this system. For example, let $\Sigma = ([], x :: y :: z :: [], [], (x \oplus y = z) :: [])$. $\mathbf{SAT}(\Sigma)$ is the formula $\exists x, y, z. x \neq \circ \wedge y \neq \circ \wedge z \neq \circ \wedge x \oplus y = z$ which *has no solution of height 0.* It does, however, have solutions of height 1, e.g. $\rho = \{x \mapsto \overset{\frown}{\circ\bullet}, y \mapsto \overset{\frown}{\bullet\circ}, z \mapsto \bullet\}$. Can one simply extend the search space to 1 extra height? Unfortunately, the answer is negative: $\Sigma' = ([], x_1 :: x_2 :: x_3 :: x_4 :: x_5 :: [], [], (x_1 \oplus x_2 = x_3) :: (x_3 \oplus x_4 = x_5) :: [])$ has no solution of height 0 or 1. Even search space is known, another major concern is how one can solve **SAT** and **IMPL** *effectively*? Our old procedure employs ϕ to uniformly break down

the system into sub-systems of height zero which are handled by SMT solvers. Negative constraints fail to satisfy the crucial property 7, therefore cannot be broken down uniformly with other positive constraints.

It turns out if we restrict $l_{\mathtt{nz}}$ to be *at most* singleton, i.e. $l_{\mathtt{nz}}$ contains at most one element then there is a complete scheme to handle such systems. We call these systems *singleton*. Moreover, is it possible to equivalently reduce Σ and (Σ_1, Σ_2) to a set of singleton systems. Using two above results, **SAT** and **IMPL** become solvable. We call this technique *"separate, cut and glue"*. At the moment, out tool only supports negative constraints $\neg(v = \circ)$. However the correctness proof below can be generalized in a nature way to handle negative constraints $\neg(v_1 \oplus v_2 = v_3)$ by using properties 10 and 11 instead of properties 12 and 13. First, we introduce some wrappers for the Constraint System:

Definition 2 (System Wrappers for SAT). *Let $\Sigma = (l_{nz}, l_{ex}, l_{eql}, l_{eqn})$. We define $\mathsf{NES}(\Sigma)$ to be a new system with l_{nz} dropped out, and $\mathsf{SES}(\Sigma)$ be a set of systems such that each of them contains a single non-zero variable from l_{nz}:*

- $\mathsf{NES}(\Sigma) = ([], l_{ex}, l_{eql}, l_{eqn})$
- $\mathsf{SES}(\Sigma) = \{(v :: [], l_{ex}, l_{eql}, l_{eqn}) \mid v \in l_{nz}\}$

Lemma 3 (Decompose for SAT). *Let $\Sigma = (v :: [], l_{ex}, l_{eql}, l_{eqn})$, $\phi(\Sigma) = (\Sigma_l, \Sigma_r)$ and assume $\mathsf{NES}(\Sigma)$ is satisfiable. Σ is satisfiable iff either Σ_l or Σ_r is satisfiable.*

Proof. Verified in Coq. Notice that our old method [13] requires both Σ_l and Σ_r to be satisfiable. But that is the case without non-zero variables. One may ask why such a more complicated structure has a simpler condition. It is because we require $\mathbf{SAT}(\mathsf{NES}(\Sigma))$ checked first and thus checking $\mathbf{SAT}(\Sigma)$ is a tedious 2-step process. The main idea of the proof is the following:

$\Rightarrow$: Let $\rho \models \Sigma$ and $\phi(\rho) = (\rho_l, \rho_r)$. Using property 7, one can show ρ_l (ρ_r) satisfies positive constraints of Σ_l (Σ_r), i.e. $\mathsf{NES}(\Sigma_l)$ ($\mathsf{NES}(\Sigma_r)$). Property 13 tells us either ρ_l or ρ_r satisfies the non-zero variable constraint. As a result, either Σ_l or Σ_r is satisfiable.

$\Leftarrow$: WLOG assume $\mathbf{SAT}(\Sigma_l)$ and $\rho_l \models \Sigma_l$. Also, let $\rho' \models \mathsf{NES}(\Sigma)$ and $\phi(\rho') = (\rho'_l, \rho'_r)$. By property 7, $\rho'_l \models \mathsf{NES}(\Sigma_l)$ and $\rho'_r \models \mathsf{NES}(\Sigma_r)$. One *glues* ρ_l and ρ'_r together using ϕ^{-1} to create a new context ρ such that $\forall v \in V. \phi(\rho(v)) = (\rho_l(v), \rho'_r(v))$. It is not hard to see $\rho \models \Sigma$.

Lemma 4 (Finite Search at height 0 for SAT). *Let $\Sigma = (v :: [], l_{ex}, l_{eql}, l_{eqn})$ and $|\Sigma| = 0$. Σ is satisfiable iff there exists a solution $\rho \models \Sigma$ and $|\rho| = 0$.*

Proof. Verified in Coq. It suffices to show $\Rightarrow$ direction. Let $\rho \models \Sigma$, $\phi(\Sigma) = (\Sigma_l, \Sigma_r)$, $\phi(\rho) = (\rho_l, \rho_r)$ and assume $|\rho| > 0$. By properties of ϕ and similar arguments as in the proof of Lemma 3, one can show $\rho_l \models \Sigma_l \vee \rho_r \models \Sigma_r$ and $\max(|\rho_l|, |\rho_r|) < |\rho|$. At height zero $\Sigma_l = \Sigma_r = \Sigma$. Thus either ρ_l or ρ_r is a solution of Σ. Start this process over with the new solution until its height is zero.

Lemma 5 (Constraint Separation for SAT). *Let* $\Sigma = (l_{nz}, l_{ex}, l_{eql}, l_{eqn})$ *and assume* $\mathsf{NES}(\Sigma)$ *is satisfiable.* Σ *is satisfiable iff each* $\Sigma' \in \mathsf{SES}(\Sigma)$ *is satisfiable.*

Proof. Verified using Coq. $\Rightarrow$ direction is trivial. For $\Leftarrow$ direction, let $\rho' \models \mathsf{NES}(\Sigma)$, $l_{\mathtt{nz}} = x_1 :: \ldots :: x_n :: []$, $\mathsf{SES}(\Sigma) = \{\Sigma_i = (v_i :: [], l_{\mathtt{ex}}, l_{\mathtt{eql}}, l_{\mathtt{eqn}}) | i = 1 \ldots n\}$ and $\rho_i \models \Sigma_i$. We use similar *gluing technique* in Lemma 3 but this time with $\bigtriangledown$. Simply speaking, one starts with ρ' as the *carrier* and keeps *gluing* ρ_i into it to create a context that satisfies all the negative constraints. We define $\rho'_1 = \rho' \bigtriangledown_n \rho_1$ where $n = \max(|\rho'|, |\rho_1|, |\Sigma|) + 1$ then $\rho'_1 \models \mathsf{NES}(\Sigma) \wedge \rho'_1 \models v_1 :: []$ by property 4, 8 and 14. Glue ρ'_1 and ρ_2 together creates $\rho'_2 \models \mathsf{NES}(\Sigma) \wedge \rho'_2 \models v_1 :: v_2 :: []$. This process is repeated until ρ'_n which satisfies $\mathsf{NES}(\Sigma)$ and $l_{\mathtt{nz}}$. Thus $\rho'_n \models \Sigma$.

We are now ready to state our main result for **SAT**:

Theorem 2 (Decision Procedure for SAT). *Let* $\Sigma = (l_{nz}, l_{ex}, l_{eql}, l_{eqn})$. *There is an algorithm to solve* $\boldsymbol{SAT}(\Sigma)$.

Proof. The description of the algorithm is as follow: first we check $\mathbf{SAT}(\mathsf{NES}(\Sigma))$ using the procedure in [13]. If it is unsatisfiable, so is Σ. Otherwise, consider $S = \mathsf{SES}(\Sigma) = \{\Sigma_i | i = 1 \ldots n\}$: by Lemma 5, it is sufficient to check satisfiability of each system Σ_i in S. We use ϕ to decompose Σ_i into sub-systems of height zero. By Lemma 3, Σ_i is satisfiable iff one of its zero sub-systems is satisfiable (notice the assumption in the Lemma 3 is automatically satisfied at each decompose level due to $\mathbf{SAT}(\mathsf{NES}(\Sigma))$ and by properties of ϕ). Finally, by Lemma 4, we can treat each tree variable in the zero sub-system as boolean variable. Thus, **SAT** for such sub-systems can be equivalently transformed into a boolean sentence and solved by SMT solvers.

4.3 Decision Procedure for IMPL

IMPL is significantly more subtle than **SAT** because non-zero variables are allowed in both systems and one cannot simply ignore $l_{\mathtt{ex}}$ as in **SAT**. Fortunately, it is still solvable by applying similar technique for **SAT**: one tries to *separate* the negative constraints by dividing the original system into *singleton* systems whose solutions are easy to verify. The main Theorem for **IMPL** is followed by a sequence of intermediate Lemmas. The readers should know in advance that the proof for our **IMPL** procedure is simplified because we ignore the presence of existential variables. In our defense, the existential variables make the proof more tedious and less intuitive but not harder as one needs to deal with the nuisance of function overriding. Our full proof has been implemented in Coq and can be found at `http://www.comp.nus.edu.sg/~lxbach/share_prover/`.

Definition 3 (System Wrappers for IMPL). *Let* $\Sigma_i = (l^i_{nz}, l^i_{ex}, l^i_{eql}, l^i_{eqn})$, $i \in \{1, 2\}$. *We define* $\mathsf{NIS}(\Sigma_1, \Sigma_2)$ *the new system which both* l^1_{nz}, l^2_{nz} *are removed,* $\mathsf{SIS}(\Sigma_1, \Sigma_2)$ *the set of systems whose first non-zero list is removed and second non-zero list is singleton,* $\mathsf{PIS}(\Sigma_1, \Sigma_2)$ *the set of systems which both non-zero lists are singleton:*

- $\mathsf{NIS}(\Sigma_1, \Sigma_2) = (\mathsf{NES}(\Sigma_1), \mathsf{NES}(\Sigma_2))$
- $\mathsf{SIS}(\Sigma_1, \Sigma_2) = \{(\mathsf{NES}(\Sigma_1), \Sigma_2') \mid \Sigma_2' \in \mathsf{SES}(\Sigma_2)\}$
- $\mathsf{PIS}(\Sigma_1, \Sigma_2) = \{(\Sigma_1', \Sigma_2') \mid \Sigma_1' \in \mathsf{SES}(\Sigma_1) \wedge \Sigma_2' \in \mathsf{SES}(\Sigma_2)\}$

Lemma 6 (Decompose for IMPL). *Let* $\Sigma_i = (l^i_{nz}, l^i_{ex}, l^i_{eql}, l^i_{eqn})$, $\phi(\Sigma_i) = (\Sigma_{i,l}, \Sigma_{i,r})$, $\Pi = (\Sigma_1, \Sigma_2)$, $\phi(\Pi) = (\Pi_1, \Pi_2) = ((\Sigma_{1,l}, \Sigma_{2,l}), (\Sigma_{1,r}, \Sigma_{2,r}))$.

a. *Assume* $\boldsymbol{IMPL}(\mathsf{NIS}(\Pi))$ *and* $(l^1_{nz}, l^2_{nz}) = ([], v :: [])$. $\boldsymbol{IMPL}(\Pi)$ *holds iff either* $\boldsymbol{IMPL}(\Pi_l)$ *or* $\boldsymbol{IMPL}(\Pi_r)$ *holds.*
b. *Assume* $\boldsymbol{IMPL}(\mathsf{NIS}(\Pi))$, $\neg\boldsymbol{IMPL}(\mathsf{SIS}(\Pi))$ *and* $(l^1_{nz}, l^2_{nz}) = (v_1 :: [], v_2 :: [])$. $\boldsymbol{IMPL}(\Pi)$ *holds iff both* $\boldsymbol{IMPL}(\Pi_l)$ *and* $\boldsymbol{IMPL}(\Pi_r)$ *hold.*

Proof. Verified in Coq. For part a, $\Rightarrow$ direction is achieved using proof by contradiction. Let $\rho_l \models \Sigma_{1,l} \wedge \neg(\rho_l \models \Sigma_{2,l})$ and $\rho_r \models \Sigma_{1,l} \wedge \neg(\rho_r \models \Sigma_{2,l})$. We *glue* ρ_l and ρ_r together using ϕ^{-1} to create a new context ρ. By properties of ϕ, $\rho \models \Sigma_1$ thus $\rho \models \Sigma_2$. But $\phi(\rho) = (\rho_l, \rho_r)$ so either $\rho_l \models \Sigma_{2,l}$ or $\rho_r \models \Sigma_{2,r}$ (this disjunction is due to the negative constraint in Σ_2), which is a contradiction. For the other direction, WLOG assume $\mathbf{IMPL}(\Pi_l)$. Let $\rho \models \Sigma_1$ and $\phi(\rho) = (\rho_l, \rho_r)$ then $\rho_l \models \Sigma_{1,l} \wedge \rho_r \models \Sigma_{1,r}$. Using the assumption, we derive $\rho_l \models \Sigma_{2,l}$. Furthermore, as $\mathbf{IMPL}(\mathsf{NIS}(\Pi))$ implies both $\mathbf{IMPL}(\mathsf{NIS}(\Pi_1))$ and $\mathbf{IMPL}(\mathsf{NIS}(\Pi_2))$, one can show $\rho_r \models \mathsf{NES}(\Sigma_{2,r})$. As ρ is the gluing context between ρ_l and ρ_r, we conclude $\rho \models \Sigma_2$.

In b, the $\Leftarrow$ direction is similar as in part a. For $\Rightarrow$ direction, it suffices to show $\mathbf{IMPL}(\Pi_l)$. Let $\rho_l \models \Sigma_{1,l}$. From the assumptions $\neg\mathbf{IMPL}(\mathsf{SIS}(\Pi))$ and $\mathbf{IMPL}(\mathsf{NIS}(\Pi))$, one can find ρ' such that $\rho' \models \mathsf{NES}(\Sigma_1) \wedge \neg(\rho' \models v_2)$. Let $\phi(\rho') = (\rho'_l, \rho'_r)$ then $\rho'_l \models \mathsf{NES}(\Sigma_{1,l}) \wedge \rho'_r \models \mathsf{NES}(\Sigma_{1,r})$ and $\neg(\rho'_l \models v_2) \wedge \neg(\rho'_r \models v_2)$. Let ρ be the gluing context between ρ_l and ρ'_r via ϕ^{-1} then $\rho \models \Sigma_1$, which implies $\rho \models \Sigma_2$. As a result, $\rho \models v_2$ so $\rho_l \models v_2 \vee \rho'_r \models v_2$. One concludes $\rho_l \models v_2$ as the other case is exclusive. Also, by $\rho \models \mathsf{NES}(\Sigma_2)$, we have $\rho_l \models \mathsf{NES}(\Sigma_{2,l})$. Thus $\rho_l \models \Sigma_{2,l}$.

Lemma 7 (Finite Search at height 0 for IMPL). *Let* $\Sigma_i = (l^i_{nz}, l^i_{ex}, l^i_{eql}, l^i_{eqn})$, $\Pi = (\Sigma_1, \Sigma_2)$ *and* $|\Pi| = 0$.

a. *Assume* $(l^1_{nz}, l^2_{nz}) = ([], v :: [])$. $\boldsymbol{IMPL}(\Pi)$ *holds iff it holds for all contexts of height 0.*
b. *Assume* $(l^1_{nz}, l^2_{nz}) = (v_1 :: [], v_2 :: [])$ *and* $\boldsymbol{IMPL}(\mathsf{NIS}(\Pi))$ *holds.* $\boldsymbol{IMPL}(\Pi)$ *holds iff it holds for all contexts of height 0.*

Proof. Verified in Coq. For part a, it suffices to prove $\Leftarrow$ direction. We apply induction on the height of the context. Assume $\mathbf{IMPL}(\Pi)$ holds for all contexts of height at most n. Let $\rho \models \Sigma_1$, $|\rho| = n + 1$, $\phi(\rho) = (\rho_l, \rho_r)$, $\phi(\Sigma_1) = (\Sigma_{1,l}, \Sigma_{1,r}) = (\Sigma_1, \Sigma_1)$. Thus $\rho_l \models \Sigma_1 \wedge \rho_r \models \Sigma_1$ and by our induction hypothesis, $\rho_l \models \Sigma_2 \wedge \rho_r \models \Sigma_2$. As a result, $\rho \models \Sigma_2$.

Part b can be proved in a similar fashion as part a, except we have $\rho_l \models \Sigma_1 \vee \rho_r \models \Sigma_1$ instead. WLOG, let $\rho_l \models \Sigma_1$. As $\rho \models \mathsf{NES}(\Sigma_1)$, we have $\rho_r \models \mathsf{NES}(\Sigma_1)$. Thus $\rho_l \models \Sigma_2$ and $\rho_r \models \mathsf{NES}(\Sigma_2)$. This result implies $\rho \models \Sigma_2$.

Lemma 8 (Constraint Separation for IMPL). *Let* $\Sigma_i = (l^i_{nz}, l^i_{ex}, l^i_{eql}, l^i_{eqn})$, $\Pi = (\Sigma_1, \Sigma_2)$.

a. *Assume* $l^2_{nz} = []$ *and* $\textbf{\textit{SAT}}(\Sigma_1)$ *holds.* $\textbf{\textit{IMPL}}(\Pi)$ *holds iff* $\textbf{\textit{IMPL}}(\mathsf{NIS}(\Pi))$ *holds.*
b. *Assume* $l^1_{nz} = []$, $l^2_{nz} \neq []$. $\textbf{\textit{IMPL}}(\Pi)$ *holds iff for every* $\Sigma' \in \mathsf{SIS}(\Pi)$, $\textbf{\textit{IMPL}}(\Sigma')$ *holds.*
c. *Assume* $l^1_{nz} \neq []$, $l^2_{nz} \neq []$. $\textbf{\textit{IMPL}}(\Pi)$ *holds iff for each* $v_i \in l^2_{nz}$, *there exists* $\Pi' \in \mathsf{SIS}(\Pi)$ *such that* $\Pi' = (\Sigma'_1, (v_i :: [], t^2_{ex}, t^2_{eql}, t^2_{eqn}))$ *and* $\textbf{\textit{IMPL}}(\Pi')$ *holds.*

Proof. Verified in Coq.

a. $\Leftarrow$ direction is trivial. For the other direction, let $\rho \models \mathsf{NES}(\Sigma_1)$. As $\mathbf{SAT}(\Sigma_1)$ holds, choose $\rho' \models \Sigma_1$. Glue ρ, ρ' via $\bigtriangledown$ with $n = \max(|\rho|, |\rho'|, |\Pi|) + 1$ to create a new context ρ'' such that $\rho'' \models \Sigma_1$ and thus $\rho'' \models \Sigma_2$. One *cuts* ρ'' using $\overleftarrow{[]}$ to get back ρ and by properties of $\overleftarrow{[]}$, $\rho \models \Sigma_2$.

b. Both directions are trivial.

c. $\Rightarrow$: Pick an arbitrary $v \in l^2_{\mathtt{nz}}$ and let $l^1_{\mathtt{nz}} = v_{k_1} :: \ldots :: v_{k_j} :: []$, $\Pi_{k_i} = (\Sigma_{1,k_i}, \Sigma_{2,k_i}) \in \mathsf{SIS}(\Pi)$ that has the non-zero v_{k_i} in its antecedent system. Assume there is no such Π' then $\exists \rho_{k_i}.\rho_{k_i} \models \Sigma_{1,k_i} \wedge \neg(\rho_{k_i} \models \Sigma_{2,k_i})$, $\forall k_i \in \{1, \ldots, j\}$. Gluing $\rho_{k_1}, \ldots, \rho_{k_j}$ together via $\bigtriangledown$ to create a new context ρ such that $\rho \models \Sigma_1 \wedge \neg(\rho \models \Sigma_2)$, which is a contradiction. For the other direction, let $\rho \models \Sigma_1$ and $\Pi' = (\Sigma'_1, \Sigma'_2) \in \mathsf{SIS}(\Pi)$ then $\rho \models \Sigma'_1$. From the given condition, it is not hard to derive $\rho \models (v_i :: [], t^2_{\mathtt{ex}}, t^2_{\mathtt{eql}}, t^2_{\mathtt{eqn}})$, $\forall v_i \in l^2_{\mathtt{nz}}$. Thus $\rho \models \Sigma_2$.

Lemma 9. *Let* $\Sigma_i = (l^i_{nz}, l^i_{ex}, l^i_{eql}, l^i_{eqn})$.

- $\neg\textbf{\textit{SAT}}(\Sigma_1)$ *is a sufficient condition for* $\textbf{\textit{IMPL}}(\Sigma_1, \Sigma_2)$.
- $\textbf{\textit{IMPL}}(\mathsf{NES}(\Sigma_1), \Sigma_2)$ *is a sufficient condition for* $\textbf{\textit{IMPL}}(\Sigma_1, \Sigma_2)$.
- *Assume* $\textbf{\textit{SAT}}(\Sigma_1)$ *holds then* $\textbf{\textit{IMPL}}(\mathsf{NIS}(\Sigma_1, \Sigma_2))$ *is a necessary condition for* $\textbf{\textit{IMPL}}(\mathsf{NES}(\Sigma_1), \Sigma_2)$ *and* $\textbf{\textit{IMPL}}(\Sigma_1, \Sigma_2)$.

Proof. Verified in Coq.

Theorem 3 (Decision Procedure for IMPL). *Let* $\Sigma_i = (l^i_{nz}, l^i_{ex}, l^i_{eql}, l^i_{eqn})$ *and* $\Pi = (\Sigma_1, \Sigma_2)$. *There is an algorithm to solve* $\textbf{\textit{IMPL}}(\Pi)$.

Proof. Here is the description of the algorithm: first we check $\mathbf{SAT}(\Sigma_1)$ using the procedure in Theorem 2: an answer No for **SAT** implies a trivial answer Yes for **IMPL**. Otherwise, we check $\mathbf{IMPL}(\mathsf{NIS}(\Pi))$ whose No answer yields a No answer for **IMPL**. Next, we classify Π based on the criteria in Lemma 8 and use its results together with Lemma 6 to decompose Π into sub-systems of height 0. Lemma 7 helps transform the **IMPL** for each sub-system into an equivalent boolean formula which can be solved by SMT solvers.

5 Coq implementation and experimental results

Our decision procedure follows Theorems 2 and 3 with the addition of numerous heuristics to improve the performance. The overall architecture of our tool was discussed in §3 (figure 2). Here we give additional details on the PARTITION and DECOMPOSER modules and benchmark our performance.

The PARTITION *module.* The goal of this module is to separate a constraint system (Σ for **SAT** and (Σ_1, Σ_2) for **IMPL**) into *independent systems*. Two systems are independent of each other if they do not share any common variable (existentially bound variables are locally bound and thus ignored).

The partition function is implemented *generically*. To build the module, we must specify types of *variables* V, *equations* E, and contexts C. We must also provide a function $\sigma : E \to L(V)$ that can extract 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 properties that relate extraction, overriding, and evaluation as follows:

- Disjointness: $\forall \rho, \rho', e, l.\ (\rho \models e) \wedge (\sigma(e) \cap l = []) \ \Rightarrow\ \rho[\rho' \Leftarrow l] \models e$.
- Inclusion: $\forall \rho, \rho', e, l.\ (\rho \models e) \wedge (\sigma(e) \subset l) \ \Rightarrow\ \rho'[\rho \Leftarrow l] \models e$.

In some sense these properties are dual, jointly specifying 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 override the variables that are in the equation then we can ignore the original context.

To use PARTITION in our system is simple for **SAT**, but to handle **IMPL** we need to "tag" equations and variables as coming from the antecedent or consequent before partitioning and then separate the resulting partitioned systems into antecedents and consequents afterwards using these tags, a technique similar to the cut and glue method we outlined in §4.

The implementation of PARTITION is nontrivial in functional languages like Coq. One reason is that we need a purely functional union-find data structure, which is in turn built upon finite partial maps implemented as red-black trees. The termination of "find" turns out to be rather subtle as it relies on the acylicity of chains of indirect links in the red-black trees.

Given union-find, the core algorithm is straightforward: each variable is put into a singleton set and while processing an equation we union the corresponding sets. Lastly, we extract the sets and filter the equations into components.

The DECOMPOSER *module.* DECOMPOSER: this component mainly uses ϕ whose domain is extended to Constraint Systems. The decomposition follows the guidance from Lemmas 3, 5 for **SAT** and Lemmas 6, 8 for **IMPL** with some efficient modifications: we decompose the system without modifying the non-zero variable list until the system is at height zero. Only once we are at height zero do we separate the nonzero list; this saves us many duplicate decompositions.

Experimental results. Our implementation is *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 and Haskell, although at present a small bug in Coq 8.4pl6's extraction code requires a small human edit to the generated code.

We benchmarked our tool using the 102 standalone test cases developed for previous work (53 **SAT** and 49 **IMPL**) [13] as well as the new tests described in §2. These tests cover a variety tricky cases such as large number of variables, deep tree constants, etc. We used a Intel i7 with 8GB RAM and compiled the OCaml code with ocamlopt. The total running time to test all 102 previous tests is **0.06 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 [6] or MiniSAT [8] if performance bottlenecks in the future. Even running as interpreted Gallina code within Coq, the time is extremely encouraging at **25 seconds**. After we port to Coq 8.5 we will be able to use the native_compute tactic to substantially increase performance.

6 Related work, future work, and conclusion

Boyland first proposed fractional shares over $\mathbb{Q}$ [3]. Parkinson showed that $\mathbb{Q}$'s lack of disjointness caused trouble and proposed modelling shares as subsets of $\mathbb{N}$ [14]. Dockins *et al.* proposed the tree share model used in the present paper to fix issues with Parkinson's model [7]. Hobor *et al.* were the first to use tree shares in a program logic [10], followed by Hobor and Gherghina [11] and Villard [16]. Hobor and Gherghina [12], Villiard [16], and Appel *et al.* [1] subsequently integrated shares into program verification tools with various incomplete solvers.

Our previous work Le *et al.* [13] is the direct precursor to the present work, providing a decision procedure for tree shares. Our current work improves upon this in the theory by enabling negative clauses and in practice by providing a certified and significantly better-performing implementation.

Future work. We have plans to examine the theory further to support general logical formulae (including arbitrary quantifier use) and perhaps monadic 2nd order logic. Dockins *et al.* also define a kind of multiplicative operation $\bowtie$ between shares should be better understood. In addition, we plan to improve our tool to handle generalized negative clauses as supported by our current theory.

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 using our "separate, cut and glue" technique. The two algorithms were implemented and certified in Coq with numerous heuristics to achieve good performance. Our code was extracted to OCaml and benchmarked.

References

1. 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, New York, NY, USA, 2014.
2. Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! - A framework for higher-order separation logic in Coq. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, pages 315–331, 2012.
3. John Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72, 2003.
4. Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 366–378, 2007.
5. Adam Chlipala. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 391–402, 2013.
6. Leonardo de Moura and Nikolaj Bjrner. Z3: An efficient SMT solver. In *TACAS*, 2008.
7. Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, pages 161–177, 2009.
8. N. Een and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–508, 2003.
9. Aquinas Hobor. *Oracle Semantics*. PhD thesis, Princeton University, Department of Computer Science, Princeton, NJ, October 2008.
10. Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *17th European Symposium of Programming (ESOP 2008)*, pages 353–367, April 2008.
11. Aquinas Hobor and Cristian Gherghina. Barriers in concurrent separation logic. In *ESOP*, pages 276–296, 2011.
12. Aquinas Hobor and Cristian Gherghina. Barriers in concurrent separation logic: Now with tool support! *Logical Methods in Computer Science*, 8(2), 2012.
13. Xuan Bach Le, Cristian Gherghina, and Aquinas Hobor. Decision procedures over sophisticated fractional permissions. In *APLAS*, 2012.
14. Matthew Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.
15. Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. Modular verification of a non-blocking stack. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 297–302, 2007.
16. Jules Villard. *Heaps and Hops*. Ph.D. thesis, Laboratoire Spécification et Vérification, École Normale Supérieure de Cachan, France, February 2011.

A Formal definitions

A.1 From §1: definition of tree shares, canonicalization, and $\oplus$

The formal definition of tree shares begins with boolean binary tree $\mathbb{B}$:

$$t \in \mathbb{B} \quad \triangleq \quad \circ \mid \bullet \mid \widehat{t_1\ t_2} \tag{15}$$

We use the metavariable t to range over boolean binary tree. Let $|t|$ be the height of t, starting from 0. As in §1, define an equivalence relation $\cong$ between trees as follows:

$$\frac{}{\circ \cong \circ} \qquad \frac{}{\bullet \cong \bullet} \qquad \frac{}{\circ \cong \widehat{\circ\ \circ}} \qquad \frac{}{\bullet \cong \widehat{\bullet\ \bullet}} \qquad \frac{t_1 \cong t_1' \quad t_2 \cong t_2'}{\widehat{t_1\ t_2} \ \cong \ \widehat{t_1'\ t_2'}}$$

We use the metavariable τ to range over tree shares $\tau \in \mathbb{T} \subset \mathbb{B}$, whose formal definition adds the restriction that no sub-tree $\widehat{\circ\ \circ}$ or $\widehat{\bullet\ \bullet}$ is allowed:

$$\tau \in \mathbb{T} \quad \triangleq \quad \circ \mid \bullet \mid \widehat{\tau_1\ \tau_2} \text{ assuming } (\tau_1, \tau_2) \notin \{(\circ, \circ), (\bullet, \bullet)\} \tag{16}$$

$\mathbb{T}$ is isomorphic to the set $\mathbb{B}/\cong$, where $\tau \in \mathbb{T}$ corresponds to the set $\{t \in \mathbb{B} | t \cong \tau\}$.

We can coerce an element $t \in \mathbb{B}$ into its principal representation τ using the following canonicalization function $\delta : \mathbb{B} \to \mathbb{T}$, defined as:

Definition 4 (Canonicalization).

$$\delta(t) \quad \triangleq \quad \begin{cases} \circ & \text{if } t \in \{\circ, \widehat{\circ\ \circ}\} \\ \bullet & \text{if } t \in \{\bullet, \widehat{\bullet\ \bullet}\} \\ t & \text{if } t \in \{\widehat{\bullet\ \circ}, \widehat{\circ\ \bullet}\} \\ \delta(\widehat{\delta(t_l)\ \delta(t_r)}) & \text{if } t = \widehat{t_l\ t_r}, |t_l| > 0, |t_r| > 0 \end{cases}$$

Note that $\forall t.\ t \cong \delta(t)$. If $\tau = \delta(t)$ then we say τ is the *canonical form* of t.

Canonicalization is used to define the partial operator *join*, $\oplus : \mathbb{T} \times \mathbb{T} \to \mathbb{T}$, which states how two tree shares can be combined:

Definition 5 (Join operator).

$$\tau \oplus \tau' \quad \triangleq \quad \begin{cases} \tau & \text{if } \tau' = \circ \\ \tau' & \text{if } \tau = \circ \\ \bullet & \text{if } (\tau, \tau') \in \{(\bullet, \circ), (\circ, \bullet)\} \\ \delta(\widehat{\tau_1\ \tau_2}) & \text{if } \tau = \widehat{\tau_l\ \tau_r}, \tau' = \widehat{\tau_l'\ \tau_r'}, \tau_l \oplus \tau_l' = \tau_1, \tau_r \oplus \tau_r' = \tau_2 \\ \text{undefined otherwise} \end{cases}$$

$\oplus$ is a partial function because *e.g.* $\bullet \oplus \bullet$ and $\widehat{\bullet\ \circ} \oplus \widehat{\bullet\ \circ}$ are undefined.

A.2 From §4.1 : definition of tree operators

Here are the definitions of three operators: left (right) rounding $\overleftarrow{\lfloor\rfloor}, \overrightarrow{\lfloor\rfloor} : \mathbb{T}\times\mathbb{N}\to\mathbb{T}$ and average $\bigtriangledown : \mathbb{T}\times\mathbb{T}\times\mathbb{N}\to\mathbb{T}$. Their constructions make use of three sub-functions σ, ω and γ. Let $\sigma : \mathbb{T}\times\mathbb{N}\to\mathbb{B}$ be a partial function that takes $\tau\in\mathbb{T}$ and $n\in\mathbb{N}$ then *expands* τ into a full boolean binary tree t' of height n, $\omega : \mathbb{B}\to\mathbb{B}\times\mathbb{B}$ that *splits* a tree t into 2 trees (t_l, t_r) such that $t_l(t_r)$ contains only *left*(*right*) leaves of t, and $\gamma : \mathbb{B}\times\mathbb{B}\to\mathbb{B}$ be the function that *combines* leaves of two trees t_1, t_2 pairwise such that the result tree has left leaves from t_1 and right leaves from t_2. $\lfloor\overleftarrow{\tau}\rfloor_n(\lfloor\overrightarrow{\tau}\rfloor_n)$ expands τ to height n using σ, produces the *left tree*(*right tree*) using ω and the result tree is reduced to its canonical form by δ. $\tau_1 \bigtriangledown_n \tau_2$ expands τ_1, τ_2 to height $n-1$ using σ then combines them via γ and applies δ to reduce the tree to its canonical form. It is noticed that $(\overleftarrow{\lfloor\rfloor}, \overrightarrow{\lfloor\rfloor}, \bigtriangledown)$ are partial due to σ. Lastly, we define the *decompose function* (and its inverse), $\phi : \mathbb{T}\to\mathbb{T}\times\mathbb{T}$, that returns the left and right sub-tree.

Definition 6 (Tree Operators [13]).

$$\sigma(\tau,n)=\begin{cases}\tau & \text{if } \tau\in\{\circ,\bullet\}\wedge n=0\\ \overbrace{\sigma(\tau,n')\ \sigma(\tau,n')} & \text{if } \tau\in\{\circ,\bullet\}\wedge n=n'+1\\ \overbrace{\sigma(\tau_l,n')\ \sigma(\tau_r,n')} & \text{if } \tau=\widehat{\tau_l\ \tau_r}\wedge n=n'+1\\ \text{undefined} & \text{otherwise}\end{cases}$$

$$\omega(t)=\begin{cases}(t',t') & \text{if } t'\in\{\circ,\bullet\}\\ (\widehat{t'_{1,l}\ t'_{2,l}}, \widehat{t'_{1,r}\ t'_{2,r}}) & \text{if } t'=\widehat{t'_1\ t'_2}\wedge\omega(t'_1)=(t'_{1,l},t'_{1,r})\wedge\omega(t'_2)=(t'_{2,l},t'_{2,r})\end{cases}$$

$$\gamma(t,t')=\begin{cases}\widehat{t\ t'} & \text{if } t,t'\in\{\circ,\bullet\}\\ \overbrace{\gamma(t,t'_l)\ \gamma(t,t'_r)} & \text{if } t\in\{\circ,\bullet\}\wedge t'=\widehat{t'_l\ t'_r}\\ \overbrace{\gamma(t_l,t')\ \gamma(t_r,t')} & \text{if } t'\in\{\circ,\bullet\}\wedge t=\widehat{t_l\ t_r}\\ \overbrace{\gamma(t_l,t'_l)\ \gamma(t_r,t'_r)} & \text{if } t=\widehat{t_l\ t_r}\wedge t'=\widehat{t'_l\ t'_r}\end{cases}$$

$$\lfloor\overleftarrow{\tau}\rfloor_n=\begin{cases}\delta(t_l) & \text{if } \omega(\sigma(\tau,n))=(t_l,t_r)\\ \text{undefined} & \text{otherwise}\end{cases}$$

$$\lfloor\overrightarrow{\tau}\rfloor_n=\begin{cases}\delta(t_r) & \text{if } \omega(\sigma(\tau,n))=(t_l,t_r)\\ \text{undefined} & \text{otherwise}\end{cases}$$

$$\tau\bigtriangledown_n\tau'=\begin{cases}\delta(t'') & \text{if } \sigma(\tau,n)=t\wedge\sigma(\tau',n)=t'\wedge\gamma(t,t')=t''\\ \text{undefined} & \text{otherwise}\end{cases}$$

$$\phi(\tau)=\begin{cases}(\tau,\tau) & \text{if } \tau\in\{\circ,\bullet\}\\ (\tau_l,\tau_r) & \text{if } \tau=\widehat{\tau_l\ \tau_r}\end{cases}$$

$$\phi^{-1}(\tau,\tau')=\begin{cases}\tau & \text{if } \tau=\tau'\in\{\circ,\bullet\}\\ \widehat{\tau\ \tau'} & \text{otherwise}\end{cases}$$