Decision Procedures Over Sophisticated Fractional Permissions

Le Xuan Bach Cristian Gherghina* Aquinas Hobor**

National University of Singapore

Abstract. Fractional permissions enable sophisticated management of resource accesses in both sequential and concurrent programs. Entailment checkers for formulae that contain fractional permissions must be able to reason about said permissions to verify the entailments. We show how entailment checkers for separation logic with fractional permissions can extract equation systems over fractional shares. We develop a set decision procedures over equations drawn from the sophisticated boolean binary tree fractional permission model developed by Dockins *et al.* [4]. We prove that our procedures are sound and complete and discuss their computational complexity. We explain our implementation and provide benchmarks to help understand its performance in practice. We detail how our implementation has been integrated into the HIP/SLEEK verification toolset. We have machine-checked proofs in Coq.

1 Introduction

Separation logic is fundamentally a logic of resource accounting [13]. Control of some resource (*i.e.*, a cell of memory) allows the owner to take certain actions with that resource. Traditionally, ownership is a binary property, with full ownership associated with complete control (*e.g.*, the ability to read, modify, and deallocate the cell), and empty ownership associated with no control.

Many programs, particularly many concurrent programs, are not easy to verify with such a coarse understanding of access control [2, 1]. Fractional permissions track ownership—*i.e.*, access control—at a finer level of granularity. For example, partial ownership might allow for reading, while full ownership might in addition enable writing and deallocation. This access control scheme helps verify concurrent programs that allow multiple threads to share read access to heap cells as long as no thread has write access.

A share model defines the representation for fractions π (*e.g.*, a rational number between 0 and 1) and a three-place join relation $\oplus$ that combines them (*e.g.*, addition, allowed only when the sum is no more than 1). The join relation must satisfy a number of technical properties such as functionality, associativity, and commutativity. The fractional π-ownership of the memory cell ℓ, whose value is currently v, can then be written in separation logic as $\ell \overset{\pi}{\mapsto} v$. When π is full

* Supported by MoE Tier-2 grant MOE2009-T2-1-063
** Supported by a Lee Kuan Yew Postdoctoral Fellowship (T1-251RES0902)

ownership we simply omit it. We modify the standard separating conjunction $\star$ to respect fraction permissions via the equivalence $\ell \overset{\pi_1 \oplus \pi_2}{\mapsto} v \;\Leftrightarrow\; \ell \overset{\pi_1}{\mapsto} v \star \ell \overset{\pi_2}{\mapsto} v$.

Unfortunately, while they are very intuitive, rational numbers are not a good model for fractional ownership. Consider the following attempt at a recursively defined predicate for fractionally-owned binary trees:

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

This tree predicate is obtained directly from the standard recursive predicate for wholly-owned 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 $\pi \leq 0.5$, then $\mathsf{tree}_{\mathbb{Q}}$ can describe some non-tree directed acyclic graphs.

Parkinson then developed a share model that avoided this problem, but at the cost of certain other technical shortcomings and a total loss of decidability (even for equality testing) [12]. Decidability is crucial for developing automated tools to reason about separation logic formulae containing fractional permissions. Finally, Dockins *et al.* then developed a tree-share model detailed in §3 that overcame the technical shortcomings in Parkinson's model and in addition enjoyed a decidable test for equality and a computable join relation $\oplus$ [4]. The pleasant theoretical properties of the tree-share model led to its use in the design and soundness proofs of several flavors of concurrent separation logic [7, 8], and the basic computability results led to its incorporation in two program verification tools: HIP/SLEEK [11] and Heap-Hop [15].

However, it is one thing to incorporate a technique into a verification tool, and another thing to make it complete enough to work well. Heap-Hop employed a simplistic heuristic to prove entailments involving tree shares [14], and although HIP/SLEEK did better, the techniques were still highly incomplete [9]. Even verifying small programs can require hundreds of share entailment checks, so in practice this incompleteness was a significant barrier to the use of these tools to reason about programs whose verification required fractional shares.

Our work overcomes this barrier. We show how to extract a system of equations over shares from separation logic formulae such that the truth of the system is equivalent to the truth of the share portion of the formulae. This extraction can be done with no knowledge about the underlying model for shares. These systems of equations are then handed to our solver: a standalone library of sound and complete decision procedures over fraction tree shares. Although the worst-case complexity is high, our benchmarks demonstrate that our library is fast enough in practice to be incorporated into practical entailment checkers.

Contributions.

- We demonstrate how to extract a system of equations over fractional shares from separation logic formulae (§2).
- We prove that the key problems over these systems are decidable.
- We develop a tool that solves the problems and benchmark its performance.
- We incorporated our tool into the HIP/SLEEK verification toolset.
- Our prototype is available at:
  `www.comp.nus.edu.sg/~cristian/projects/prover/shares.html`

2 Extracting Shares from Separation Logic Formulae

Program verification tools, such as HIP, usually do not verify programs on their own. Instead, a program verifier 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 formulae containing such shares to its entailment checker. The entailment checker needs to know a bit more: how to separate share information from formulae 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 new backend share prover (detailed in §4).

To demonstrate that the entailment checker can treat the shares abstractly, we defer the share model until §3, and will first outline the extraction of systems of equations over shares from separation logic formulae. 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. Our actual domain is more sophisticated and is given in §3, 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 subdomain 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 [9, §8.4]. Extracting share information from more complex formulae depends on the exact nature of said formulae but usually follows the pattern we give here in a straightforward way; the end result is just larger systems of equations.

The logic formulae we will consider here are of the following form:

$$\begin{array}{llll} \Phi & := \exists\, v.\kappa \wedge \beta \;\mid\; \kappa \wedge \beta & \kappa & := \kappa * \kappa \;\mid\; v \overset{\pi}{\mapsto} v \\ \beta & := \beta \wedge \beta \;\mid\; v = \pi \;\mid\; \pi \oplus \pi = \pi & \pi & := v \mid \chi \end{array}$$

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 formulae are normalized in order to ensure that the heap component does not contain two distinct points-to predicates when the pointers are provably aliased. This reduction step can be described as:

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

Second, formulae 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[1].

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.

After the heap constraints are discharged, the share relevant portion of the entailment consists of sets of formulae over **non empty** shares with the syntax:

$$\phi \ ::= \ \exists v.\phi \mid \phi_1 \wedge \phi_2 \mid \pi_1 \oplus \pi_2 = \pi_3 \mid v_1 = v_2 \mid v = \chi$$

That is, share formulae ϕ contain share variables v, existential binders $\exists$, conjunctions $\wedge$, join facts $\oplus$, equalities between variables, and assignments of variables to constants χ. 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 [9] we have not needed arbitrary nesting of quantifiers. We will view the share formulae as equation systems Σ, *i.e.* as a pair of sets: (1) a set of equations of the form $a \oplus b = c$ or $v = w$, and (2) a set of existentially quantified variables.

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

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

First, the two entailments need to be normalized and the shares floated out[2]:

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

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.

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

[1] We are almost always able to avoid a serious exponential search by using the search prunings described in [9].

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

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, Heap-Hop, and coreStar) 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.

We have now given two examples of extracting share equations from separation logic formulae. Once the translation is finished, a separation logic entailment checker can ask our share prover two questions:

1. (SAT) A *solution* S of Σ is a (finite) mapping from the variables of Σ into tree shares. We say that a solution S satisfies the equation system Σ, written $S \models \Sigma$, when the mapping makes the equations and equalities in Σ true. The SAT query asks if an equation system Σ is satisfiable, *i.e.*, $\exists S.\ S \models \Sigma$? SLEEK uses SAT checks to help prune unfeasible proof searches.
2. (IMPL) Given two systems Σ_a and Σ_c, does Σ_a entail Σ_c? That is: $\Sigma_a \vdash \Sigma_c$ iff $\forall S.\ S \models \Sigma_a \Rightarrow S \models \Sigma_c$.

In practice this is sufficient; in §4 we will detail how we answer these questions.

3 Binary Boolean Trees as a Fractional Share Model

Here we briefly explain the tree-share fractional permissions model of Dockins *et al.* [4]. A tree share τ is inductively defined as a binary tree with boolean leaves:

$$\tau ::= \circ \mid \bullet \mid \tau\ \tau$$

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: $\circ\ \bullet$ and $\bullet\ \circ$, and four "quarter" shares, beginning with $\bullet\ (\circ\ \circ)$. Notice that the two half shares are not identical; this is a feature, not a bug: this property ensures that the definition of tree from equation (1) really describes fractional trees instead of DAGs.

Notice also that we presented the first quarter share as $\bullet\ (\circ\ \circ)$ instead of $\bullet\ (\circ\ (\circ\ \circ))$. 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 $\cong$:

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

The canonical representation is needed to guarantee some of the technical properties described below. Managing the canonicality is a minor performance cost for the computable parts of our system but a major technical hassle in the proofs. Our strategy for this presentation is to gloss over some of these details, folding and unfolding trees into canonical form when required by the narrative. We justify our informalism in the presentation because all of the operations we define on trees have been verified in Coq to respect the canonicality.

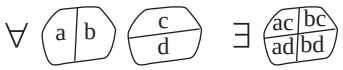
Functional:	$x \oplus y = z_1 \;\Rightarrow\; x \oplus y = z_2 \;\Rightarrow\; z_1 = z_2$
Commutative:	$x \oplus y \;=\; y \oplus x$
Associative:	$x \oplus (y \oplus z) \;=\; (x \oplus y) \oplus z$
Cancellative:	$x_1 \oplus y = z \;\Rightarrow\; x_2 \oplus y = z \;\Rightarrow\; x_1 = x_2$
Unit:	$\exists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

The join relation for trees is inductively defined by unfolding both trees to the same shape and joining leafwise using the rules $\circ \oplus \circ = \circ$, $\circ \oplus \bullet = \bullet$, and $\bullet \oplus \circ = \bullet$; afterwards the result is refolded into canonical form as in this example:

Because $\bullet \oplus \bullet$ is undefined, the join relation on trees is a partial operation. Dockins *et al.* prove that the join relation satisfies a number of useful properties detailed in Figure 1. The tree share model is the only model that simultaneously satisfies Disjointness (forces the tree predicate—equation 1— to behave properly), Cross-split (used *e.g.* in settings involving overlapping data structures), and Infinite splittability (to verify divide-and-conquer algorithms). In the domain of tree shares, Disjointness is equivalent to $x \oplus x = y \;\Rightarrow\; x = \circ$; the name Disjointness comes from a related axiom at the level of formulae by Parkinson.

Unfortunately, while the $\oplus$ operation has many nice properties useful for verifying programs, they fall far short of those necessary to permit traditional algebraic techniques like Gaussian elimination. Dockins also defines a kind of multiplicative operation $\bowtie$ between shares used to manage a token counting setting (as opposed to the divide-and-conquer algorithms we can verify), but our decision procedures do not support $\bowtie$ at this time.

4 Decision Procedures over Tree Shares

Here we introduce a decision procedure for discharging tree share proof obligations generated by program verifiers. Recall from §2 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 the system Σ is satisfied by solution S; the SAT query is then simply $\exists S.S \models \Sigma$. Furthermore, we write $\Sigma_a \vdash \Sigma_c$ to mean that every solution S that satisfies Σ_a also satisfies Σ_c, *i.e.*, $\forall S.\ S \models \Sigma_a \Rightarrow S \models \Sigma_c$; this is exactly the IMPL query.

```
REDUCE(Σ)
  Σ′ = SIMPLIFY(Σ)
  If (|Σ′| = 0)
    Return FORMULA(Σ′)
  Else
    (Σ^l, Σ^r) = DECOMPOSE(Σ′)
    Φ = REDUCE(Σ^l)∧REDUCE(Σ^r)
    Return Φ

SAT(Σ)
  Φ = REDUCE(Σ)
  Return SMT_SOLVER(Φ)
```

Fig. 2. SAT

```
REDUCEI(Σ_a, Σ_b)
  Σ′_a = SIMPLIFY(Σ_a)
  Σ′_c = SIMPLIFY(Σ_c)
  If (|Σ′_a| = 0 ∧ |Σ′_c| = 0)
    Return (FORMULA(Σ′_a), FORMULA(Σ′_c))
  Else
    (Σ^l_a, Σ^r_a) = DECOMPOSE(Σ′_a)
    (Σ^l_c, Σ^r_c) = DECOMPOSE(Σ′_c)
    (Φ^l_a, Φ^l_c) = REDUCEI(Σ^l_a, Σ^l_c)
    (Φ^r_a, Φ^r_c) = REDUCEI(Σ^r_a, Σ^r_c)
    Return(Φ^l_a ∧ Φ^r_a,  Φ^l_c ∧ Φ^r_c)

IMPL(Σ_a, Σ_c)
  (Φ_a, Φ_c) = REDUCEI(Σ_a, Σ_c)
  Return ¬ SMT_SOLVER(Φ_a ∧ ¬Φ_c)
```

Fig. 3. IMPL

The key reason SAT and IMPL are nontrivial is that the space is dense[3]. 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 $\Sigma_a \vdash \Sigma_c$ 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 IMPL 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[4]. For solutions S, let $|S|$ be the highest constant in its codomain. In §5, we will prove our key theoretical result: that for both SAT and IMPL 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.

Of course, we do not want to blindly search through an exponentially large space if we can avoid it! Our goal for this section is to describe and prove sound the algorithms for SAT and IMPL given in Figures 2 and 3. The core of our decision procedures are the 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 and Z3. The other three subroutines are discussed in detail below.

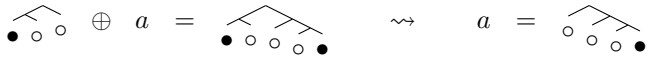
SIMPLIFY. SAT/SMT solvers can require a lot of computation time. Accordingly, SIMPLIFY attempts reduce the size of the problem with a combination of several techniques. First, each equation that contains two or three tree constants is simplified into an equality (or $\top/\bot$). To do so, SIMPLIFY sometimes 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

[3] 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.

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

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 canonical form, *e.g.*,

$\ominus$ is needed when one of the constants appears on the RHS of an equation, *e.g.*,[5]



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:

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

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 1. *For all solutions S, $S \models \Sigma$ iff $S \models \mathsf{SIMPLIFY}(\Sigma)$.*

We will also need to know that SIMPLIFY does not increase the height of an equation system. To prove this, we need the following fact about $\oplus$ and $\ominus$:

Lemma 2. *If $a \oplus b = c$ or $a \ominus b = c$ then $|c| \leq \mathsf{max}(|a|, |b|)$.*

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

Lemma 3. $|\mathsf{SIMPLIFY}(\Sigma)| \leq |\Sigma|$

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. 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:

$$\begin{array}{llll} v \rightsquigarrow (v_l, v_r) & & & \textit{vars} \\ \circ \rightsquigarrow (\circ, \circ) & \bullet \rightsquigarrow (\bullet, \bullet) & \widehat{\tau_1\ \tau_2} \rightsquigarrow (\tau_1, \tau_2) & \textit{consts} \\ \left.\begin{array}{l} a \rightsquigarrow (a_l, a_r) \\ b \rightsquigarrow (b_l, b_r) \\ c \rightsquigarrow (c_l, c_r) \end{array}\right\} & \begin{array}{ll} a \oplus b = c & \rightsquigarrow (a_l \oplus b_l = c_l,\ a_r \oplus b_r = c_r) \\ a = b & \rightsquigarrow (a_l = b_l,\ a_r = b_r) \end{array} & & \textit{eqs} \end{array}$$

[5] In §4 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.

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

Lemma 5. *If* $\mathsf{DECOMPOSE}(\Sigma) = (\Sigma_l, \Sigma_r)$*, then* $|\Sigma| > \mathsf{max}(|\Sigma_l|, |\Sigma_r|)$ *or we were at height 0 to begin with,* i.e.*,* $|\Sigma| = |\Sigma_l| = |\Sigma_r| = 0$.

FORMULA. 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 formulae. Accordingly, FORMULA translates the equations and equalities in a system of equations of height zero into logic formulae as follows:

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

Each resulting formula is $\wedge$-conjoined together to get a single formula that represents the entire system, as indicated by the following lemma:

Lemma 6. *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 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.

SAT. We are almost ready to prove the correctness of the SAT function. The last puzzle piece we need is one of the two major theoretical insights of this paper:

Theorem 1. *Σ is satisfiable if and only if Σ can be satisfied with a solution S whose height is* $|\Sigma|$*,* i.e.*,* $\exists S.\ S \models \Sigma$ *iff* $\exists S.\ |S| = |\Sigma| \wedge S \models \Sigma$.

We will defer the proof of Theorem 1 until §5.1; our task in this section is to show how it fits into our correctness proof for SAT, *i.e.*,

Theorem 2. $\mathsf{SAT}(\Sigma) = \top$ *if and only if* 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 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 1, Σ' is satisfiable iff Σ was satisfiable; moreover, by Lemma 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 6 is equivalent to Σ' as long as the solution has height 0. Theorem 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 DECOMPOSE Σ' into Σ_{l} and Σ_{r}. Lemma 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 completes the inductive step by telling us that the conjunction of Σ^{l} and Σ^{r} is equivalent to Σ'. □

IMPL. We need the second major theoretical insight of this paper to verify IMPL.

Theorem 3. $\Sigma_a \vdash \Sigma_c$ *iff* $\Sigma_a \vdash \Sigma_c$ *for all solutions* S *s.t.* $|S| = |\Sigma_a|$, i.e., $\forall S.\ S \models \Sigma_a \Rightarrow S \models \Sigma_c$ *iff* $\forall S.\ |S| = |\Sigma_a| \Rightarrow S \models \Sigma_a \Rightarrow S \models \Sigma_c$.

We will defer the proof until §5.2; just as we did with Theorem 1 above, our task here is to show how Theorem 3 fits into our correctness proof for IMPL, *i.e.*,

Theorem 4. $\mathsf{IMPL}(\Sigma_a, \Sigma_c)$ *if and only if* $\Sigma_a \vdash \Sigma_c$, i.e., $\forall S.\ S \models \Sigma_a \Rightarrow S \models \Sigma_c$.

Proof. The major effort is proving that REDUCEI correctly transforms Σ_a and Σ_c into equivalent logical formulae Φ_a and Φ_c such that $\Sigma_a \vdash \Sigma_c$ iff $\Phi_a \Rightarrow \Phi_c$; afterwards we simply use the standard SAT/SMT solver trick of converting a validity check for $\Phi_a \Rightarrow \Phi_c$ into an **un**satisfiability check for $\Phi_a \wedge \neg\Phi_c$.

The proof of REDUCEI is largely in parallel with the proof of REDUCE in Theorem 2. We proceed by complete induction, this time on $\mathsf{max}(|\Sigma_a|, |\Sigma_c|)$. Again the base and inductive cases begin in the same way. We apply SIMPLIFY to reach Σ'_a and Σ'_c and again use Lemma 1 to guarantee that $\Sigma'_a \vdash \Sigma'_c$ iff $\Sigma_a \vdash \Sigma_c$; Lemma 3 ensures that $\mathsf{max}(|\Sigma'_a|, |\Sigma'_c|) \leq \mathsf{max}(|\Sigma_a|, |\Sigma_c|)$.

After simplification, the base and inductive cases diverge. In the base case, $\mathsf{max}(|\Sigma'_a|, |\Sigma'_c|) = 0$ and we call FORMULA to reach two logical formulae, the first equivalent to Σ'_a and the second equivalent to Σ'_c, as long as the solutions are of height zero (Lemma 6). Theorem 3 completes the base case by observing that it is sufficient to check only the solutions of height $|\Sigma'_a|$, *i.e.* zero.

In the inductive case, we DECOMPOSE Σ'_a and Σ'_c into $\Sigma^{\mathsf{l}}_{\mathsf{a}}$, etc., decreasing the maximum of their heights (Lemma 5), and thus letting us use the induction hypothesis for the recursive calls. Afterwards, we have four formulae ($\Phi^{\mathsf{l}}_{\mathsf{a}}$, etc.); we then conjoin both antecedents and both consequents using Lemma 4. □

Optimizations. The algorithms presented in figures 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 include dropping equalities after substitution and a lazier on-demand version of DECOMPOSE. In addition to utilizing the lazier version of DECOMPOSE, optimizations during IMPL 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 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.*, $\mathsf{max}(|S(v_1)|, |S(v_2)|) < |\chi|$). By Lemma 2 we also know that $|\chi| \leq \mathsf{max}(|S(v_1)|, |S(v_2)|)$, 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. For future work we would like to develop a verified implementation.

Complexity. One might wonder what complexity class SAT and IMPL belong to. Tree-SAT when restricted to systems of height zero is already NP-COMPLETE.

Proof. We can use the following clause-by-clause reduction from 3-SAT, in which new variables (X, Y, Z, M, and N) are chosen fresh for each disjunctive clause:

$$\begin{array}{rcl}
A \vee B \vee C & \rightsquigarrow & (A \oplus X = \bullet) \wedge (X \oplus Y = Z) \wedge (B \oplus M = Z) \wedge (M \oplus N = C) \\
\neg A \vee B \vee C & \rightsquigarrow & (A \oplus X = Y) \wedge (B \oplus Z = Y) \wedge (Z \oplus M = C) \\
\neg A \vee \neg B \vee C & \rightsquigarrow & (A \oplus X = Y) \wedge (B \oplus Z = M) \wedge (C \oplus X = M) \\
\neg A \vee \neg B \vee \neg C & \rightsquigarrow & (A \oplus X = Y) \wedge (B \oplus Z = M) \wedge (X \oplus Z = C)
\end{array}$$

The clause on the LHS is satisfiable iff the clause on the RHS is satisfiable. □

We hypothesize that tree-SAT on systems of arbitrary height is still "only" NP-COMPLETE because our SAT algorithm seems to scale the formulae polynomially with the description of the system. Going a bit further onto a limb, we further hypothesize that tree-IMPL is no worse than $\mathrm{NP}^{\mathrm{NP}}$-COMPLETE. Happily, as we show in §7, performance seems to be adequate in practice.

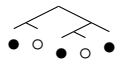
5 Sufficiency of Finite Search over Tree Shares

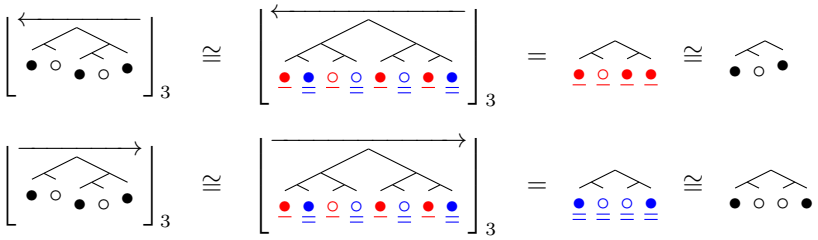
The SAT and IMPL algorithms presented in §4 are basically doing a shape-guided search through a finite domain. Our key theoretical insight is that a finite

search is sufficient, as formalized in the statement of Theorems 1 and 3 in §4. Our next task is to prove these theorems, which is the focus of the remainder of this section. The most technical parts—Lemmas 8 and 10—have been mechanically verified in Coq. The remaining proofs have been carefully checked on paper.

5.1 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 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 8 (Properties of $\lfloor \overleftarrow{\tau} \rfloor_n$ and $\lfloor \overrightarrow{\tau} \rfloor_n$).

1. *If* $n > |\tau|$ *then* $\lfloor \overleftarrow{\tau} \rfloor_n = \lfloor \overrightarrow{\tau} \rfloor_n = \tau$
2. *If* $n = |\tau|$, $\tau_l = \lfloor \overleftarrow{\tau} \rfloor_n$, *and* $\tau_r = \lfloor \overrightarrow{\tau} \rfloor_n$ *then* $\mathsf{max}(|\tau_l|, |\tau_r|) < n$
3. *If* $\tau_1 \oplus \tau_2 = \tau_3$ *and* $n = \mathsf{max}(|\tau_1|, |\tau_2|, |\tau_3|)$, *then* $\lfloor \overleftarrow{\tau_1} \rfloor_n \oplus \lfloor \overleftarrow{\tau_2} \rfloor_n = \lfloor \overleftarrow{\tau_3} \rfloor_n$ *and* $\lfloor \overrightarrow{\tau_1} \rfloor_n \oplus \lfloor \overrightarrow{\tau_2} \rfloor_n = \lfloor \overrightarrow{\tau_3} \rfloor_n$

Proved in Coq. Lemma 8 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 the height of the equation.

We extend $\lfloor \overleftarrow{\cdot} \rfloor_n$ and $\lfloor \overrightarrow{\cdot} \rfloor_n$ to work over solutions S pointwise 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 8 after using a solution S to instantiate variables in a system Σ.

Lemma 9. *Let* $S \models \Sigma$, $n = |S| > |\Sigma|$, $S_l = \lfloor \overleftarrow{S} \rfloor_n$, *and* $S_r = \lfloor \overrightarrow{S} \rfloor_n$. *Then* $S_l \models \Sigma$, $S_r \models \Sigma$, *and* $\mathsf{max}(|S_l|, |S_r|) < n$.

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 1.

Theorem 1. *Σ is satisfiable if and only if Σ can be satisfied with a solution S whose height is $|\Sigma|$,* i.e., *$\exists S.\ S \models \Sigma$ iff $\exists S.\ |S| = |\Sigma| \wedge S \models \Sigma$.*

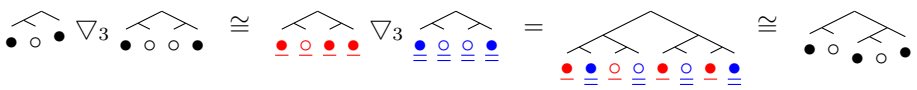
Proof. $\Leftarrow$: Immediate. $\Rightarrow$: Suppose $S \models \Sigma$. By Lemma 7, we have $|S| \geq |\Sigma|$, *i.e.*, $|S| = |\Sigma| + n$ for some n. We proceed by strong induction on n. If $n = 0$ we are done. Otherwise, by Lemma 9 we know that $S_l = \lfloor \overleftarrow{S} \rfloor_{|\Sigma|+n}$ satisfies Σ and $|S_l| < |S|$, letting us apply the induction hypothesis. □

5.2 The Sufficiency of Finite Search for IMPL

IMPL is more complicated than SAT due to the contravariance. Suppose we have computationally checked that all solutions S of height $|\Sigma_a|$ that satisfy Σ_a also satisfy Σ_c. Now suppose that $S \models \Sigma_a$ for some S such that $|S| = |\Sigma_a| + 1$, and we wish to know if $S \models \Sigma_c$. Lemma 9 tells us that $\lfloor \overleftarrow{S} \rfloor_{|\Sigma_a|+1} \models \Sigma_a$. Our computational verification then tells us that $\lfloor \overleftarrow{S} \rfloor_{|\Sigma_a|+1} \models \Sigma_c$, but then we are stuck: on its own, $\lfloor \overleftarrow{S} \rfloor_{|\Sigma_a|+1} \models \Sigma_c$ is too weak to prove $S \models \Sigma_c$.

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.

We illustrate with another example, highlighting again with red and blue:



Lemma 10 (Properties of $\tau_l \bigtriangledown_n \tau_r$).

1. *If $n > |\tau|$ then $\tau \bigtriangledown_n \tau = \tau$.*
2. *If $n \geq |\tau|$, then $\lfloor \overleftarrow{\tau} \rfloor_n \bigtriangledown_n \lfloor \overrightarrow{\tau} \rfloor_n = \tau$.*
3. *If $n > \mathsf{max}(|\tau_1|, |\tau_2|, |\tau_3|, |\tau_1'|, |\tau_2'|, |\tau_3'|)$, $\tau_1 \oplus \tau_2 = \tau_3$, and $\tau_1' \oplus \tau_2' = \tau_3'$, then $(\tau_1 \bigtriangledown_n \tau_1') \oplus (\tau_2 \bigtriangledown_n \tau_2') = (\tau_3 \bigtriangledown_n \tau_3')$.*

Proved in Coq. 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 $\oplus$.

Given a system Σ, Lemma 10 contains the facts we need to prove that the $\bigtriangledown_n$-combination 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 11 (Properties of $S_l \bigtriangledown_n S_r$).

1. *For all S, if $n \geq |S|$ then $\lfloor \overleftarrow{S} \rfloor_n \bigtriangledown_n \lfloor \overrightarrow{S} \rfloor_n = S$.*
2. *Let S_l, S_r be solutions of Σ and $n > \mathsf{max}(|S_l|, |S_r|)$. Then $S = S_l \bigtriangledown_n S_r$ is a solution of Σ.*

Direct from Lemma 10. We are now ready to attack the main IMPL theorem.

Theorem 3. *$\Sigma_a \vdash \Sigma_c$ iff $\Sigma_a \vdash \Sigma_c$ for all solutions S s.t. $|S| = |\Sigma_a|$,* i.e., *$\forall S.\ S \models \Sigma_a \Rightarrow S \models \Sigma_c$ iff $\forall S.\ |S| = |\Sigma_a| \Rightarrow S \models \Sigma_a \Rightarrow S \models \Sigma_c$.*

Proof. $\Rightarrow$: Immediate. $\Leftarrow$: We apply complete induction, starting from $|\Sigma_a|$, on the height of solutions S of Σ_a. The base case ($|S| = |\Sigma_a|$) is immediate. For the inductive case, we know $S \models \Sigma_a$ and that all solutions S' of Σ_a such that $|S'| < |S|$ are also solutions of Σ_c. By Lemma 9, we know that $\lfloor \overleftarrow{S} \rfloor_{|S|}$ and $\lfloor \overrightarrow{S} \rfloor_{|S|}$ are both solutions to Σ_a 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_c. Lemma 11 completes the proof by telling us that $\lfloor \overleftarrow{S} \rfloor_{|S|} \bigtriangledown_{|S|} \lfloor \overrightarrow{S} \rfloor_{|S|} = S$ is also a solution of Σ_c. $\square$

6 Handling Non-zeros

For fear of cluttering the presentation we omitted showing how to restrict a variable to non-zero shares in SAT and IMPL queries.

However, our methods are able to handle this detail: each system of equations also contains a list of "non-zero" variables. This list is taken into account when constructing the first-order boolean formula: for each non-zero variable, an extra disjunctive clause over all the decompositions of that variable is generated. This forces at least one of the boolean variables corresponding to the initial non-zero variable to be true in each solution. In the tree domain, this clause ensures that the non-zero variable has at least one $\bullet$ leaf.

The full algorithms have an extra call to the NON_ZERO function, which returns a conjunction of the clauses encoding the non-zero disjunctions (not shown: sometimes we need to lift existentials to the top level). To force a variable v to be strictly positive at least one of the variables decomposed from v needs to be true (*i.e.*, the tree value has at least one $\bullet$ leaf). If v_i is the set of variables obtained from decomposing v then positivity is encoded by $\bigvee_i v_i$.

$\mathsf{SAT}(\Sigma)$
  $\Phi = \mathsf{REDUCE}(\Sigma)$
  $\Phi_\mathsf{r} = \Phi \wedge \mathsf{NON\_ZERO}(\Sigma)$
  Return $\mathsf{SMT\_SOLVER}(\Phi_\mathsf{r})$

$\mathsf{IMPL}(\Sigma_\mathsf{a}, \Sigma_\mathsf{c})$
  $(\Phi_\mathsf{a}, \Phi_\mathsf{c}) = \mathsf{REDUCEI}(\Sigma_\mathsf{a}, \Sigma_\mathsf{c})$
  $\Phi'_\mathsf{a} = \Phi_\mathsf{a} \wedge \mathsf{NON\_ZERO}(\Sigma_1)$
  $\Phi'_\mathsf{c} = \Phi_\mathsf{c} \wedge \mathsf{NON\_ZERO}(\Sigma_2)$
  Return $\neg\ \mathsf{SMT\_SOLVER}(\Phi'_\mathsf{a} \wedge \neg\Phi'_\mathsf{c})$

Because these *non-zero* constraints relate otherwise disjoint equation subsystems to each other, it is not obvious how to verify each subsystem independently, which is why we produce one large boolean formula rather than many small ones.

Furthermore, the non-zero set forces extra system decompositions. To illustrate this point, observe that the equation $v_1 \oplus v_2 = \bullet$ has no solution of depth 0 in which both v_1 and v_2 are non-empty. However, decomposing the system once will yield the system: $v_1^l \oplus v_2^l = \bullet \wedge v_1^r \oplus v_2^r = \bullet$ with two possible solutions $(v_1^l = \circ; v_1^r = \bullet; v_2^l = \bullet; v_2^r = \circ)$ and $(v_1^l = \bullet; v_1^r = \circ; v_2^l = \circ; v_2^r = \bullet)$ which translate into $(v_1 = \overset{\frown}{\circ\ \bullet}; v_2 = \overset{\frown}{\bullet\ \circ})$ and $(v_1 = \overset{\frown}{\bullet\ \circ}; v_2 = \overset{\frown}{\circ\ \bullet})$. We have proved that for each non-zero variable, $\lceil log_2(n) \rceil$ is an upper bound on the number of extra decompositions, where n is the total number of variables. In practice we do not need to decompose nearly that much, and we have not noticed a meaningful performance cost. We speculate that we avoid most of the cost of the additional decompositions because the extra variables are often handled by some of the fast simplification rules we have incorporated into our tool.

7 Solver Implementation

Here we discuss some implementation issues. Our prototype is an OCaml library that implements (an optimized version of) the algorithms from §4 to resolve the SAT and IMPL 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. 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 [5] and Z3 [3]; 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 [9, §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 $\overset{\frown}{\circ\bullet} \oplus x = y$ it would deduce $\circ \leq x \leq \overset{\frown}{\bullet\circ}$ and $\overset{\frown}{\circ\bullet} \leq y$. The resulting highly 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 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 IMPL queries.

The first two test groups were developed for BndP in [9] 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 IMPL 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 IMPL 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 IMPL 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 IMPL; 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.

	SAT					IMPL				
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 1. Experimental timing results

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 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, 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 IMPLY queries, many of which we specifically designed to stress a share prover in various tricky ways, including heavily skewed tree constants, evil mixes of non-zero variables, 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 IMPL checks reached the underlying solver. BndP could solve fewer than 10% of the queries.

8 Related and Future Work

Simpler fractional permissions are used in a variety of logics [2, 1] and verification tools [10]. Their use is by no means restricted to separation logic as indicated by their use in CHALICE [6]. Despite the simpler domain, and associated loss of useful technical properties, we could find no completeness claims in the literate. It is our hope that other program verification tools will decide to incorporate more sophisticated share models now that they can use our solver.

In the future we would like to improve the performance of our tool by trying to mix the sound but incomplete bounds-based method [9] with the techniques described here; make a number of performance-related engineering enhancements, integrate the $\bowtie$ operation, and develop a mechanically-verified implementation.

9 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.

References

1. Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.
2. John Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72, 2003.
3. Leonardo de Moura and Nikolaj Bjrner. Z3: An efficient SMT solver. In *TACAS*, 2008.
4. Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, pages 161–177, 2009.
5. N. Een and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–508, 2003.
6. Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. Fractional permissions without the fractions. In *FTfJP*, 2011.
7. Aquinas Hobor. *Oracle Semantics*. PhD thesis, Princeton University, Department of Computer Science, Princeton, NJ, October 2008.
8. Aquinas Hobor and Cristian Gherghina. Barriers in concurrent separation logic. In *ESOP*, pages 276–296, 2011.
9. Aquinas Hobor and Cristian Gherghina. Barriers in concurrent separation logic: Now with tool support! *Logical Methods in Computer Science*, 8(2), 2012.
10. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frederic Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods*, volume 6617, pages 41–55, 2011.
11. 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.
12. Matthew Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.
13. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
14. Jules Villard. personal communication, 2012.
15. Jules Villard, Étienne Lozes, and Cristiano Calcagno. Tracking heaps that hop with Heap-Hop. In *TACAS*, pages 275–279, 2010.