A Certified Decision Procedure for Tree Shares

Xuan-Bach Le*, Thanh-Toan Nguyen*, Wei-Ngan Chin*, and Aquinas Hobor$^{+}_{*}$

*School of Computing and $^{+}$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 can reason about shared ownership of resources, *e.g.* in a concurrent program. We imported our certified procedure into the HIP/SLEEK verification system and found bugs in both the previous, uncertified, decision procedure and HIP/SLEEK itself. In addition to being certified, our new procedure improves previous work by correctly handling negative clauses and enjoys better performance.

1 Introduction

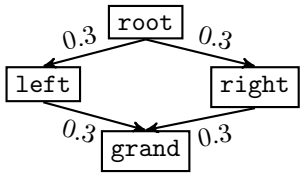
The last decade has enjoyed much progress in formal methods for concurrency in both theoretical understanding [14, 22, 24, 36, 37] and tool support [16, 21, 25, 26, 30, 35, 13]. Fractional shares enable reasoning about shared ownership of resources between multiple parties, *e.g.* in a concurrent program [7]. The original model for fractional shares was rational numbers in $[0, 1]$, with 0 representing no ownership, 1 representing full ownership, and $0 < x < 1$ representing partial ownership. A *policy* maps permission quanta to allowed actions. One simple policy maps 1 to the ability to both read and write a memory cell, $0 < x < 1$ to the ability to read—but not write—the cell, and 0 denying both reading and writing. We can prevent dangerous read/write and write/write data races by enforcing that the combined total ownership of each address is no more than 1.

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

$$\begin{array}{l}\mathsf{tree}(\ell,\pi) \stackrel{\text{def}}{=} (\ell = \mathsf{null} \ \wedge \mathsf{emp}) \ \vee \\ \quad \exists \ell_l, \ell_r.\ (\ell \stackrel{\pi}{\mapsto} (\ell_l, \ell_r) \star \mathsf{tree}(\ell_l, \pi) \star \mathsf{tree}(\ell_r, \pi))\end{array} \tag{1}$$

Here we write $a \stackrel{\pi}{\mapsto} b$ to indicate that memory location a contains value b and is owned with (positive/nonempty) share π. We can split and join ownership of a cell with addition: $a \stackrel{\pi_1}{\mapsto} b \star a \stackrel{\pi_2}{\mapsto} b \dashv\vdash a \stackrel{\pi_1 \oplus \pi_2}{\longmapsto} b$; note we use $\oplus$ instead of $+$ to indicate that the addition is bounded in [0,1] and thus partial (*e.g.* $0.6 \oplus 0.6$ is undefined). 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 $\pi \in (0, 0.5]$, then tree can describe some non-tree directed acyclic graphs such as the following:

$$\begin{array}{l} \texttt{root} \overset{0.3}{\mapsto} (\texttt{left}, \texttt{right}) \ * \\ \texttt{left} \overset{0.3}{\mapsto} (\mathsf{null}, \texttt{grand}) \ * \\ \texttt{right} \overset{0.3}{\mapsto} (\texttt{grand}, \mathsf{null}) \ * \\ \texttt{grand} \overset{0.6}{\mapsto} (\mathsf{null}, \mathsf{null}) \end{array}$$



This heap satisfies $\mathsf{tree}(\texttt{root}, 0.3)$ despite actually being a DAG (`grand` is owned with share $0.3 \oplus 0.3 = 0.6$).

Parkinson proposed a model based on sets of natural numbers that solved this issue but introduced others [33], and then Dockins *et al.* [15] proposed the following "tree share" model, which fixes all of the aforementioned issues. A tree share $\tau \in \mathbb{T}$ is inductively defined as a binary tree with boolean leaves:

$$\tau \triangleq \circ \mid \bullet \mid \tau \frown \tau$$

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: $\circ \frown \bullet$ and $\bullet \frown \circ$, and four "quarter" shares, beginning with $(\bullet \frown \circ) \frown \circ$. It is a feature, rather than a bug, that the two half shares are distinct from each other.

Notice that we presented the first quarter share as $(\bullet \frown \circ) \frown \circ$ instead of *e.g.* $\bullet \frown (\circ \frown \circ)$. 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 relation $\cong$:

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

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.

Defining the "join" operation $\oplus$ on tree shares formally is somewhat technical due to the necessity of managing the canonical forms [29, §A] 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:

$$(\bullet \frown \circ) \frown \circ \ \oplus \ \circ \frown (\bullet \frown \bullet) \ \cong \ \bullet \frown (\circ \frown \circ) \ \oplus \ \circ \frown (\bullet \frown \bullet) \ = \ \bullet \frown (\bullet \frown \bullet) \ \cong \ (\bullet \frown \bullet) \frown \circ$$

Because $\bullet \oplus \bullet$ is undefined, the join relation on trees is a partial operation. Dockins *et al.* [15] prove that the join relation satisfies a number of useful axioms *e.g.* associativity and commutativity (§2.1 has the full list). One key axiom, not satisfied by $(\mathbb{Q}, \oplus)$, is "disjointness": $x \oplus x = y \ \Rightarrow \ x = \circ$. Disjointness is the axiom that forces the tree predicate—equation 1—to behave properly: we saw above that we get a DAG in $\mathbb{Q}$ since $x \oplus x$ need not be 0.

Due to their good metatheorical properties, various program logics [19, 18] and tools [38, 21, 3] incorporate tree shares. Gherghina detailed a number of programs whose verifications used tree shares heavily [17, Ch.4]; these form the core

of our benchmark in §4.2. However, most tools have avoided using tree shares in part because they lacked algorithms that could decide entailments involving fractionals. Hobor and Gherghina [21] 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; this encouraged shares to be studied as a standalone domain.

Le *et al.* developed a tool to decide tree share entailments [27]. The present paper improves on their 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 implementation and underlying theory are rock solid. By comparing our new tool with Le *et al.*'s, we discovered weaknesses in both the latter's implementation and its theory. Moreover, a trend in recent years has been to develop verification toolsets within Coq [4, 10, 3]; since certified tools generally only depend on other certified tools, such tools have not been able to use Le *et al.*'s implementation, but they can use our new tool. Happily, despite the challenges involved in developing an implementation in Coq, our new tool exhibits improved performance over Le *et al.*'s due to a number of heuristics that meaningfully improve performance without sacrificing soundness or completeness; some of these heuristics should be applicable to future certified procedures.

From a theoretical point of view, our major improvement over Le *et al.* is a sound treatment of negations. Negative clauses in logic are often more difficult to handle than positive ones are. Le *et al.*'s theory purported to support a very limited form of negation, which allowed them to force variables to be nonempty, *i.e.* $\pi \neq \circ$. We believe the previous theory is unsound when there are a sufficiently high number of nonempty variables on both sides of an implication. Our new theory handles arbitrary negative clauses, *i.e.* $\neg(\pi_1 \oplus \pi_2 = \pi_3)$ and is fully mechanized in Coq. A second theoretical improvement is a more careful treatment of existential variables.

The rest of this paper is organized as follows. In §2 we define the central decision problem and give an overview of our procedure. In §3 we show the key algorithms and outline why they are correct. All our proofs are mechanized in Coq; additional pen-and-paper details are also available in our appendix [29]. In §4 we discuss our 38.6k LOC certified implementation, describe how we have incorporated it into the HIP/SLEEK verification toolset [32], and benchmark its performance. Finally, in §5 we discuss related and future work and conclude.

2 Share constraints and their decision procedures

In §2.1, we introduce the decision problems over tree shares, satisfiability and entailment over *share equation systems*. Next we overview our decision procedure in §2.2 together with a brief description of their components' functionality. For **convenience**, we will use the symbol $\mathcal{L}$ to represent $\overset{\frown}{\bullet\ \circ}$ and $\mathcal{R}$ for $\overset{\frown}{\circ\ \bullet}$.

2.1 Share constraints

Given a SL entailment $P \vdash Q$ with fractional permissions, there are standard procedures to separately extract a heap constraint and a share constraint [21,

17, 27]. For example, the entailment $x \overset{v_1}{\longmapsto} 1 \star y \overset{\mathcal{R}}{\longmapsto} 2 \vdash x \overset{\mathcal{L}}{\longmapsto} 1$ yields constraints $v_1 \neq \circ \wedge v_2 = \mathcal{R} \vdash \exists v_3.\ \mathcal{L} \oplus v_3 = v_1$. Tree constraints pose a technical difficulty due to the infinite tree domain, *e.g.*, $v_1 \oplus v_2 = \bullet$ has infinitely many solutions $\{(\bullet, \circ), (\mathcal{L}, \mathcal{R}), \ldots\}$. The type of tree constraints we need to deal with can be represent as $\Sigma_1 \vdash \Sigma_2$ where Σ_i is *share equation system*:

Definition 1. *A share equation system Σ is a quadruple $(l^{\exists}, l^{=}, l^{+}, l^{-})$ in which:*

1. *$l^{\exists}$ is the list of existential variables.*
2. *$l^{=}$ is the list of equalities $\pi_1 = \pi_2$.*
3. *l^{+} is the list of equations $\pi_1 \oplus \pi_2 = \pi_3$.*
4. *l^{-} is the list of disequations $\neg(\pi_1 \oplus \pi_2 = \pi_3)$.*

The entailment $\Sigma_1 \vdash \Sigma_2$ can be informally understood as "all solutions of Σ_1 are also solutions of Σ_2". In theory, it is conventional 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. For **convenience**, we will usually illustrate equation system as $\Sigma = \{x_1, \ldots, x_n, g_1, \ldots, g_m\}$ in which x_i is existential variable and g_i is either equality, equation or disequation.

To define the semantics of Σ, let *context* ρ be a mapping from variable names to tree shares. We then override ρ over tree constants as identity, *i.e.*, $\rho(\tau) = \tau$. To handle existential variable lists, we define the notion of a *context override*:

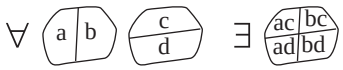
$$\rho[\rho' \Leftarrow l] \quad \overset{\text{def}}{=} \quad \lambda x.\ \rho'(v) \text{ if } x \in l \text{ else } \rho(v)$$

The semantics of forcing, written $\rho \models \Phi$, follows natural, *e.g.*, $\rho \models \pi_1 \oplus \pi_2 = \pi_3$ iff $\rho(\pi_1) \oplus \rho(\pi_2) = \rho(\pi_3)$ and $\rho \models P \wedge Q$ iff $\rho \models P$ and $\rho \models Q$. We say ρ is a solution of Σ, denoted by $\rho \models \Sigma$, if there exists a context ρ' such that $\rho[\rho' \Leftarrow l^{\exists}] \models l^{=} \wedge l^{+} \wedge l^{-}$. Consequently, we say Σ_1 entails Σ_2 if all solutions of Σ_1 are also solutions of Σ_2. In this paper, we propose *certified algorithms* to solve the satisfiability and entailment over tree shares:
Problem. Let Σ_1, Σ_2 be share equation systems. Construct a sound and complete procedure to handle the following queries:

1. **SAT**(Σ_1): Is Σ_1 satisfiable, *i.e.*, $\exists \rho.\ \rho \models \Sigma_1$?
2. **IMP**(Σ_1, Σ_2): Does Σ_1 entail Σ_2, *i.e.*, $\forall \rho.\ \rho \models \Sigma_1 \Rightarrow \rho \models \Sigma_2$?

Despite allowing negative clauses, entailment is not subsumed by satisfiability due to the quantifier alternation in the consequent. One interesting exercise is to examine the metatheoretical properties of tree shares described by Dockins et al. [15]; these are given in Figure 1. Several of these are the standard properties of separation algebras [8], but others are part of what make the tree share model special. In particular, tree shares are one of the fractional permission models 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).

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

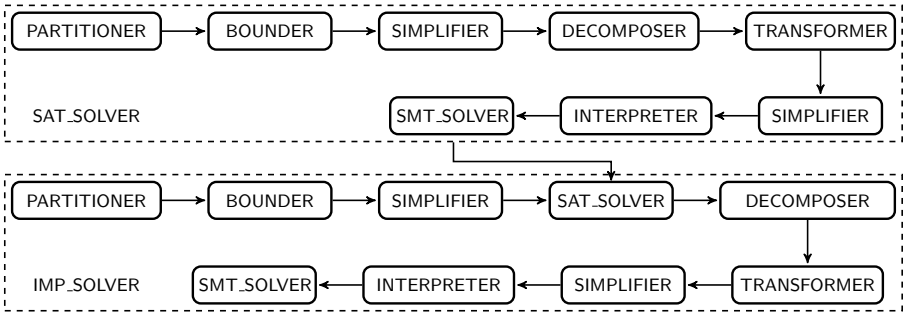


Fig. 2. SAT solver and IMP solver

Encouragingly, all of the properties except for "Unit" are expressible as entailments in our format; *e.g.* associativity is expressed as:

$$\{x \oplus a = b, y \oplus z = a\} \quad \vdash \quad \{c, x \oplus y = c, c \oplus z = b\}$$

Unit requires the order of quantifiers to swap; our format can express the weaker "Multiunit axiom" $\forall x.\ \exists u.\ x \oplus u = x$ as well as $\forall x.\ x \oplus \circ = x$.

2.2 Overview of our decision procedure

We use **SAT** and **IMP** for the problems and SAT and IMP for the decision procedures themselves. Although the entailment checker IMP is our main concern, the satisfiability checker SAT is helpful for at least two reasons. First, SAT helps to prune the search space; *e.g.*, if the antecedent Σ_1 for IMP is unsatisfiable, we can immediately conclude $\Sigma_1 \vdash \Sigma_2$. Second, the correctness of some of the transformations in IMP require that Σ_1 be satisfiable.

The architecture of our system is given in Figure 2. We have two procedures to solve problems over share formulas, one for satisfiability and the other for entailment, both written in Gallina and certified in Coq. Identically-named components in the two procedures are similar in spirit but not identical in operation; thus *e.g.* there are two different SIMPLIFIER components, one for SAT and

another for IMP. The PARTITIONER, BOUNDER, and SIMPLIFIER components substantially improve the performance of our procedures in practice but are not complete solvers: in the worst case they do nothing. Since they are included for performance we will discuss them in more detail in §4.

The DECOMPOSER and TRANSFORMER components form the heart of our procedure. While the $\oplus$ operation has many useful properties that enable sophisticated reasoning about shared ownership in program verifications (*e.g.* Figure 1), they are not strong enough for techniques like Gaussian elimination (which even in $\mathbb{Q}$ cannot handle negative clauses). In §3 we will describe DECOMPOSER in detail after developing the necessary theory. Briefly, DECOMPOSER takes a system of equations with constants of arbitrary complexity and eventually produces a much larger equivalent system in which each constant is either $\circ$ or $\bullet$ (*i.e.*, the final system has *height* zero).

TRANSFORMER is a very sophisticated component mathematically, yet also the simplest computationally: it just changes the **type** of the system. That is, it inputs a **tree** system of height zero and outputs an equivalent, essentially identical **Boolean** system. The only actual computational content is by swapping $\circ$ for $\bot$ and $\bullet$ for $\top$. The join relation on Booleans is simply disjoint disjunction:

$$\top \oplus \bot = \top \qquad \bot \oplus \top = \top \qquad \bot \oplus \bot = \bot$$

The last option, $\top \oplus \top$, is undefined.

INTERPRETER translates Boolean systems of equations into Boolean sentences by rewriting positive and negative equations using the rules

$$\begin{aligned} \pi_1 \oplus \pi_2 = \pi_3 &\rightsquigarrow (\pi_1 \wedge \neg\pi_2 \wedge \pi_3) \vee (\neg\pi_1 \wedge \pi_2 \wedge \pi_3) \vee (\neg\pi_1 \wedge \neg\pi_2 \wedge \neg\pi_3) \\ \neg(\pi_1 \oplus \pi_2 = \pi_3) &\rightsquigarrow (\neg\pi_1 \vee \pi_2 \vee \neg\pi_3) \wedge (\pi_1 \vee \neg\pi_2 \vee \neg\pi_3) \wedge (\pi_1 \vee \pi_2 \vee \pi_3) \end{aligned}$$

Next, it adds the appropriate quantifiers depending on the query type to reach a closed sentence. INTERPRETER's code and correctness proof are straightforward.

SMT_SOLVER uses simple quantifier elimination to check the validity of boolean sentences. Our SMT solver is rather naïve, and thus is the performance bottleneck of our tool, but we could not find a suitable Gallina alternative. As discussed in §4, despite its naïveté our overall performance seems acceptable in practice due to the heuristics in PARTITION, BOUNDER, and SIMPLIFIER.

3 Core algorithms for the decision procedures

We begin with some basic definitions and notions in §3.1 that are essential for the algorithms and their correctness proofs. In §3.2 and §3.3, we propose our decision procedures to solve **SAT** and **IMP** together with illustrated examples.

3.1 Definitions and notations

We adopt the following definitions and notations. We use nil to denote empty list, $[e_1, \ldots, e_n]$ to represent list's content, and $l +\!\!+ l'$ for list concatenation. We

Algorithm 1 Solver SAT for systems with disequations

```
1: function SAT(Σ)
2:     if SAT⁺(Σ⁺) = ⊥ then return ⊥
3:     else if l⁻ = nil then                    ▷ l⁻ is the disequation list in Σ
4:         return ⊤
5:     else   let l⁻ = [η₁, ..., ηₙ]
6:         return ⋀ᵢ₌₁ⁿ SSAT(Σ^{ηᵢ})
```

use the metavariable η to represent a single disequation. The symbols Σ and Π are reserved for systems and pairs of systems respectively; if the exact form of our systems is not important or is clear from the context, we may refer it as Γ. The symbol ρ and S are for contexts and solutions respectively. We use $|\tau|$ to indicate the height of τ. Also, we will override the height function $|\cdot|$ for equation systems and contexts to indicate the height of the highest tree constant. For a tree τ, we let τ_l and τ_r to be the left and right sub-trees of τ, *i.e.*, $\tau = \tau_l = \tau_r$ if $\tau \in \{\circ, \bullet\}$ and $\tau = \overset{\frown}{\tau_l\ \tau_r}$ otherwise. We define several basic systems for SAT and IMP as the building blocks of the decision procedures:

Definition 2. *Let $\Sigma, \Sigma_1, \Sigma_2$ be share equation systems and η, η_1, η_2 disequations. Let l be a list of disequations, we define Σ^l to be the new equation system in which the disequation list in Σ is replaced with l. For convenience, we write Σ^η as shortcut for $\Sigma^{[\eta]}$, and Σ^+ as shortcut for Σ^{nil}. Then:*

1. *If the disequation list in Σ is empty then Σ is called a positive system.*
2. *If there is exactly one disequation in Σ then Σ is called a singleton system.*
3. *If Σ_1 is positive and Σ_2 is singleton then (Σ_1, Σ_2) is called a Z-system.*
4. *If both Σ_1 and Σ_2 are singleton then (Σ_1, Σ_2) is called a S-system.*

In particular, Σ^+ is always a positive system, Σ^η is always a singleton system, $(\Sigma_1^+, \Sigma_2^\eta)$ is always a Z-system, and $(\Sigma_1^{\eta_1}, \Sigma_2^{\eta_2})$ is always an S-system.

3.2 Decision procedure for SAT

We propose the procedure SAT (Alg. 1) to solve **SAT** of systems with disequations. For $\mathbf{SAT}(\Sigma)$, the existential list is redundant and thus will be ignored. Our new decision procedure SAT also makes use of the old decision procedure SAT^+ from previous work [27] for systems without disequations, *e.g.*, positive systems. To help the readers gain intuition, we will abstract away all the tedious low-level implementations and only discuss about the high-level structure. The execution of SAT consists of two major steps which are described in Alg. 1. First, the system Σ is separated into a list of singleton systems; each contains a single disequation taken from the disequation list of Σ. In the second step, each singleton system is solved individually using the subroutine SSAT, then their results are conjoined to determine the result of $\mathsf{SAT}(\Sigma)$.

The solver SSAT for singleton system (Alg. 2) calls subroutine $\mathsf{DECOMPOSE}$ (Alg. 3) that helps decompose a share system into sub-systems of height zero.

Algorithm 2 Solver SSAT for singleton systems

function $\mathsf{SSAT}(\Sigma^\eta)$
Require: Σ^η is singleton and Σ^+ is satisfiable
  $[\Sigma_1, \ldots, \Sigma_n] \leftarrow \mathsf{DECOMPOSE}(\Sigma^\eta)$
  transform each Σ_i into Boolean formula Φ_i
  $\Phi \leftarrow \bigvee_{i=1}^{n} \Phi_i$
  return $\mathsf{SMT\_SOLVER}(\Phi)$

Algorithm 3 Decompose system into sub-systems of height zero

function $\mathsf{DECOMPOSE}(\Gamma)$
Require: Γ is either one system (**SAT**) or pair of systems (**IMP**)
Ensure: A list of sub-systems of height zero
  if $|\Gamma| = 0$ **then return** $[\Gamma]$
  else
    $(\Gamma_1, \Gamma_2) \leftarrow \mathsf{SINGLE\_DECOMPOSE}(\Gamma)$
    return $\mathsf{DECOMPOSE}(\Gamma_1) +\!\!+ \mathsf{DECOMPOSE}(\Gamma_2)$

function $\mathsf{SINGLE\_DECOMPOSE}(\Gamma)$
Require: Γ is either one system (**SAT**) or pair of systems (**IMP**)
Ensure: A pair of left and right sub-system
  if $|\Gamma| = 0$ **then return** (Γ, Γ)
  else
    $\Gamma_l \leftarrow$ replace each tree constant τ in Γ with its left sub-tree τ_l
    $\Gamma_r \leftarrow$ replace each tree constant τ in Γ with its right sub-tree τ_r
    return (Γ_l, Γ_r)

These subs-systems subsequently go though a 2-phase process to be transformed into Boolean formulas. In the first phase, the subroutine TRANSFORM trivially converts tree type into Boolean type using the conversions $\bullet \rightsquigarrow \top$ and $\circ \rightsquigarrow \bot$. Correspondingly, the share system is converted into the Boolean system. In the second phase, the subroutine INTERPRET helps to interpret the Boolean system into an equivalent Boolean formula by adding necessary quantifiers ($\exists$ for **SAT**, $\forall$ for **IMP**) and conjunctives among equations and disequations. Finally, Theorem 1 states the correctness of SAT whose proof is verified in Coq.

Theorem 1. *Let Σ be a share system then Σ is satisfiable iff* $\mathsf{SAT}(\Sigma) = \top$.

Example 1. Let $\Sigma = \{v_1 \oplus v_2 = \bullet, \neg(v_1 = \mathcal{L}), \neg(v_2 = \circ)\}$ then $\mathbf{SAT}(\Sigma)$ is the valid formula ($v_1 = \mathcal{R}$, $v_2 = \mathcal{L}$ is a solution):

$$\exists v_1 \exists v_2.\ v_1 \oplus v_2 = \bullet \ \wedge\ \neg(v_1 = \mathcal{L}) \ \wedge\ \neg(v_2 = \circ)$$

First, $\mathsf{SAT}^+(\Sigma^+)$ is called to check $\exists v_1 \exists v_2.\ v_1 \oplus v_2 = \bullet$ (which returns $\top$ as $v_1 = \circ$, $v_2 = \bullet$ is a solution). After that, Σ is split into two singleton systems:

$$\Sigma^1 = \{v_1 \oplus v_2 = \bullet, \neg(v_1 = \mathcal{L})\} \text{ and } \Sigma^2 = \{v_1 \oplus v_2 = \bullet, \neg(v_2 = \circ)\}$$

Algorithm 4 Solver IMP for entailment of share systems with disequations

1: **function** $\mathsf{IMP}(\Sigma_1, \Sigma_2)$
2: **if** $\mathsf{SAT}(\Sigma_1) = \bot$ **then return** $\bot$
3: **else if** $\mathsf{IMP}^+(\Sigma_1^+, \Sigma_2^+) = \bot$ **then return** $\bot$
4: **else** let l_1^-, l_2^- be disequation lists of Σ_1, Σ_2
5: **if** $l_2^- = \mathsf{nil}$ **then return** $\top$
6: **else** let $l_2^- = [\eta_2^1, \ldots, \eta_2^n]$
7: **if** $l_1^- = \mathsf{nil}$ **then return** $\bigwedge_{i=1}^n \mathsf{ZIMP}(\Sigma_1^+, \Sigma_2^{\eta_2^i})$
8: **else** let $l_1^- = [\eta_1^1, \ldots, \eta_1^m]$
9: **for** $i = 1 \ldots n$ and $j = 1 \ldots m$ **do**
10: let $Z_i \leftarrow \mathsf{ZIMP}(\Sigma_1^+, \Sigma_2^{\eta_2^i})$ and $S_i^j \leftarrow \mathsf{SIMP}(\Sigma_1^{\eta_1^j}, \Sigma_2^{\eta_2^i})$
11: **return** $\bigwedge_{i=1}^n (Z_i \vee (\bigvee_{j=1}^m S_i^j))$

When $\mathsf{NSAT}(\Sigma^1)$ is called, Σ^1 is split into Σ_1^1 and Σ_2^1 by DECOMPOSER:

$$\Sigma_1^1 = \{v_1 \oplus v_2 = \bullet, \neg(v_1 = \bullet)\} \text{ and } \Sigma_2^1 = \{v_1 \oplus v_2 = \bullet, \neg(v_1 = \circ)\}$$

The two systems Σ_1^1, Σ_2^1 are transformed into boolean formulas Φ_1^1 and Φ_2^1:

$$\begin{aligned} \Phi_1^1 &= \exists v_1 \exists v_2.\ ((v_1 \wedge \neg v_2) \ \vee \ (\neg v_1 \wedge v_2)) \ \wedge \ \neg v_1 \\ \Phi_2^1 &= \exists v_1 \exists v_2.\ ((v_1 \wedge \neg v_2) \ \vee \ (\neg v_1 \wedge v_2)) \ \wedge \ v_1 \end{aligned}$$

As both Φ_1^1 and Φ_2^1 are valid, $\mathsf{NSAT}(\Sigma^1)$ returns $\top$. Similarly, one can verify that $\mathsf{NSAT}(\Sigma^2)$ also returns $\top$ and thus $\mathsf{SAT}(\Sigma)$ returns $\top$ as the result. □

Additional details of the soundness proof for SAT can be found in [29, §B.1], which uses a technique we call "domain reduction", explained in [29, §A.1]. We finish §3.2 by pointing out a decidability result of $\oplus$:

Corollary 1. *The $\exists$-theory of $\mathcal{M} = \langle \mathbb{T}, \oplus, = \rangle$ is decidable.*

Proof. Let Ψ be a quantifier-free formula in $\mathcal{M}$, we convert Ψ into Disjunctive Normal Form $\bigvee_{i=1}^n \Psi_i$ then each Ψ_i can be represented as a constraint system Σ_i. As a result, Ψ is satisfiable iff some Σ_i is satisfiable which can be solved using Algo. 1. Thus the result follows.

3.3 Decision procedure for IMP

Our IMP procedure (Alg. 4) deploys a similar strategy as for SAT by reducing the entailment into several entailments of the basic systems (*e.g.* Z-system and S-system). In detail, IMP verifies the entailment $\Sigma_1 \vdash \Sigma_2$ by first calling two solvers $\mathsf{SAT}(\Sigma_1)$ and $\mathsf{IMP}^+(\Sigma_1^+, \Sigma_2^+)$[1] (line 2 and 3). Then the lengths of the two disequation lists (l_1^- in Σ_1 and l_2^- in Σ_2) critically determine the subsequent flow of IMP. To be precise, there are three different cases of l_1^- and l_2^- that fully cover all the possibilities:

[1] This is the entailment checker for positive constraints from previous work [27].

Algorithm 5 Solvers for entailment of Z-systems and S-systems

1: **function** $\mathsf{ZIMP}(\Sigma_1, \Sigma_2)$
Require: (Σ_1, Σ_2) is Z-system, Σ_1 is satisfiable and $\Sigma_1 \vdash \Sigma_2^+$
2: $[\Gamma_1, \ldots, \Gamma_n] \leftarrow \mathsf{DECOMPOSE}(\Sigma_1, \Sigma_2)$
3: transform each Γ_i into Boolean formula Φ_i
4: $\Phi \leftarrow \bigvee_{i=1}^{n} \Phi_i$
5: **return** $\mathsf{SMT\_SOLVER}(\Phi)$
6:
7: **function** $\mathsf{SIMP}(\Sigma_1, \Sigma_2)$
Require: (Σ_1, Σ_2) is S-system, Σ_1^+ is satisfiable, $\Sigma_1^+ \vdash \Sigma_2^+$ and $\Sigma_1^+ \nvdash \Sigma_2$
8: $[\Gamma_1, \ldots, \Gamma_n] \leftarrow \mathsf{DECOMPOSE}(\Sigma_1, \Sigma_2)$
9: transform each Γ_i into Boolean formula Φ_i
10: $\Phi \leftarrow \bigwedge_{i=1}^{n} \Phi_i$
11: **return** $\mathsf{SMT\_SOLVER}(\Phi)$

1. If $l_2^- = \mathsf{nil}$ (line 5) then the answer is equivalent to $\mathsf{IMP}^+(\Sigma_1^+, \Sigma_2^+)$, *i.e.*, $\top$.
2. Otherwise, we check whether $l_1^- = \mathsf{nil}$ (line 7) from which the answer is conjoined from several entailments of Z-systems $(\Sigma_1, \Sigma_2^{\eta_2^i})$; each is constructed from (Σ_1, Σ_2) by removing all disequations in Σ_2 except for one. Here we call the subroutine ZIMP which is a specialized for entailment of Z-systems.
3. The third case is neither l_1^- nor l_2^- is empty (line 8). Then $\Sigma_1 \vdash \Sigma_2$ is derived by taking the conjunction of several entailments of Z-systems and S-systems altogether. Here we use SIMP to solve S-system entailments.

Two specialized solvers ZIMP and SIMP are described in Alg. 5. For ZIMP, we first call the subroutine $\mathsf{DECOMPOSE}$ to split the Z-system into sub-systems of height zero. Next, each sub-system is transformed in to Boolean formula by adding necessary quantifiers and logical connectives. These Boolean formulas are then combined using disjunctions to form a single Boolean formula; and this formula is solved using standard SMT solvers to determine the result of the entailment. The procedure for SIMP has a similar structure, except that the final Boolean formula is formed using conjunctions. Also, it is worth noticing that there are certain preconditions for both solvers; and all of them are important to shape the correctness of the solvers. Last but not least, the correctness of IMP is mentioned in Theorem 2; and its proof is verified entirely in Coq.

Theorem 2. *Let Σ_1, Σ_2 be share systems then $\Sigma_1 \vdash \Sigma_2$ iff $\mathsf{IMP}(\Sigma_1, \Sigma_2) = \top$.*

Example 2. The infinite splitability of tree share (Fig.1):

$$\forall v.\ (v \neq \circ \Rightarrow \exists v_1 \exists v_2.\ v_1 \oplus v_2 = v \wedge v_1 \neq \circ \wedge v_2 \neq \circ)$$

can be represented as the entailment $\Sigma_1 \vdash \Sigma_2$ s.t.:

$$\Sigma_1 = \{\neg(v = \circ)\} \text{ and } \Sigma_2 = \{v_1, v_2, v_1 \oplus v_2 = v, \neg(v_1 = \circ), \neg(v_2 = \circ)\}$$

This entailment will go though Algo. 4 until line 8 because both disequation lists are nonempty. As there are two disequations in Σ_2, namely $\eta_1 : v_1 \neq \circ$ and

$\eta_2 : v_2 \neq \circ$, we need to verify the conjunction $P_1 \wedge P_2$ s.t.:

$$P_1 = \mathsf{ZIMP}(\Sigma_1^+, \Sigma_2^{\eta_1}) \vee \mathsf{SIMP}(\Sigma_1, \Sigma_2^{\eta_1}) \text{ and } P_2 = \mathsf{ZIMP}(\Sigma_1^+, \Sigma_2^{\eta_2}) \vee \mathsf{SIMP}(\Sigma_1, \Sigma_2^{\eta_2})$$

For P_1, $\mathsf{ZIMP}(\Sigma_1^+, \Sigma_2^{\eta_1})$ is equivalent to $\forall v.\ (\top \Rightarrow \exists v_1, v_2.\ v_1 \oplus v_2 = v \wedge v_1 \neq \circ)$ which is false by choosing $v = \circ$ so that both v_1 and v_2 must also be $\circ$. Likewise, $\mathsf{SIMP}(\Sigma_1, \Sigma_2^{\eta_1})$ is equivalent to $\forall v.\ (v \neq \circ \Rightarrow \exists v_1, v_2.\ v_1 \oplus v_2 = v \wedge v_1 \neq \circ)$ which is transformed into the boolean formula:

$$\Phi_1 \ = \ \forall v.\ (v \Rightarrow \exists v_1, v_2.\ ((\neg v_1 \wedge \neg v_2 \wedge \neg v) \vee (v_1 \wedge \neg v_2 \wedge v) \vee (\neg v_1 \wedge v_2 \wedge v)) \wedge v_1)$$

As Φ_1 is valid, P_1 is true. Same result holds for P_2 and thus $\Sigma_1 \vdash \Sigma_2$. □

Additional details of the soundness proof for IMP can be found in [29, §B.2], again using domain reduction [29, §A.1].

4 Performance, evaluation, and implementation

Having described the heart of our decision procedures, what remains is to describe the practical aspects of their development and evaluation. In §4.1 we describe various techniques that enable good performance in practice. In §4.2 we describe how we benchmarked our tool running inside Coq, running as a standalone compiled program, and after incorporating it into the HIP/SLEEK verification toolset. In [29, §C] we document the files in the development itself; we have approximately 38.6k lines of code in 31 files.

4.1 Performance-enhancing components

The architecture of our tool was given in §2.2 (Figure 2). The key DECOMPOSER, TRANSFORMER and INTERPRETER components were discussed in §2.2, §3.2, and §3.3. Here we give details on the PARTITIONER, BOUNDER, and SIMPLIFIER modules. Their principal goal is to shrink the search space and uncover contradictions, although they each do so in a very different way. Although in practice they can substantially improve performance, none of these components is a complete solver. The key ideas in these components were developed previously [21, 27], although not all together. We have made a number of incremental enhancements, but our major contribution for these is components is the development of high-performing general-purpose certified implementations.

PARTITIONER. The goal of this module is to separate a constraint system into *independent subsystems*. Two systems are independent of each other if they do not share any common variable (with existential variables bound locally).

The partition function is implemented *generically*: in other words it does not assume very much about the underlying domain. To build the module, we must specify types of *variables* V, *equations* E, and contexts C. We also provide a function $\sigma : E \Rightarrow L(V)$ that extracts a list of variables from an equation, an

overriding function written $\rho'[\rho \Leftarrow l]$, and an evaluation relation written $c \models e$. The soundness proof requires two properties that relate these inputs as follows:

$$\frac{\rho \models e \qquad \sigma(e) \cap l = \varnothing}{\rho[\rho' \Leftarrow l] \models e}\ \textit{disjointness} \qquad\qquad \frac{\rho \models e \qquad \sigma(e) \subset l}{\rho'[\rho \Leftarrow l] \models e}\ \textit{inclusion}$$

Disjointness and inclusion jointly specify that satisfaction of an equation only depends on the variables it contains: overriding variables not in the equation does not matter; and from any context, if we override all of the variables that are in an equation then we can ignore the original context.

It is simple to use PARTITIONER for **SAT**, but to handle **IMP** is harder. We can "tag" equations and variables as coming from the antecedent or consequent before partitioning and then use these tags to separate the resulting partitioned systems into antecedents and consequents afterwards.

The implementation of PARTITIONER is nontrivial in purely functional languages like Coq. One reason is that we need a purely functional union-find data structure, which we obtain via the impure-to-pure transformation of Pippenger [34] applied to the canonical imperative algorithm [11]. In other words, we substitute red-black trees for memory (mapping "addresses" to "cell contents") and pay a logarithmic access penalty, yielding an $O\big(n \cdot \mathsf{log}(n) \cdot \alpha(n)\big)$ algorithm.

The termination of "find" turns out to be subtle. Parent pointers are represented as cells that "point to" other cells; however, those parent cells can be anywhere in the red-black tree (*e.g.* item 5 can be the parent of item 10, or the other way around.) Accordingly an important invariant of the structure is that "nonlocal links" form acyclic chains, which is the key termination argument.

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

BOUNDER. The bounder uses order theory to prune the space. Each variable v is given an initial bound $\circ \subseteq v \subseteq \bullet$. The bounder then tries to narrow these bounds by forward and backward propagation. For example, if $\tau_1 \subseteq v_1 \subseteq \bullet$, $\tau_2 \subseteq v_2 \subseteq \bullet$, and $\circ \subseteq v_3 \subseteq \bullet$, then if $v_1 \oplus v_2 = v_3$ is an clause we can conclude that v_3's lower bound can be increased from $\circ$ to $\tau_1 \sqcup \tau_2$ (where $\sqcup$ computes the union in an underlying lattice on trees). In some cases, the bounds for a variable can be narrowed all the way to a point, in which case we can substitute the variable away. In other cases we can find a contradiction (when the upper bound goes below the lower bound), allowing us to terminate the procedure.

The bounder is an updated version of the incomplete solver developed by Hobor *et al.* [21]. Although our main contribution here is the certified implementation, we managed to tighten the bounds in certain cases.

SIMPLIFIER. The simplifier is a combination of a substitution engine and several effective heuristics for reducing the overall difficulty via calculation. For example, from $v \oplus \tau_1 = \tau_2$, where τ_i are constants, we can compute an exact value for v using an inverse of $\oplus$: $v = \tau_2 \ominus \tau_1$. SIMPLIFIER also hunts for contradictions: for

example, from $v \oplus v = \bullet$ we can reach a contradiction due to the "disjointness" axiom from Figure 1. The core idea of simplifier was contained in the work of Le *et al.* [27], so our main contribution here is our certified implementation.

4.2 Experimental evaluation

Our procedures are implemented and certified in Coq. Users who wish to use our code outside of Coq can use Coq's extraction feature to generate code in OCaml and Haskell, although at present a small bug in Coq 8.4pl5's extraction mechanism requires a small human edit to the generated code.

We benchmarked our code in three ways using an Intel i7 with 8GB RAM. First, we used a suite of 102 standalone test cases developed by Le *et al.* (53 **SAT** and 49 **IMP**) [27] and the 9 metatheoretic properties described in §2. These tests cover a variety tricky cases such as large number of variables, deep tree constants, etc. Even running as interpreted Gallina code within Coq, the time is extremely encouraging at **17 seconds** to check all 111 tests. After we port to Coq 8.5 we can use the native_compute tactic to increase performance.

Second, we compiled the extracted OCaml code with ocamlopt. The total running time to test all 111 previous tests is **0.02 seconds**, despite our naïve SMT solver; our previous tool took 1.4 seconds. Since our SMT solver is a separate module, it can be replaced with a more robust external solver such as Z3 [12] if performance is bottleneck in that spot in the future.

Finally, we incorporated our solver into the HIP/SLEEK verification toolset, which was previously using the uncertified solver by Le *et al.*. We did so by writing a short (approximately 150 line) "shim" that translated the format used by the previous tool into the format expected by the new tool.

We then benchmarked our tool against a suite of 23 benchmark programs as shown in Figure 3. 15 of those programs were developed by Gherghina [17] and utilize a concurrent separation logic for pthreads-style barriers that exercise share provers extensively. Another 7 tests were developed for the HipCAP project [9], which extended HIP/SLEEK to reason in a Concurrent Abstract Predicate [14] style. Finally, we wrote a simple fork/join program for our initial testing.

The results are rather interesting! The left column gives the input file name to HIP/SLEEK and the second the number of lines in that file. The third column is the total number of calls into the solver (both **SAT** and **IMP**). **The fourth column is the number of times the previous solver by Le *et al.* answered the query incorrectly.** The fifth column gives the time (in seconds) spent by Le *et al.*'s uncertified solver and the sixth column gives the time spent by our new certified solver. HIP/SLEEK was benchmarked on a more powerful machine with 16 cores and 64GB RAM.

The uncertified solver got approximately 5.2% of the queries wrong! In our subsequent investigation, we discovered a number of bugs in the original solver: code rot (due to a change in the correct mechanism to call the SMT backend), improper error handling and signaling, general coding errors, and the incorrect treatment of nonzero variables. We also discovered bugs in HIP/SLEEK itself, which did not always use the result of the solver in the correct way; this is

File	LOC	# calls	# wrong	Le *et al.* [27]	Our tool
MISD_ex1_th1.ss	36	294	48	2.21	2.37
MISD_ex1_th2.ss	36	495	67	4.36	4.48
MISD_ex1_th3.ss	36	726	94	6.95	6.58
MISD_ex1_th4.ss	36	1,003	123	9.09	8.36
MISD_ex1_th5.ss	36	1,320	134	15.74	12.38
MISD_ex2_th1.ss	47	837	107	16.77	18.97
MISD_ex2_th2.ss	52	1,044	157	29.34	26.02
MISD_ex2_th3.ss	87	1,841	260	69.09	64.21
MISD_ex2_th4.ss	105	3,023	374	194.17	194.64
PIPE_ex1_th2.ss	35	283	7	2.49	2.78
PIPE_ex1_th3.ss	44	467	12	4.92	4.65
PIPE_ex1_th4.ss	56	678	15	7.00	7.53
PIPE_ex1_th5.ss	66	931	18	9.67	9.37
SIMD_ex1_v2_th1.ss	74	1,167	281	18.46	17.64
SIMD_ex1_v2_th2.ss	95	2,029	392	63.83	53.50
cdl-ex1a-fm.ss	49	7	0	0.10	0.08
cdl-ex2-fm.ss	50	9	0	0.12	0.09
cdl-ex3-fm.ss	51	10	0	0.11	0.12
cdl-ex4-race.ss	50	5	0	0.09	0.09
cdl-ex4a-race.ss	50	9	0	0.10	0.08
cdl-ex5-deadlock.ss	42	5	0	0.10	0.10
cdl-ex5a-deadlock.ss	42	9	0	0.08	0.08
ex-fork-join.ss	25	47	22	0.19	0.16
total		**10,252**	**534**	**455.01**	**434.30**

Fig. 3. Evaluation of our procedures using HIP/SLEEK

why the regression tests were passing even though the solver was reporting the incorrect answer. Our discovery of bugs on this scale, despite the large benchmarks developed by Le *et al.* [27] and Gherghina [17], illustrates the value of developing certified decision procedures.

Our timing results are reasonable: despite our naïve SMT solver backend and the difficulties in writing the algorithms in a purely functional style, our tool is approximately 4.6% faster than Le *et al.*'s uncertified solver.

5 Related work, future work, and conclusion

Boyland first proposed fractional shares over $\mathbb{Q}$ [7]. Subsequently, Bornat *et al.* [5] improved the rational model by adding natural counting permissions to reason about critical sections. Other notable refinements of the rationals are achieved by Boyland *et al.* [6], Huisman *et al.* [23] and Müller *et al.* [31] that work well on programs with fork, join and lock. Parkinson showed that $\mathbb{Q}$'s lack of disjointness caused trouble and proposed modelling shares as subsets of $\mathbb{N}$ [33]. Dockins *et al.* proposed the tree share model used in the present paper to fix issues with Parkinson's model [15]. Hobor *et al.* were the first to use tree shares in a program logic [20], followed by Hobor and Gherghina [18] and Villard [38]. Hobor and Gherghina [21], Villiard [38], and Appel *et al.* [3] subsequently integrated shares into program verification tools with various incomplete solvers. Le *et al.* [27]

developed sound and complete procedures to handle tree share constraints but their correctness proof only justifies the case when there is no disequation.

Future work. We have plans to examine the theory further to support general logical formulae (including arbitrary quantifier use) and perhaps monadic second-order logic. Dockins *et al.* also define a kind of multiplicative operation $\bowtie$ between shares whose computability and complexity was first analyzed by Le *et al.* [28]. Interestingly, this operator can be used to scale permissions over arbitrary predicates and thus our decision procedures need to be generalized to handle constraints that contain both $\oplus$ and $\bowtie$.

Conclusion. We have used tree shares to model permissions for integration into program logics. We proposed two decision procedures for tree shares and proved their correctness in Coq. The two algorithms perform well in practice and have been integrated into a sizable verification toolset.

References

1. Andrew W. Appel. Efficient verified red-black trees, 2011.
2. Andrew W. Appel, Robert Dockins, and Aquinas Hobor. Mechanized semantic library, 2009.
3. Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.
4. Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! - A framework for higher-order separation logic in Coq. In *ITP*, pages 315–331, 2012.
5. Richard Bornat, Cristiano Calcagno, Peter O'H, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.
6. Boyland, John Tang, Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Constraint semantics for abstract read permissions. In *FTfJP*, pages 2:1–2:6, 2014.
7. John Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72, 2003.
8. Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS*, pages 366–378, 2007.
9. Wei Ngan Chin, Ton Chanh Le, and Shengchao Qin. Automated verification of countdownlatch, 2017.
10. Adam Chlipala. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In *ICFP*, pages 391–402, 2013.
11. Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.
12. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
13. Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. Caper: Automatic verification for fine-grained concurrency. In *ESOP*, pages 420–447, 2017.
14. Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.

15. Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, pages 161–177, 2009.
16. Jan Fiedor, Zdeněk Letko, João Lourenço, and Tomáš Vojnar. Dynamic validation of contracts in concurrent code. In *EUROCAST*, pages 555–564, 2015.
17. Cristian A. Gherghina. *Efficiently Verifying Programs with Rich Control Flows*. PhD thesis, National University of Singapore, 2012.
18. Hobor and Cristian Gherghina. Barriers in concurrent separation logic. In *ESOP*, pages 276–296, 2011.
19. Aquinas Hobor. *Oracle Semantics*. PhD thesis, Princeton University, Department of Computer Science, Princeton, NJ, October 2008.
20. Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, pages 353–367, 2008.
21. Aquinas Hobor and Cristian Gherghina. Barriers in concurrent separation logic: Now with tool support! *Logical Methods in Computer Science*, 8(2), 2012.
22. Jochen Hoenicke, Rupak Majumdar, and Andreas Podelski. Thread modularity at many levels: A pearl in compositional verification. In *POPL*, pages 473–485, 2017.
23. Marieke Huisman and Wojciech Mostowski. A symbolic approach to permission accounting for concurrent reasoning. In *ISPDC*, pages 165–174, 2015.
24. Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650, 2015.
25. Bohuslav Křena, Zdeněk Letko, Tomáš Vojnar, and Shmuel Ur. A platform for search-based testing of concurrent software. In *PADTAD*, pages 48–58, 2010.
26. Duy-Khanh Le, Wei-Ngan Chin, and Yong Meng Teo. Threads as resource for concurrency verification. In *PEPM*, pages 73–84, 2015.
27. Xuan-Bach Le, Cristian Gherghina, and Aquinas Hobor. Decision procedures over sophisticated fractional permissions. In *APLAS*, 2012.
28. Xuan-Bach Le, Aquinas Hobor, and Anthony W. Lin. Decidability and complexity of tree shares formulas. In *FSTTCS*, 2016.
29. Xuan-Bach Le, Thanh-Toan Nguyen, Wei-Ngan Chin, and Aquinas Hobor. A certified decision procedure for tree shares (extended), 2017. `http://www.comp.nus.edu.sg/~lxbach/certtool/`.
30. Wenrui Meng, Fei He, Bow-Yaw Wang, and Qiang Liu. Thread-modular model checking with iterative refinement. In *NFM*, pages 237–251, 2012.
31. Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, pages 41–62, 2016.
32. 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.
33. Matthew Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.
34. Nicholas Pippenger. Pure versus impure LISP. In *POPL*, pages 104–109, 1996.
35. Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87, 2015.
36. Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, pages 149–168, 2014.
37. Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390, 2013.
38. Jules Villard. *Heaps and Hops*. Ph.D. thesis, Laboratoire Spécification et Vérification, École Normale Supérieure de Cachan, France, February 2011.