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 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:

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

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 consequent entailments to separate checkers such as SLEEK. In turn, entailment checkers often do not prove the entire entailment: the typical pattern is to do some processing and then send specialized decision problems to a stable of backend provers, *e.g.* Omega for Presberger arithmetic.

Our goal is to follow the same pattern for fractional shares to allow our tool to be more easily integrated into other toolchains. The program verifier itself—the part of the toolchain that applies the Hoare rules—needs know almost nothing about fractional shares (except for parsing share constants), 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 extract share information from formulae into a specialized domain: 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 prover only knows about certain basic operations such as equality testing, combining and splitting shares, and so forth. To actually 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 show how simple separation logic formulae get distilled into systems of equations. Translating more complex formulae depends on the exact nature of said formulae but usually follows the pattern we give here in a straightforward way; *e.g.*, we detail how we modified SLEEK to extract equation systems from formulae containing user-defined inductive predicates in [9, §8.4]. The end result is just larger systems of equations.

Consider the following entailment, in which we write χ_i for share constants and s_i for share variables and $\ell \overset{\pi}{\mapsto} v$ is the fractional points-to predicate:

$$x \overset{s_1}{\mapsto} v \wedge s_1 = \chi_1 \wedge s_2 = \chi_2 \ \vdash \ x \overset{s_2}{\mapsto} v$$

We can prove this entailment by matching each cell in the consequent with the corresponding cell in the antecedent. We must then prove that the cells contain the same value, *i.e.* $v = v$, which is easy to discharge. We must then consider the shares, for example by requiring that the left- and right-hand shares match exactly, leading to the following subentailment:

$$s_1 = \chi_1 \wedge s_2 = \chi_2 \ \vdash \ s_1 = s_2$$

This entailment is true if and only if $\chi_1 = \chi_2$, which is easy to check with decidable equality. Notice that our subentailment is written as a formula over shares, but is really just describing a (simple) system of equations over same.

Of course, things are usually not quite that simple. Consider a slightly different entailment that has an existential quantification over the share variable:

$$x \overset{s_1}{\mapsto} v \wedge s_1 = \chi_1 \ \vdash \ \exists s_2 \cdot x \overset{s_2}{\mapsto} v$$

This generates the obligation $s_1 = \chi_1 \vdash \exists s_2 \cdot s_1 = s_2$, which is also trivial.

The previous entailment can occur 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 and Heap-Hop) only allow *positive* (non-empty) fractional shares over a points-to predicate; thus, the above existential must be restricted so that it cannot choose the empty share.

In both previous examples all the resources in the antecedent were consumed to prove the consequent. However, 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. We next consider the case in which the antecedent contains a bigger share than required to prove the consequent (χ_1 is larger than χ_2, that is exists a non-empty share χ_3 such that $\chi_2 \oplus \chi_3 = \chi_1$):

$$x \overset{s_1}{\mapsto} v \wedge s_1 = \chi_1 \ \vdash \ \exists s_2 \cdot x \overset{s_2}{\mapsto} v \wedge s_2 = \chi_2$$

To prove this entailment it is sufficient to consume only part of the share χ_1 in the antecedent: the subshare χ_2; the residue will then be, χ_3. To handle this, a separation logic entailment checker should submit two share entailments:

– First, try to consume the entire antecedent of the node:
  $s_1 = \chi_1 \ \vdash \ \exists s_2 \cdot s_1 = s_2 \wedge s_2 = \chi_2$ (*fails*)
– Second, split the antecedent such that one part is exactly what is needed to prove the consequent:
  $s_1 = \chi_1 \ \vdash \ \exists s'_1, s_2 \cdot s_1 = s_2 \oplus s'_1 \wedge s_2 = \chi_2$ (*succeeds*)
  If the second check succeeds then the residue is $x \overset{s'_1}{\mapsto} v \wedge s'_1 = \chi$.

We have now given three examples of extracting share equations from separation logic formulae. The translations for simple separation logics is not difficult. The result is a formula over **non empty** shares with the following syntax:

$$\phi \ ::= \ \exists v.\phi \mid \phi_1 \wedge \phi_2 \mid v_1 \oplus v_2 = v_3 \mid v_1 = v_2 \mid v = \chi$$

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.

A separation logic entailment checker can ask our share prover two questions:

1. (SAT) Is a given formula ϕ satisfiable?
2. (IMPL) Given two formulae ϕ_1 and ϕ_2, does ϕ_1 entail ϕ_2?

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:

$$\tau ::= \circ \quad | \quad \bullet \quad | \quad \widehat{\tau\ \tau}$$

Here $\circ$ denotes an "empty" leaf while $\bullet$ a "full" leaf. The tree $\circ$ is thus the empty share, and $\bullet$ the full share. There are two "half" shares: $\widehat{\circ\ \bullet}$ and $\widehat{\bullet\ \circ}$, and four "quarter" shares, beginning with $\widehat{\widehat{\bullet\ \circ}\ \circ}$. 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) to really describe fractional trees instead of DAGs.

Notice also that we presented the first quarter share as $\widehat{\widehat{\bullet\ \circ}\ \circ}$ instead of $\widehat{\widehat{\bullet\ \circ}\ \widehat{\circ\ \circ}}$. 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 $\cong$:

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

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.

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:

$$\widehat{\widehat{\bullet\ \circ}\ \circ} \oplus \widehat{\widehat{\circ\ \bullet}\ \widehat{\bullet\ \circ}} \quad \cong \quad \widehat{\widehat{\bullet\ \circ}\ \widehat{\circ\ \circ}} \oplus \widehat{\widehat{\circ\ \bullet}\ \widehat{\bullet\ \circ}} \quad = \quad \widehat{\widehat{\bullet\ \bullet}\ \widehat{\bullet\ \circ}} \quad \cong \quad \widehat{\bullet\ \widehat{\bullet\ \circ}}$$

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).

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 although we know of no fundamental difficulty, our decision procedures and associated completeness results do not support $\bowtie$ at this time.

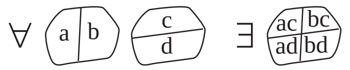
Functional:	$x \oplus y = z_1 \quad \Rightarrow \quad x \oplus y = z_2 \quad \Rightarrow \quad z_1 = z_2$
Commutative:	$x \oplus y \quad = \quad y \oplus x$
Associative:	$x \oplus (y \oplus z) \quad = \quad (x \oplus y) \oplus z$
Cancellative:	$x_1 \oplus y = z \quad \Rightarrow \quad x_2 \oplus y = z \quad \Rightarrow \quad x_1 = x_2$
Unit:	$\exists u.\ \forall x.\ x \oplus u = x$
Disjointness:	$x \oplus x = y \quad \Rightarrow \quad x = y$
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:	$x \neq \circ \quad \Rightarrow \quad \exists x_1, x_2.\ x_1 \neq \circ \ \wedge \ x_2 \neq \circ \ \wedge \ x_1 \oplus x_2 = x$

Fig. 1. Properties of tree shares

4 Formal Description

Here we introduce a decision procedure for discharging tree share proof obligations generated by program verifiers. We will view the share formulas of §2 as equation systems Σ, *i.e.* as a pair of sets: a set of equations of the form $a \oplus b = c$ or $v = w$ and a set of existentially quantified variables.

We want to answer the two queries SAT and IMPL defined in §2. The key reason that the problems are nontrivial is that the space is dense[1]. That is, there exist trees of arbitrary depth, seeming to rule out a brute force search. If we do not find a SAT solution to Σ at depth 5, how do we know that one is not lurking at depth 10,000? If we check IMPL($\Sigma \vdash \Sigma'$) when the variables are restricted to constants of depth 5, how do we know that the entailment will continue to hold when the variables are able to range over constants of arbitrary depth?

Our key theoretical insight is that despite the infinite domain, both SAT and IMPL are decidable by searching in the subdomain of trees with bounded height. Define the *system depth* $|\Sigma|$ as the depth of the deepest tree constant in Σ or 0 if Σ contains only variables[2]. A *solution* S of Σ is a (finite) mapping from the variables of Σ into tree shares; let $|S|$ be the largest constant in its codomain. In §5 we prove that for both SAT and IMPL queries, if the depth of the system(s) of equations is n, it is sufficient to restrict the search to solutions of depth n.

Of course, we do not want to blindly search through an exponentially large space if we can avoid it! The core of our decision procedure is a transformation DECOMPOSE that takes an equation system Σ of depth n and produces two systems Σ_l and Σ_r with depths at most $n-1$. We will show that it is sufficient to prove queries over Σ_l and Σ_r to prove queries over Σ. By repeatedly applying DECOMPOSE , Σ can be reduced to depth 0 (*i.e.*, the embedded constants are only $\circ$ and $\bullet$), at which point SAT and IMPL queries over trees can be naturally translated into similar queries over propositional formulae: $a \oplus b = c$ corresponds

[1] 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.

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

to $(\neg a \wedge \neg b \wedge \neg c) \vee (\neg a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c)$ while $a = b$ constraints translate to $(\neg a \wedge \neg b) \vee (a \wedge b)$. At this point we invoke an off-the-shelf SAT solver.

Decomposition. DECOMPOSE works by decomposing each constraint in $\sigma \in \Sigma$ into a left constraint σ_l and right constraint σ_r, and similarly dividing the existential variables. To decompose a constraint, DECOMPOSE is applied to each operand. A tree constant is decomposed into its left and right subtrees; a variable is decomposed by deterministically generating two fresh variables for its left and right subtree. The result is a pair of independent subsystems Σ_l and Σ_r.

$$\mathsf{DECOMPOSE}(v) = (v_l, v_r) \qquad \textit{vars}$$

$$\begin{array}{l} \mathsf{DECOMPOSE}(\circ) = (\circ, \circ) \quad \mathsf{DECOMPOSE}(\bullet) = (\bullet, \bullet) \\ \mathsf{DECOMPOSE}(\widehat{t_1\ t_2}) = (t_1, t_2) \end{array} \qquad \textit{consts}$$

$$\left.\begin{array}{l} \mathsf{DECOMPOSE}(a) = (a_l, a_r) \\ \mathsf{DECOMPOSE}(b) = (b_l, b_r) \\ \mathsf{DECOMPOSE}(c) = (c_l, c_r) \end{array}\right\} \begin{array}{r} \mathsf{DECOMPOSE}(a \oplus b = c) = \begin{pmatrix} a_l \oplus b_l = c_l, \\ a_r \oplus b_r = c_r \end{pmatrix} \\ \mathsf{DECOMPOSE}(a = b) = \begin{pmatrix} a_l = b_l, \\ a_r = b_r \end{pmatrix} \end{array} \qquad \textit{eqs}$$

DECOMPOSE can be also be applied to a solution S of Σ, and furthermore is injective because left and right subtrees of a binary tree are independent.

We can prove the soundness and completeness of DECOMPOSE as follows:

Lemma 1. *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 is a solution of Σ iff S_l is a solution of Σ_l and S_r is a solution of Σ_r.*

Then, given $\mathsf{DECOMPOSE}(\Sigma) = (\Sigma^1, \Sigma^2)$ and $\mathsf{DECOMPOSE}(\Sigma_i) = (\Sigma_i^1, \Sigma_i^2)$:

Theorem 1. *SAT(Σ) holds iff both SAT(Σ^1) and SAT(Σ^2) hold.*

Theorem 2. *IMPL($\Sigma_1 \vdash \Sigma_2$) holds iff IMPL($\Sigma_1^1 \vdash \Sigma_2^1$) and IMPL($\Sigma_1^2 \vdash \Sigma_2^2$) hold.*

Algorithm outline. Figures 2 and 3 outline the algorithms that discharge SAT and IMPL queries. The REDUCE and REDUCEI functions call both DECOMPOSE and the transformation to a boolean formula (TO_SAT) discussed earlier and return one/two first order boolean formulae encoding the input equation systems.

The IMPL algorithm prescribes one additional step: adding the existential quantifications on the consequent in order for them to be carried over to the SAT solver. As an optimization, we do not add them for SAT checks as the SAT question implicitly treats each free variable as an existential one.

SIMPLIFY attempts to aggressively apply available substitutions and reductions to reduce the size of the formulae fed to the SAT solvers. In §7 we outline some simplifications during the decomposition process.

```
REDUCE($\Sigma'$)
  $\Sigma$ = SIMPLIFY($\Sigma'$)
  If ($|\Sigma|$ = 0)
    Return TO_SAT($\Sigma$)
  Else
    ($\Sigma_l, \Sigma_r$) = DECOMPOSE($\Sigma$)
    $\Phi$ = REDUCE($\Sigma_l$)$\wedge$REDUCE($\Sigma_r$)
    Return $\Phi$

SAT($\Sigma$)
  $\Phi$ = REDUCE($\Sigma$)
  Return SAT_SOLVER($\Phi_r$)
```

Fig. 2. SAT

```
REDUCEI($\Sigma'_1, \Sigma'_2$)
  $\Sigma_1$ = SIMPLIFY($\Sigma'_1$)
  $\Sigma_2$ = SIMPLIFY($\Sigma'_2$)
  If ($|\Sigma_1|$ = 0 $\wedge$ $|\Sigma_2|$ = 0)
    Return (TO_SAT($\Sigma_1$), TO_SAT($\Sigma_2$))
  Else
    ($\Sigma^l_1, \Sigma^r_1$) = DECOMPOSE($\Sigma_1$)
    ($\Sigma^l_2, \Sigma^r_2$) = DECOMPOSE($\Sigma_2$)
    ($\Phi^l_a, \Phi^l_c$) = REDUCEI($\Sigma^l_1, \Sigma^l_2$)
    ($\Phi^r_a, \Phi^r_c$) = REDUCEI($\Sigma^r_1, \Sigma^r_2$)
    Return($\Phi^l_a \wedge \Phi^r_a$, $\Phi^l_c \wedge \Phi^r_c$)

IMPL($\Sigma_1, \Sigma_2$)
  ($\Phi_a, \Phi_c$) = REDUCEI($\Sigma_1, \Sigma_2$)
  $\Phi''_a$ = ADD_EXISTS($\Phi'_a$, $\Sigma_1$)
  $\Phi''_c$ = ADD_EXISTS($\Phi'_c$, $\Sigma_2$)
  Return $\neg$ SAT_SOLVER($\Phi''_a \wedge \neg\Phi''_c$)
```

Fig. 3. IMPL

Complexity. One interesting question is what complexity class SAT and IMPL belong to. Unfortunately, we do not have any firm convictions other than that it is unlikely to be pretty. We have proved that tree-SAT when restricted to systems of depth 0 is already NP-COMPLETE. We hypothesize that tree-SAT on systems of arbitrary depth is still "only" NP-COMPLETE because we believe that *the branching of the system scales polynomially with the description of the system.* Going a bit further onto a limb, we further hypothesize that tree-IMPL is no worse than NP-COMPLETE$^{\text{NP-COMPLETE}}$. Happily, as we show in §7.1, performance seems to be adequate in practice.

5 Correctness Result

Here we present the correctness proofs for SAT and IMPL, modulo the soundness and completeness of the off-the-shelf SAT solver.

5.1 SAT

The SAT algorithm from Figure 2 is in essence doing a shape-guided search through a finite domain and the proof that it in fact does so correctly is not novel. What is not obvious is why the finite search is sufficient. Our goal is thus to prove Theorem 3: SAT(Σ) holds iff Σ has a solution S with $|S|{=}|\Sigma|$.

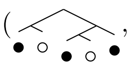
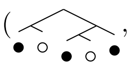
Let $CT_{T_{\mathbb{B}}}$ be the domain of canonical trees and $\mathsf{CT}(\mathsf{t})$ be the reduction of t into its canonical form. Conversely, $\mathsf{Expand}(t, n)$ returns the expansion of t into a (non-canonical $\cong$-equivalent) fully-unfolded tree of depth n. That is, $\mathsf{Expand}(t, n)$ is the full tree of depth n in t's equivalence class.

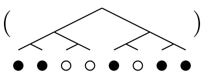
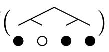
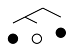
Definition 1. *Let* $\mathsf{take_L}(\mathsf{t})$ *be the operation of replacing in t, all subtrees* $\widehat{t_1\ t_2}$ *of height 1* ($t_1, t_2 \in \{\circ, \bullet\}$) *with* t_1. *Similarly,* $\mathsf{take_R}(\mathsf{t})$ *replaces* $\widehat{t_1\ t_2}$ *with* t_2.

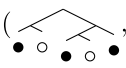
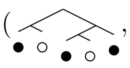
Definition 2. $\mathsf{Lround},\ \mathsf{Rround} : CT_{T_{\mathbb{B}}} \times \mathbb{N} \to CT_{T_{\mathbb{B}}}$. *If* $|t| \leq n$ *then :*

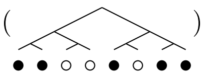
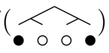
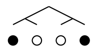
$\mathsf{Lround}(t,n){=}\mathsf{CT}(\mathsf{take_L}(\mathsf{Expand}(t,n))) \qquad \mathsf{Rround}(t,n){=}\mathsf{CT}(\mathsf{take_R}(\mathsf{Expand}(t,n)))$

Given a tree t and a depth n such that $|t| \leq n$, Lround returns the left rounded tree, a tree t' such that $|t'| < n$. Conversely, Rround gives the right rounded tree.

e.g. $\mathsf{Lround}($  $, 3) \;=\; \mathsf{CT}(\mathsf{take_L}(\mathsf{Expand}($  $, 3))) \;=$

$\mathsf{CT}(\mathsf{take_L}($  $)) \;=\; \mathsf{CT}($  $) \;=$ 

and $\mathsf{Rround}($  $, 3) \;=\; \mathsf{CT}(\mathsf{take_R}(\mathsf{Expand}($  $, 3))) \;=$

$\mathsf{CT}(\mathsf{take_R}($  $)) \;=\; \mathsf{CT}($  $) \;=$ 

Following $\mathsf{DECOMPOSE}$, we extend Lround and Rround to work over systems Σ and solutions S. We just cover Lround in detail because Rround is similar:

$$\mathsf{Lround}(v,n) \;= \mathsf{v} \qquad\qquad \mathsf{Lround}(S,n) = \{x_i = \mathsf{Lround}(v_i,n) \mid x_i = v_i \in S\}$$

$$\begin{aligned}\mathsf{Lround}(\Sigma,n) = {} & \{\mathsf{Lround}(a,n) \;=\; \mathsf{Lround}(b,n) \mid a \;=\; b \in \Sigma\} \\ & \cup \{\mathsf{Lround}(a,n) \oplus \mathsf{Lround}(b,n) = \mathsf{Lround}(c,n) \mid a \;\oplus\; b = c \in \Sigma\}\end{aligned}$$

Lemma 2 **(**Lround **properties).**

1. *If* $n > |t|$ *then* $\mathsf{Lround}(t,n) = t$
2. *If* $n = |t|$ *and* $t' = \mathsf{Lround}(t,n)$ *then* $|t'| \leq n - 1$
3. *If* $a \oplus b{=}c$ *and* $n{=}|a \oplus b{=}c|$ *then* $\mathsf{Lround}(a,n) \oplus \mathsf{Lround}(b,n){=}\mathsf{Lround}(c,n)$
4. *If* $n < |t|$ *then* $\mathsf{Lround}(t,n)$ *is undefined*

Proved in Coq. Lemma 2 states that $\mathsf{Lround}(t,n)$ does not affect t if $n > |t|$ and will transform t to t' decreasing the depth by at least 1 if $n = |t|$. Lround preserves the join relation, so given a system Σ and solution S of depth n such that $n > |\Sigma|$, we can construct a solution S' of smaller depth by applying Lround to S. Also, $S' = \mathsf{Lround}(S,n)$ is a solution of $\mathsf{Lround}(\Sigma,n)$. Thus, when $n > |\Sigma|$, then $\mathsf{Lround}(\Sigma,n) = \Sigma$ which implies that $\mathsf{Lround}(S,n)$ is a solution of Σ.

Lemma 3. *Let S be a solution of Σ and $n = |S| > |\Sigma|$. Then $S' = \mathsf{Lround}(S,n)$ is also a solution of Σ and $|S'| \leq n - 1$.*

Direct from Lemma 2. Also, the depth of solution S of Σ has a lower bound:

Lemma 4. *Let S be a solution of Σ. Then $|S| \geq |\Sigma|$.*

Proof. WLOG assume that Σ only contains equations $a \oplus b = c$ with at most one constant[3]. If $|\Sigma| = 0$, we are done. Otherwise, by definition of $|\Sigma| = n$, there must be a constant with depth n. Then the equation that contains such constant must also contain at least 1 variable with depth at least n. Hence, $|S| \geq n$.

[3] A join fact with two or three constants can be simplified.

Theorem 3. *$SAT(\Sigma)$ holds iff Σ has a solution S with $|S|{=}|\Sigma|$.*

Proof.
$\Leftarrow$: Immediate.
$\Rightarrow$: Suppose that Σ is satisfiable and S is a solution of Σ. By Lemma 4, 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, we can reduce the depth of S by at least 1 by Lemma 3 and apply the induction hypothesis.

5.2 IMPL

As with SAT, the interesting question is not whether the algorithm in Figure 3 correctly checks the entailment in the finite domain, but rather why the finite domain is sufficient. Here we focus on Theorem 4: IMPL($\Sigma_1 \vdash \Sigma_2$) is true iff every solution S of Σ_1 with $|S| \leq d = \max\{|\Sigma_1|, |\Sigma_2|\}$ is also a solution of Σ_2.

IMPL is more complicated than SAT due to the contravariance: given Σ_1, Σ_2 and $d = \max\{|\Sigma_1|, |\Sigma_2|\}$ then IMPL($\Sigma_1 \vdash \Sigma_2$) implies IMPL($\mathsf{Lround}(\Sigma_1, d) \vdash \mathsf{Lround}(\Sigma_2, d)$). Unfortunately, this is not enough.

The difficulty results from Lround not being injective: because half of the leaves are removed by $\mathsf{take_L}$, we cannot reconstruct the original. However, this drawback can be overcome by applying both Lround and Rround and then reconstructing the original tree using both rounded trees and the original tree depth. We encode the reconstruction operation in the "average" function $\bigtriangledown$.

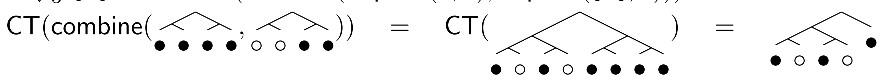
First, given two complete (non-canonical) trees of same depth, define the $\mathsf{combine}$ operation such that, $\mathsf{combine}(t_l, t_r)$, replaces each leaf l_l in t_l with $\widehat{l_l\ l_r}$, where l_r is the leaf corresponding to l_l in t_r.

Definition 3. $\bigtriangledown : CT_{T_\mathbb{B}} \times CT_{T_\mathbb{B}} \times \mathbb{N} \to CT_{T_\mathbb{B}}$
Given two trees, t_l, t_r and n, the depth of the original tree such that $\max\{|t_l|, |t_r|\} < n$*:*

$$t_l \bigtriangledown_n t_r \;= \mathsf{CT}(\mathsf{combine}(\mathsf{Expand}(t_l, n-1), \mathsf{Expand}(t_r, n-1)))$$

We illustrate with an example as before:

$$\bullet \bigtriangledown_3 \widehat{\circ\ \bullet} \quad = \quad \mathsf{CT}(\mathsf{combine}(\mathsf{Expand}(\bullet, 2), \mathsf{Expand}(\widehat{\circ\ \bullet}, 2))) \quad =$$



Lemma 5 gives the key properties of the average function $\bigtriangledown$:

Lemma 5.

1. *If $n > |t|$ then $t \bigtriangledown_n t = t$.*
2. *Let $n = |t|$ then $t_l = \mathsf{Lround}(t, n)$ and $t_r = \mathsf{Rround}(t, n)$ iff $t_l \bigtriangledown_n t_r = t$.*
3. *If $a_1 \oplus b_1 = c_1, a_2 \oplus b_2 = c_2$ and $n \geq \max\{|a_i|, |b_i|, |c_i|\}$ then $(a_1 \bigtriangledown_n a_2) \oplus (b_1 \bigtriangledown_n b_2) = (c_1 \bigtriangledown_n c_2)$.*

Proved in Coq. Lemma 5 contains the facts required to construct another solution S of Σ by using two solutions S_1, S_2 and the average function[4]. Moreover, $S = S_1 \bigtriangledown_n S_2$ is a solution of $\Sigma' = \Sigma \bigtriangledown_n \Sigma$, where n is $\max\{|S_1|, |S_2|\}$; in addition, in this case we also know that $\Sigma' = \Sigma$. Finally, $\bigtriangledown$ is the inverse function of both Lround and Rround.

Lemma 6. *Let S_1, S_2 be solutions of Σ and $n > \max\{|S_1|, |S_2|\}$. Then $S = S_1 \bigtriangledown_n S_2$ is also a solution of Σ.*

Direct from Lemma 5. We are now ready to attack the main theorem.

Theorem 4. *IMPL($\Sigma_1 \vdash \Sigma_2$) is true iff every solution S of Σ_1 with $|S| \leq d = \max\{|\Sigma_1|, |\Sigma_2|\}$ is also a solution of Σ_2.*

Proof.
$\Rightarrow$: Immediate.
$\Leftarrow$: It suffices to prove that for all $n \geq d$, all solutions S of Σ_1 with $|S| \leq n$ are also solutions of Σ_2. We apply strong induction on n starting from d:
The base case $n = d$ follows simply.
Induction step: Assume that each solution S of Σ_1 with $|S| \leq k$ is also a solution of Σ_2. Consider a solution S' of Σ_1 with depth $k+1$. Then $S_l = \mathsf{Lround}(S', k+1)$ is a solution of $\mathsf{Lround}(\Sigma_1, k+1) = \Sigma_1$ with $|S_l| \leq k$ (by Lemma 3). Similarly, $S_r = \mathsf{Rround}(S', k+1)$ is also a solution of Σ_1 with $|S_r| \leq k$. By our induction hypothesis, S_l and S_r are also solutions of Σ_2. Thus $S' = S_l \bigtriangledown_{k+1} S_r$ is also a solution of Σ_2 by Lemma 6.

6 Handling Non-zeros

For fear of cluttering the presentation we omitted showing how to guarantee that SAT and IMPL can ensure that certain variables be strictly positive.

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 the given variable is generated. This forces at least one of the boolean variables corresponding to the initial non-zero variable to be true in each considered system solution. In the tree domain, this clause ensures that the non-zero variable has at least one true leaf.

[4] $S = S_1 \bigtriangledown_n S_2 = \{x_i = v_i^1 \bigtriangledown_n v_i^2 | x_i = v_i^1 \in S_1, x_i = v_i^2 \in S_2\}$ and $\Sigma = \Sigma_1 \bigtriangledown_n \Sigma_2$ requires $\Sigma, \Sigma_1, \Sigma_2$ must contain the same variables and equations modulo different constants. The constants in Σ are constructed by applying the average function on the corresponding constants in Σ_1, Σ_2.

The full algorithms have an extra call to the NON_ZERO function which returns a conjunction of the clauses encoding the non-zero disjunctions:

```
$SAT(\Sigma)$
  $\Phi = REDUCE(\Sigma)$
  $\Phi_r = \Phi \wedge \mathsf{NON\_ZERO}(\Sigma)$
  $Return\ SAT\_SOLVER(\Phi_r)$
```

```
$IMPL(\Sigma_1, \Sigma_2)$
  $(\Phi_a, \Phi_c) = REDUCEI(\Sigma_1, \Sigma_2)$
  $\Phi'_a = \Phi_a \wedge \mathsf{NON\_ZERO}(\Sigma_1)$
  $\Phi'_c = \Phi_c \wedge \mathsf{NON\_ZERO}(\Sigma_2)$
  $\Phi''_a = ADD\_EXISTS(\Phi'_a,\ \Sigma_1)$
  $\Phi''_c = ADD\_EXISTS(\Phi'_c,\ \Sigma_2)$
  $Return\ \neg\ SAT\_SOLVER(\Phi''_a \wedge \neg\Phi''_c)$
```

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 $(v_1 = \widehat{\circ\ \bullet}; v_2 = \widehat{\bullet\ \circ})$ and $(v_1 = \widehat{\bullet\ \circ}; v_2 = \widehat{\circ\ \bullet})$. We have proved that for each non-zero variable,v, $\lceil log_2(n) \rceil$ 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 unfold 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. We packaged our solver as an OCaml library that implements the algorithms outlined in Figures 2 and 3 to resolve the SAT and IMPL queries issued by the entailment checker. As input, the queries take systems of equations, each system consisting of:

- a list of variable-constant pairs, denoting instantiations $(\pi = \chi)$
- a list of pairs of variables denoting equalities between variables $(\pi_1 = \pi_2)$
- a list of triples $a_i \oplus a_j = a_k$, where a_x are either variables or constants
- a list of variables that need to be strictly positive
- a list of variables that are existentially quantified

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.

$$\begin{array}{lcr}
\circ \oplus \circ = \circ \ ; \ \ \circ \oplus \bullet = \bullet & \Leftrightarrow & \mathsf{true} \\
(\circ \mid v) \oplus \bullet = \circ \ ; \ \ \circ \oplus \circ = \bullet \ ; \ \ \bullet \oplus \bullet = (\circ \mid \bullet \mid v) & \Leftrightarrow & \mathsf{false} \\
v \oplus \circ = \circ \ ; \ \ \circ \oplus \circ = v \ ; \ v \oplus \bullet = \bullet & \Leftrightarrow & v = \circ \\
v \oplus \circ = \bullet \ ; \ \ \circ \oplus \bullet = v & \Leftrightarrow & v = \bullet \\
v_1 \oplus v_2 = \circ & \Leftrightarrow & v_1 = v_2 = \circ \\
v_1 \oplus \circ = v_2 & \Leftrightarrow & v_1 = v_2 \\
v_1 \oplus \bullet = v_2 & \Leftrightarrow & v_1 = \circ \wedge v_2 = \bullet \\
v_1 \oplus v_2 = v_1 & \Leftrightarrow & v_2 = \circ \\
v_1 \oplus v_1 = v_2 & \Leftrightarrow & v_1 = v_2 = \circ
\end{array}$$

Fig. 4. Simplifications in the core

3. The backend: tasked with interfacing with the SAT solver: translating the output format from the core to the input format of the SAT 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].
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.

The core. The bulk of the core module translates equation systems into boolean formulas via the equation decomposition in §4 and various simplifications. The decomposition reduces the systems to constraints over trees of height 0, *i.e.*, booleans. At each step the solver applies both general (*e.g.*, rewriting equalities) and domain-specific simplifications such as those given in Figure 4. As we will see in §7.1, a considerable number of queries reduce to tautologies after repeated decomposition/simplification and can thus be discharged without the SAT 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 $v_1 \oplus v_2 = (\bullet | v_3)$. The completeness results from §5 prove that solving the join facts in the booleans is sufficient.

The back-end. In the backend, the reduced system is translated into a quantified conjunctive boolean formula. The $v_1 \oplus v_2 = \bullet$ clauses become $v_1 \ xor \ v_2$, while the $v_1 \oplus v_2 = v_3$ clauses become $(v_3 = v_1 \ xor \ v_2) \wedge \neg(v_1 \wedge v_2)$. To force a variable v to be strictly positive we require that at least one of the variables decomposed from v be true (*i.e.*, the tree value has at least one $\bullet$ leaf). If v_i ie the set of variables obtained from decomposing v then positivity is encoded by $\bigvee_i v_i$.

Finally, backends optimize to the particulars of their targeted SAT solver.

7.1 Evaluation

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 $\widehat{\circ\,\bullet} \oplus x = y$ it would deduce $\circ \leq x \leq \widehat{\bullet\,\circ}$ and $\widehat{\circ\,\bullet} \leq y$. 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.

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?

	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

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.

Standalone. While verifying programs, and their associated separation logic entailments is really The Point, it is not so easy to casually develop HIP and SLEEK input files that exercise the share provers aggressively. Accordingly, 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.