Logical Reasoning for Disjoint Permissions

Xuan-Bach Le[1] and Aquinas Hobor[1,2]

[1]National University of Singapore [2]Yale-NUS College

Abstract. Resource sharing is a fundamental phenomenon in concurrent programming where several threads have permissions to access a common resource. Logics for verification need to capture the notion of permission ownership and transfer. One typical practice is the use of rational numbers in $(0, 1]$ as permissions in which 1 is the full permission and the rest are fractional permissions. Rational permissions are not a good fit for separation logic because they remove the essential "disjointness" feature of the logic itself. We propose a general logic framework that supports permission reasoning in separation logic while preserving disjointness. Our framework is applicable to sophisticated verification tasks such as doing induction over the finiteness of the heap within the object logic or carrying out biabductive inference. We can also prove precision of recursive predicates within the object logic. We developed the ShareInfer tool to benchmark our techniques. We introduce "scaling separation algebras," a compositional extension of separation algebras, to model our logic, and use them to construct a concrete model.

1 Introduction

The last 15 years have witnessed great strides in program verification [43, 44, 39, 27, 7, 46]. One major area of focus has been concurrent programs following Concurrent Separation Logic (CSL) [40]. The key rule of CSL is Parallel:

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \qquad \{P_2\}\ c_2\ \{Q_2\}}{\{P_1 \star P_2\}\ c_1 || c_2\ \{Q_1 \star Q_2\}}\ \textsc{Parallel}$$

In this rule, we write $c_1||c_2$ to indicate the parallel execution of commands c_1 and c_2. The separating conjunction $\star$ indicates that the resources used by the threads is disjoint in some useful way, *i.e.* that there are no dangerous races. Many subsequent program logics [30, 45, 20, 18, 31] have introduced increasingly sophisticated notions of "resource disjointness" for the Parallel rule.

Fractional permissions (also called "shares") are a relatively simple enhancement to separation logic's original notion of disjointness [4]. Rather than owning a resource (e.g. a memory cell) entirely, a thread is permitted to own a part/fraction of that resource. The more of a resource a thread owns, the more actions it is permitted to take, a mapping called a *policy.* In this paper we will use the original policy of Bornat [4] to keep the examples straightforward: non-zero ownership of a memory cell permits reading while full ownership also permits writing. More modern logics allow for a variety of more flexible share policies [42, 13, 28], but our techniques still apply. Fractional permissions are less expressive than the "protocol-based" notions of disjointness used in program logics such as FCSL [44, 38], Iris [30], and TaDa [16], but are well-suited for common concurrent programming patterns such as read sharing and so have been incorporated into many program logics and verification tools [41, 36, 28, 26, 31, 19].

Since fractionals are simpler and more uniform than protocol-based logics, they are amenable to automation [26, 33]. However, previous techniques had difficulty with the inductive predicates common in SL proofs. We introduce *predicate multiplication*, a concise method for specifying the fractional sharing of complex predicates, writing $\pi \cdot P$ to indicate that we own the π-share of the arbitrary predicate P, *e.g.* $0.5 \cdot \mathsf{tree}(x)$ indicates a tree rooted at x and we own half of each of the nodes in the tree. If set up properly, predicate multiplication handles inductive predicates smoothly and is well-suited for automation because:

§3 it distributes with bientailments—*e.g.* $\pi \cdot (P \wedge Q) \dashv\vdash (\pi \cdot P) \wedge (\pi \cdot Q)$—enabling rewriting techniques and both forwards and backwards reasoning;
§4 it works smoothly with the inference process of biabduction [10]; and
§5 the side conditions required for bientailments and biabduction can be verified directly in the object logic, leveraging existing entailment checkers.

There has been significant work in recent years on tool support for protocol-based approaches [29, 19, 30, 15, 48], but they require significant user input and provide essentially no inference. Fractional permissions and protocol-based approaches are thus complementary: fractionals can handle large amounts of relatively simple concurrent code with minimal user guidance, while protocol-based approaches are useful for reasoning about the implementations of fine-grained concurrent data structures whose correctness argument is more sophisticated.

In addition to §3, §4, and §5, the rest of this paper is organized as follows.

§2 We give the technical background necessary for our work.
§6 We document ShareInfer [1], a tool that uses the logical tools developed in §3–§5 to infer frames and antiframes and check the necessary side conditions. We benchmark ShareInfer with 27 selective examples.
§7 We introduce *scaling separation algebra* that allows us to construct predicate multiplication on an abstract structure in a compositional way. We show such model can be constructed from Dockins *et al.*'s tree shares [21]. The key technical proofs in §5 and §7 have been verified in Coq [1].
§8 We prove that there are no useful share models that simultaneously satisfy disjointness and two distributivity axioms. Consequently, at least one axioms has to be removed, which we choose to be the left distributivity. We also prove that the failure of two-sided distributivity forces a side condition on a key proof rule for predicate multiplication.
§9 We discuss related work before delivering our conclusion.

2 Technical preliminaries

Share models. An (additive) share model $(\mathcal{S}, \oplus)$ is a partial commutative monoid with a bottom/empty element $\mathcal{E}$ and top/full element $\mathcal{F}$. On the rationals in

$$\begin{array}{l} \texttt{root} \overset{0.3}{\mapsto} (3, \texttt{left}, \texttt{right}) \;\star \\ \texttt{left} \overset{0.3}{\mapsto} (1, \mathsf{null}, \texttt{grand}) \;\star \\ \texttt{right} \overset{0.3}{\mapsto} (4, \texttt{grand}, \mathsf{null}) \;\star \\ \texttt{grand} \overset{0.6}{\mapsto} (1, \mathsf{null}, \mathsf{null}) \end{array}$$

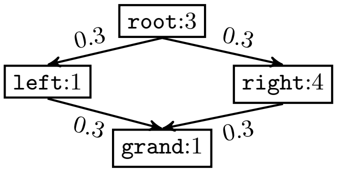

Fig. 1. This heap satisfies $\mathsf{tree}(\texttt{root}, 0.3)$ despite being a DAG

$[0, 1]$, $\oplus$ is partial addition, $\mathcal{E}$ is 0, and $\mathcal{F}$ is 1. We also require the existence of complements $\overline{\pi}$ satisfying $\pi \oplus \overline{\pi} = \mathcal{F}$; in $\mathbb{Q}$, $\overline{\pi} \overset{\text{def}}{=} 1 - \pi$.

Separation logic. Our base separation logic has the following connectives:

$$P, Q, \text{ etc.} \quad \overset{\text{def}}{=} \quad \langle F \rangle \mid P \wedge Q \mid P \vee Q \mid \neg P \mid P \star Q \mid \forall x.P \mid \exists x.P \mid \mu X.P \mid e_1 \overset{\pi}{\mapsto} e_2$$

Pure facts F are put in angle brackets, *e.g.* $\langle even(12) \rangle$. Pure facts force the empty heap, *i.e.* the usual separation logic emp predicate is just a macro for $\langle \top \rangle$. Our propositional fragment has (classical) conjunction $\wedge$, disjunction $\vee$, negation $\neg$, and the separating conjunction $\star$. We have both universal $\forall$ and existential $\exists$ quantifiers, which can be impredicative if desired. To construct recursive predicates we have the usual Tarski least fixpoint μ. The fractional points-to $e_1 \overset{\pi}{\mapsto} e_2$ means we own the π-fraction of the memory cell pointed to by e_1, whose contents is e_2, and nothing more. To distinguish points-to from emp we require that π be non-$\mathcal{E}$. For notational convenience we sometimes elide the full share $\mathcal{F}$ over a fractional maps-to, writing just $e_1 \mapsto e_2$. The connection of $\oplus$ to the fractional maps-to predicate is given by the bi-entailment:

$$\frac{}{e \overset{\pi_1}{\mapsto} e_1 \star e \overset{\pi_2}{\mapsto} e_2 \quad \dashv\vdash \quad e \overset{\pi_1 \oplus \pi_2}{\longmapsto} e_1 \wedge e_1 = e_2} \ \textsc{MapsTo Split}$$

Disjointness. Although intuitive, the rationals are not a good model for shares in SL. Consider this definition for π-fractional trees rooted at x:

$$\mathsf{tree}(x, \pi) \quad \overset{\text{def}}{=} \quad \langle x = \mathsf{null} \rangle \vee \exists d, l, r.\ x \overset{\pi}{\mapsto} (d, l, r) \star \mathsf{tree}(l, \pi) \star \mathsf{tree}(r, \pi) \qquad (1)$$

This tree predicate is obtained directly from the standard recursive predicate for binary trees 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 straightforward[1]. The problem is that when $\pi \in (0, 0.5]$, then tree can describe some non-tree directed acyclic graphs as in Figure 1. Fractional trees are a little too easy to introduce and thus unexpectedly painful to eliminate.

To prevent the deformation of recursive structures shown in Figure 1, we want to recover the "disjointness" property of basic SL: $e \mapsto e_1 \star e \mapsto e_2 \dashv\vdash \bot$. Disjointness can be specified either as an inference rule in separation logic [41] or as an algebraic rule on the share model [21] as follows:

$$\frac{}{e \overset{\pi}{\mapsto} e_1 \star e \overset{\pi}{\mapsto} e_2 \quad \dashv\vdash \quad \bot} \ \textsc{MapsTo Disjoint} \qquad\qquad \forall a, b.\ a \oplus a = b \quad \Rightarrow \quad a = \mathcal{E} \qquad (2)$$

[1] We write $x \overset{\pi}{\mapsto} (v_1, \ldots, v_n)$ for $x \overset{\pi}{\mapsto} v_1 \star (x+1) \overset{\pi}{\mapsto} v_2 \star \ldots \star (x + n - 1) \overset{\pi}{\mapsto} v_n$.

In other words, **a nonempty share π cannot join with itself**. In §3 we will see how disjointness enables the distribution of predicate multiplication over $\star$ and in §4 we will see how disjointness enables antiframe inference during biabduction.

Tree shares. Dockins *et al.* [21] proposed "tree shares" as a share model satisfying disjointness. For this paper the details of the model are not critical so we provide only a brief overview. A tree share $\tau \in \mathbb{T}$ is a binary tree with Boolean leaves, *i.e.* $\tau = \bullet \mid \circ \mid \widehat{\tau_1\,\tau_2}$, where $\circ$ is the empty share $\mathcal{E}$ and $\bullet$ is the full share $\mathcal{F}$. There are two "half" shares: $\widehat{\circ\,\bullet}$ and $\widehat{\bullet\,\circ}$, and four "quarter" shares, *e.g.* $\widehat{\widehat{\bullet\,\circ}\,\circ}$. Trees must be in *canonical form*, *i.e.*, the most compact representation under $\cong$:

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

Union $\sqcup$, intersection $\sqcap$, and complement $\bar{\cdot}$ are the basic operations on tree shares; they operate leafwise after unfolding the operands under $\cong$ into the same shape:

$$\widehat{\widehat{\bullet\,\circ}\,\circ} \sqcup \widehat{\widehat{\circ\,\bullet}\,\widehat{\bullet\,\circ}} \quad\cong\quad \widehat{\widehat{\bullet\,\circ}\,\widehat{\circ\,\circ}} \sqcup \widehat{\widehat{\circ\,\bullet}\,\widehat{\bullet\,\circ}} \quad=\quad \widehat{\widehat{\bullet\,\bullet}\,\widehat{\bullet\,\circ}} \quad\cong\quad \widehat{\bullet\,\widehat{\bullet\,\circ}}$$

The structure $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot}, \circ, \bullet \rangle$ forms a countable atomless Boolean algebra and thus enjoys decidable existential and first-order theories with precisely known complexity bounds [34]. The join operator $\oplus$ on trees is defined as $\tau_1 \oplus \tau_2 = \tau_3 \overset{\text{def}}{=} \tau_1 \sqcup \tau_2 = \tau_3 \wedge \tau_1 \sqcap \tau_2 = \circ$. Due to their good metatheoretic and computational properties, a variety of program logics [25, 24] and verification tools [3, 26, 33, 47] have used tree shares (or other isomorphic structures [19]).

3 Predicate multiplication

The additive structure of share models is relatively well-understood [33, 21, 34]. The focus for this paper is exploring the benefits and consequences of incorporating a multiplicative operator $\otimes$ into a share model. The simplest motivation for multiplication is computationally dividing some share π of a resource "in half;" the two halves of the resource are then given to separate threads for parallel processing. When shares themselves are rationals, $\otimes$ is just ordinary multiplication, *e.g.* we can divide $0.6 = (0.5 {\otimes} 0.6) {\oplus} (0.5 {\otimes} 0.6)$. Defining a notion of multiplication on a share model that satisfies disjointness is somewhat trickier, but we can do so with tree shares $\mathbb{T}$ as follows. Define $\tau_1 \otimes \tau_2$ to be the operation that replaces each $\bullet$ in τ_2 with a copy of τ_1, *e.g.*: $\widehat{\circ\,\bullet} \otimes \widehat{\widehat{\bullet\,\circ}\,\widehat{\bullet\,\circ}} = \widehat{\widehat{\widehat{\circ\,\bullet}\,\circ}\,\widehat{\widehat{\circ\,\bullet}\,\circ}}$. The structure $(\mathbb{T}, \oplus, \otimes)$ is a kind of "near-semiring." The $\otimes$ operator is associative, has identity $\mathcal{F}$ and null point $\mathcal{E}$, and is right distributive, *i.e.* $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$. It is not commutative, does not distribute on the left, or have inverses. It is hard to do better: adding axioms like multiplicative inverses forces any model satisfying disjointness ($\forall a, b.\ a \oplus a = b \ \Rightarrow\ a = \mathcal{E}$) to have no more than two elements (§8).

```
 1 struct tree {int d; struct tree* l; struct tree* r;};
 2 void processTree(struct tree* x) {
 3   if (x == 0) { return; }
 4   print(x -> d);                    ||7   print(x -> d);
 5   processTree(x -> l);              ||8   processTree(x -> l);
 6   processTree(x -> r);              ||9   processTree(x -> r);
10 }
```

Fig. 2. The parallel `processTree` function, written in a C-like language

Now consider the toy program in Figure 2. Starting from the tree rooted at `x`, the program itself is dead simple. First (line 3) we check if the `x` is null, *i.e.* if we have reached a leaf; if so, we `return`. If not, we split into parallel threads (lines 4–6 and 7–9) that do some processing on the root data in both branches. In the toy example, the processing just `print`s out the root data (lines 4 and 7); the `print` command is unimportant: what is important that we somehow access some of the data in the tree. After processing the root, both parallel branches call the `processTree` function recursively on the left `x->l` (lines 5 and 8) and right `x->r` (lines 6 and 9) branches, respectively. After both parallel processes have terminated, the function returns (line 10). The program is simple, so we would like its verification to be equally simple.

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

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

Here $\mathsf{tree}(x) \overset{\text{def}}{=} \langle x = \mathsf{null} \rangle \vee \exists d, l, r.\ x \mapsto (d, l, r) \star \mathsf{tree}(l) \star \mathsf{tree}(r)$ is exactly the usual definition of binary trees in separation logic. Predicate multiplication has allowed us to isolate the fractional ownership from the definition; compare with equation (1) above. Our precondition and postcondition both say that x is a pointer to a heap-represented π-owned tree. Critically, we want to ensure that our π-share at the end of the program is equal to the π-share at the beginning. This way if our initial caller had full $\mathcal{F}$ ownership before calling `processTree`, he will have full ownership afterwards (allowing him to *e.g.* deallocate the tree).

The intuition behind the proof is simple. First in line 3, we check if `x` is null; if so we are in the base case of the tree definition and can simply return. If not we can eliminate the left disjunct and can proceed to split the $\star$-separated bits into disjoint subtrees `l` and `r`, and then dividing the ownership of those bits into two “halves”. Let $\mathcal{L} \overset{\text{def}}{=} \widehat{\bullet\,\circ}$ and $\mathcal{R} \overset{\text{def}}{=} \overline{\mathcal{L}} = \widehat{\circ\,\bullet}$. When we start the parallel computation on lines 4 and 7 we want to pass the left branch of the computation the $\mathcal{L} \otimes \pi$-share of the spatial resources, and the right branch of the computation the $\mathcal{R} \otimes \pi$. In both branches we then need to show that we can read from the data cell, which in the simple policy we use for this paper boils down to making sure that the product of two non-$\mathcal{E}$ shares cannot be $\mathcal{E}$. This is a basic property for reasonable share models with multiplication. In the remainder of the parallel code (lines 5–6 and 8–9) we need to make recursive calls, which is done by simply instantiating π with $\mathcal{L} \otimes \pi$ and $\mathcal{R} \otimes \pi$ in the recursive specification (as well as `l`

$$\frac{P \vdash Q}{\pi \cdot P \vdash \pi \cdot Q}\ \textsc{Dot Pos} \qquad \frac{}{\pi \cdot \langle P \rangle \dashv\vdash \langle P \rangle}\ \textsc{Dot Pure} \qquad \frac{}{\pi \cdot (P \Rightarrow Q) \vdash (\pi \cdot P) \Rightarrow (\pi \cdot Q)}\ \textsc{Dot Impl}$$

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

$$\frac{\tau \neq \emptyset}{\pi \cdot \big(\forall x : \tau.\ P(x)\big) \dashv\vdash \forall x : \tau.\ \pi \cdot P(x)}\ \textsc{Dot Univ} \qquad \frac{}{\pi \cdot \big(\exists x : \tau.\ P(x)\big) \dashv\vdash \exists x : \tau.\ \pi \cdot P(x)}\ \textsc{Dot Exis}$$

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

$$\frac{\mathrm{precise}(P)}{(\pi_1 \oplus \pi_2) \cdot P \dashv\vdash (\pi_1 \cdot P) \star (\pi_2 \cdot P)}\ \textsc{Dot Plus} \qquad \frac{P \vdash \mathsf{uniform}(\pi') \qquad Q \vdash \mathsf{uniform}(\pi')}{\pi \cdot (P \star Q) \dashv\vdash (\pi \cdot P) \star (\pi \cdot Q)}\ \textsc{Dot Star}$$

Fig. 3. Distributivity of the scaling operator over pure and spatial connectives

and `r` for x). The later half proof after the parallel call is pleasantly symmetric to the first half in which we fold back the original tree predicate by merging the two halves $\mathcal{L} \otimes \pi$ and $\mathcal{R} \otimes \pi$ back into π. Consequently, we arrive at the postcondition $\pi \cdot \mathsf{tree}(x)$, which is identical to the precondition.

3.1 Proof rules for predicate multiplication

In Figure 4 we put the formal verification for `processTree`, which follows the informal argument very closely. However, before we go through it, let us consider the reason for this alignment: because the key rules for reasoning about predicate multiplication are bidirectional. These rules are given in Figure 3. The non-spatial rules are all straightforward and follow the basic pattern that predicate multiplication both pushes into and pulls out of the operators of our logic without meaningful side conditions. The DotPure rule means that predicate multiplication ignores pure facts, too. Complicating the picture slightly, predicate multiplication pushes into implication $\Rightarrow$ but does not pull out of it. Combining DotImpl with DotPure we get a one-way rule for negation: $\pi \cdot (\neg P) \vdash \neg \pi \cdot$. We will explain why we cannot get both directions in §5.1 and §8.

Most of the spatial rules are also simple. Recall that $\mathsf{emp} \overset{\mathrm{def}}{=} \langle \top \rangle$, so DotPure yields $\pi \cdot \mathsf{emp} \dashv\vdash \mathsf{emp}$. The DotFull rule says that $\mathcal{F}$ is the scalar identity on predicates, just as it is the multiplicative identity on the share model itself. The DotDot rule allows us to "collapse" repeated predicate multiplication using share multiplication; we will shortly see how we use it to verify the recursive calls to `processTree`. Similarly, the DotMapsTo rule shows how predicate multiplication combines with basic maps-to by multiplying the associated shares together. All three rules are bidirectional and require no side conditions.

While the last two rules are both bidirectional, they both have side conditions. The DotPlus rule shows how predicate multiplication distributes over $\oplus$. The $\vdash$ direction does not require a side condition, but the $\dashv$ direction we require that P be *precise* in the usual separation logic sense. Precision will be discussed in §5.2; for now a simple counterexample shows why it is necessary:

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

The premise is also consistent with $x \overset{\mathcal{L}}{\mapsto} a \star (x+1) \overset{\mathcal{R}}{\mapsto} b$.

The DotStar rule shows how predicate multiplication distributes into and out of the separating conjunction $\star$. It is also bidirectional. **Crucially, the $\dashv$ direction fails on non-disjoint share models like** $\mathbb{Q}$, which is the "deeper reason" for the deformation of recursive structures illustrated in Figure 1. On disjoint share models like $\mathbb{T}$, we get equational reasoning $\dashv\vdash$ subject to the side condition of *uniformity*. Informally, $P \vdash \mathsf{uniform}(\pi')$ asserts that any heap that satisfies P has the permission π' uniformly at each of its defined addresses. In §8 we explain why we cannot admit this rule without a side condition.

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

$$\begin{array}{lll}
\mathsf{slist}(x) & \dashv\vdash & \exists d,n.\big((\langle d = 17\rangle \star x \overset{\mathcal{L}}{\mapsto} (d,n)) \vee (\langle d \neq 17\rangle \star x \overset{\mathcal{R}}{\mapsto} (d,n))\big) \star \mathsf{slist}(n) \\
\mathsf{dlist}(x) & \dashv\vdash & \exists d,n.x \mapsto d,n \star \mathcal{L}\cdot \mathsf{dlist}(n) \\
\mathsf{dtree}(x) & \dashv\vdash & \exists d,l,r.x \mapsto d,l,r \star \mathcal{L}\cdot \mathsf{dtree}(l) \star \mathcal{R}\cdot \mathsf{dtree}(r)
\end{array}$$

The $\mathsf{slist}(x)$ predicate owns different amounts of permissions at different memory cells depending on the value of those cells. The $\mathsf{dlist}(x)$ predicate owns decreasing amounts of the list, *e.g.* the first cell is owned more than the second, which is owned more than the third. The $\mathsf{dtree}(x)$ predicate is even stranger, owning different amounts of different branches of the tree, essentially depending on the path to the root. None of these predicates mix well with DotStar, but perhaps they are not useful to verify many programs in practice, either. In §5.1 and §5.2 we will discuss how to prove predicates are precise and uniform. In §5.4 will demonstrate our techniques to do so by applying them to two examples.

3.2 Verification of `processTree` using predicate multiplication

We now explain how the proof of `processTree` is carried out in Fig. 4 using scaling rules in Fig. 3. In line 2, we unfold the definition of predicate $\mathsf{tree}(x)$ which consists of one base case and one inductive case. We reach line 3 by pushing π inward using various rules DotPure, DotDisj, DotExis, DotMapsto and DotStar. To use DotStar we must prove that $\mathsf{tree}(x)$ is $\mathcal{F}$-uniform, which we show how to do in §5.4. We prove this lemma once and use it many times.

The base base $\mathtt{x} = \mathtt{null}$ is handled in lines 4-5 by applying rule DotPure, *i.e.*, $\langle \mathtt{x} = \mathtt{null}\rangle \vdash \pi \cdot \langle \mathtt{x} = \mathtt{null}\rangle$ and then DotPos, $\pi \cdot \langle \mathtt{x} = \mathtt{null}\rangle \vdash \pi \cdot \mathsf{tree}(x)$.

```
1  void processTree(struct tree* x) { // $\{\ \pi\cdot\mathsf{tree}(\mathtt{x})\ \}$
2  // $\left\{\pi\cdot\Big(\langle\mathtt{x}=\mathtt{null}\rangle\ \vee\ (\exists d,l,r.\ \mathtt{x}\mapsto(d,l,r)\ \star\ \mathsf{tree}(l)\ \star\ \mathsf{tree}(r))\Big)\right\}$
3  // $\left\{\Big\langle\mathtt{x}=\mathtt{null}\Big\rangle\ \vee\ \Big(\exists d,l,r.\ \mathtt{x}\overset{\pi}{\mapsto}(d,l,r)\ \star\ (\pi\cdot\mathsf{tree}(l))\ \star\ (\pi\cdot\mathsf{tree}(r))\Big)\right\}$
4    if (x == null) { // $\{\langle\mathtt{x}=\mathtt{null}\rangle\}$
5    return;} // $\{\ \pi\cdot\mathsf{tree}(\mathtt{x})\ \}$
6  // $\left\{\mathtt{x}\overset{\pi}{\mapsto}(d,l,r)\ \star\ (\pi\cdot\mathsf{tree}(l))\ \star\ (\pi\cdot\mathsf{tree}(r))\right\}$
7  // $\left\{\mathcal{F}\cdot\Big(\mathtt{x}\overset{\pi}{\mapsto}(d,l,r)\ \star\ (\pi\cdot\mathsf{tree}(l))\ \star\ (\pi\cdot\mathsf{tree}(r))\Big)\right\}$
8  // $\left\{(\mathcal{L}\oplus\mathcal{R})\cdot\Big(\mathtt{x}\overset{\pi}{\mapsto}(d,l,r)\ \star\ (\pi\cdot\mathsf{tree}(l))\ \star\ (\pi\cdot\mathsf{tree}(r))\Big)\right\}$
9  // $\left\{\begin{array}{l}\Big(\mathcal{L}\cdot\Big(\mathtt{x}\overset{\pi}{\mapsto}(d,l,r)\ \star\ (\pi\cdot\mathsf{tree}(l))\ \star\ (\pi\cdot\mathsf{tree}(r))\Big)\Big)\ \star\\ \Big(\mathcal{R}\cdot\Big(\mathtt{x}\overset{\pi}{\mapsto}(d,l,r)\ \star\ (\pi\cdot\mathsf{tree}(l))\ \star\ (\pi\cdot\mathsf{tree}(r))\Big)\Big)\end{array}\right\}$
10 // $\left\{\mathcal{L}\cdot\Big(\mathtt{x}\overset{\pi}{\mapsto}(d,l,r)\ \star\ (\pi\cdot\mathsf{tree}(l))\ \star\ (\pi\cdot\mathsf{tree}(r))\Big)\right\}$      ||
11 // $\left\{\mathcal{L}\cdot\mathtt{x}\overset{\pi}{\mapsto}(d,l,r)\ \star\ \mathcal{L}\cdot\pi\cdot\mathsf{tree}(l)\ \star\ \mathcal{L}\cdot\pi\cdot\mathsf{tree}(r)\right\}$      ||
12 // $\left\{\mathtt{x}\overset{\mathcal{L}\otimes\pi}{\longmapsto}(d,l,r)\ \star\ ((\mathcal{L}\otimes\pi)\cdot\mathsf{tree}(l))\ \star\ ((\mathcal{L}\otimes\pi)\cdot\mathsf{tree}(r))\right\}$      ||
13   print(x -> d);                                                    || ...
14   processTree(x -> l);  processTree(x -> r);                        ||
15 // $\left\{\mathtt{x}\overset{\mathcal{L}\otimes\pi}{\longmapsto}(d,l,r)\ \star\ ((\mathcal{L}\otimes\pi)\cdot\mathsf{tree}(l))\ \star\ ((\mathcal{L}\otimes\pi)\cdot\mathsf{tree}(r))\right\}$      ||
16 // $\{\mathcal{L}\cdot\pi\cdot\mathtt{x}\mapsto(d,l,r)\ \star\ \mathcal{L}\cdot\pi\cdot\mathsf{tree}(l)\ \star\ \mathcal{L}\cdot\pi\cdot\mathsf{tree}(r)\}$      ||
17 // $\{\mathcal{L}\cdot\pi\cdot(\mathtt{x}\mapsto(d,l,r)\ \star\ \mathsf{tree}(l)\ \star\ \mathsf{tree}(r))\}$      ||
18 // $\left\{\begin{array}{l}\Big(\mathcal{L}\cdot\pi\cdot(\mathtt{x}\mapsto(d,l,r)\ \star\ \mathsf{tree}(l)\ \star\ \mathsf{tree}(r))\Big)\ \star\\ \Big(\mathcal{R}\cdot\pi\cdot(\mathtt{x}\mapsto(d,l,r)\ \star\ \mathsf{tree}(l)\ \star\ \mathsf{tree}(r))\Big)\end{array}\right\}$
19 // $\left\{(\mathcal{L}\oplus\mathcal{R})\cdot\pi\cdot\Big(\mathtt{x}\mapsto(d,l,r)\ \star\ \mathsf{tree}(l)\ \star\ \mathsf{tree}(r)\Big)\right\}$
20 } // $\{\ \pi\cdot\mathsf{tree}(\mathtt{x})\ \}$
```

Fig. 4. Reasoning with the scaling operator $\pi\cdot P$.

For the inductive case, we first apply DotFull in line 7 and then replace $\mathcal{F}$ with $\mathcal{L}\oplus\mathcal{R}$ (recall that $\mathcal{R}$ is $\mathcal{L}$'s compliment). On line 9 we use DotPlus to translate the split on shares with $\oplus$ into a split on heaps with $\star$.

We show only one parallel process; the other is a mirror image. Line 10 gives the precondition from the Parallel rule, and then in lines 11 and 12 we continue to "push in" the predicate multiplication. To verify the code in lines 13–14 just requires Frame. Notice that we need the DotDot rule to "collapse" the two uses of predicate multiplication into one so that we can apply the recursive specification (with the new π' in the recursive precondition equal to $\mathcal{L}\otimes\pi$).

Having taken the predicate completely apart, it is now necessary to put Humpty Dumpty back together again. Here is why it is vital that all of our proof rules are bidirectional, without which we would not be able to reach the final postcondition $\pi\cdot\mathsf{tree}(x)$. The final wrinkle is that for line 19 we must prove the precision of the $\mathsf{tree}(x)$ predicate. We show how to do so with example in §5.4, but typically in a verification this is proved once per predicate as a lemma.

4 Bi-abductive inference with fractional permissions

Biabduction is a separation logic inference process that helps to increase the scalability of verification for sizable programs [22, 49]; in recent years it has been the focus of substantial research for (sequential) separation logic [10, 8, 11, 32]. Biabduction aims to infer the missing information in an incomplete separation logic entailment. More precisely, given an incomplete entailment $A \star [??] \vdash B \star [??]$, we would like to find predicates for the two missing pieces [??] that complete the entailment in a nontrivial manner. The first piece is called the *antiframe* while the second is the *inference frame*. The standard approach consists of two sequential subroutines, namely the *abductive inference* and *frame inference* to construct the antiframe and frame respectively. Our task in this section is to show how to upgrade these routines to handle fractional permissions so that biabduction can extend to concurrent programs. As we will see, disjointness plays a crucial role in antiframe inference.

4.1 Fractional residue computation

Consider the fractional point-to bi-abduction problem with rationals:

$$a \xmapsto{\pi_1} b \star [??] \vdash a \xmapsto{\pi_2} b \star [??]$$

There are three cases to consider, namely $\pi_1 = \pi_2$, $\pi_1 < \pi_2$ or $\pi_1 > \pi_2$. In the first case, both the (minimal) antiframe F_{a} and frame F_{f} are emp; for the second case we have $F_{\mathrm{a}} = \mathsf{emp}$, $F_{\mathrm{f}} = a \xmapsto{\pi_2 - \pi_1} b$ and the last case gives us $F_{\mathrm{a}} = a \xmapsto{\pi_1 - \pi_2} b, F_{\mathrm{f}} = \mathsf{emp}$. Here we straightforwardly compute the residue permission using rational subtraction. In general, one can attempt to define subtraction $\ominus$ from a share model $\langle \mathcal{S}, \oplus \rangle$ as $a \ominus b = c \overset{\text{def}}{=} b \oplus c = a$. However, this definition is too coarse as we want subtraction to be a total function so that the residue is always computable efficiently. A solution to this issue is to relax the requirements for $\ominus$, asking only that it satisfies the following two properties:

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

where $a \ll b \overset{\text{def}}{=} \exists c.\ a \oplus c = b$. The condition C_1 provides a convenient way to compute the fractional residue in both the frame and antiframe while C_2 asserts that $a \ominus b$ is effectively the minimal element that when joined with b becomes greater than a. In the rationals $\mathbb{Q}$, $a \ominus b \overset{\text{def}}{=} \mathit{if}\,(a > b)\ \mathit{then}\ a - b\ \mathit{else}\ 0$. On tree shares $\mathbb{T}$, $a \ominus b \overset{\text{def}}{=} a \sqcap \bar{b}$. Recalling that the case when $\pi_1 = \pi_2$ is simple (both the antiframe and frame are just emp), then if $\pi_1 \neq \pi_2$ we can compute the fractional antiframe and inference frames uniquely using $\ominus$:

$$\frac{}{a \xmapsto{\pi_1} b \star a \xmapsto{\pi_2 \ominus \pi_1} b \vdash a \xmapsto{\pi_2} b \star a \xmapsto{\pi_1 \ominus \pi_2} b}\ \textsc{Msub}$$

Generally, the following rule helps compute the residue of predicate P:

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

Using C_1 and C_2 it is easy to prove that the residue is minimal w.r.t. $\ll$, *i.e.*:

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

4.2 Extension of predicate axioms

To support reasoning over recursive data structure such as lists or trees, the assertion language is enriched with the corresponding inductive predicates. To derive properties over inductive predicates, verification tools often contain a list of predicate axioms/facts and use them to aid the verification process [9, 32]. These facts are represented as entailment rules $A \vdash B$ that can be classified into "folding" and "unfolding" rules to manipulate the representation of inductive predicates. For example, some axioms for the tree predicate are:

$$F_1 : x = 0 \wedge \mathsf{emp} \vdash \mathsf{tree}(x) \quad F_2 : x \mapsto (v, x_1, x_2) \star \mathsf{tree}(x_1) \star \mathsf{tree}(x_2) \vdash \mathsf{tree}(x)$$
$$U : \mathsf{tree}(x) \wedge x \neq 0 \vdash \exists v, x_1, x_2.\ x \mapsto (v, x_1, x_2) \star \mathsf{tree}(x_1) \star \mathsf{tree}(x_2)$$

We want to transform these axioms into fractional forms. The key ingredient is the DotPos rule from Fig. 3, that lifts the fractional portion of an entailment, *i.e.* $(P \vdash Q) \Rightarrow (\pi \cdot P \vdash \pi \cdot Q)$. Using this and the other scaling rules from Fig. 3, we can upgrade the folding/unfolding rules into corresponding fractional forms:

$$F_1' : x = 0 \wedge \mathsf{emp} \vdash \pi \cdot \mathsf{tree}(x) \quad F_2' : x \overset{\pi}{\mapsto} (v, x_1, x_2) \star \pi \cdot \mathsf{tree}(x_1) \star \pi \cdot \mathsf{tree}(x_2) \vdash \pi \cdot \mathsf{tree}(x)$$
$$U : \mathsf{tree}(x) \wedge x \neq 0 \vdash \exists v, x_1, x_2.\ x \mapsto (v, x_1, x_2) \star \mathsf{tree}(x_1) \star \mathsf{tree}(x_2)$$

As our scaling rules are bi-directional, they can be applied both in the antecedent and consequent to produce a smooth transformation to fractional axioms. Also, recall that our DotStar rule $\pi \cdot (P \star Q) \dashv\vdash \pi \cdot P \star \pi \cdot Q$ has a side condition that both P and Q are π'-uniform. This condition is trivial in the transformation as standard predicates (*i.e.* those without permissions) are automatically $\mathcal{F}$-uniform. Furthermore, the precision and uniformity properties can be transferred directly to fractional forms by the following rules:

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

4.3 Abductive inference and frame inference

To construct the antiframe, Calcagno *et al.* [10] presented a general framework for antiframe inference which contains rules of the form:

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

$$x \xmapsto{\pi_1} (v, a, x_2) \star \pi_2 \cdot \mathsf{tree}(x_1) \star (x_2 = 0 \land \mathsf{emp}) \star [??] \;\vdash\; \pi_3 \cdot \mathsf{tree}(x) \star [??]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(x_2 = 0 \land \mathsf{emp}) \star [\mathsf{emp}] \rhd \mathsf{emp}}\;\textsc{Base}}{(x_2 = 0 \land \mathsf{emp}) \star [\mathsf{emp}] \rhd \pi_3 \cdot \mathsf{tree}(x_2)}\;\mathsf{F_1}'}{\pi_2 \cdot \mathsf{tree}(x_1) \star (x_2 = 0 \land \mathsf{emp}) \star [(\pi_3 \ominus \pi_2) \cdot \mathsf{tree}(x_1)] \rhd \pi_3 \cdot \mathsf{tree}(x_1) \star \pi_3 \cdot \mathsf{tree}(x_2)}\;\textsc{Psub}}{\begin{array}{c} x \xmapsto{\pi_1} (v, a, x_2) \star \pi_2 \cdot \mathsf{tree}(x_1) \star (x_2 = 0 \land \mathsf{emp}) \star [(\pi_3 \ominus \pi_2) \cdot \mathsf{tree}(x_1) \\ \star x \xmapsto{\pi_3 \ominus \pi_1} (v, a, x_2)] \rhd x \xmapsto{\pi_3} (v, a, x_2) \star \pi_3 \cdot \mathsf{tree}(x_1) \star \pi_3 \cdot \mathsf{tree}(x_2) \end{array}}\;\textsc{Msub}}{\begin{array}{c} x \xmapsto{\pi_1} (v, a, x_2) \star \pi_2 \cdot \mathsf{tree}(x_1) \star (x_2 = 0 \land \mathsf{emp}) \\ \star\;\; [a = x_1 \land (\pi_3 \ominus \pi_2) \cdot \mathsf{tree}(x_1) \star x \xmapsto{\pi_3 \ominus \pi_1} (v, a, x_2)] \rhd \pi_3 \cdot \mathsf{tree}(x) \end{array}}\;\begin{array}{c}\textsc{match}\\ +\mathrm{F}_2'\end{array}$$

Abductive inference

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{emp} \rhd \mathsf{emp} \star [\mathsf{emp}]}\;\textsc{Base}}{x \xmapsto{\pi_1 \oplus (\pi_3 \ominus \pi_1)} (v, x_1, x_2) \rhd x \xmapsto{\pi_3} (v, x_1, x_2) \star [x \xmapsto{(\pi_1 \ominus \pi_3)} (v, x_1, x_2)]}\;\textsc{Msub}}{\begin{array}{c} x \xmapsto{\pi_1 \oplus (\pi_3 \ominus \pi_1)} (v, x_1, x_2) \star (\pi_2 \oplus (\pi_3 \ominus \pi_2)) \cdot \mathsf{tree}(x_1) \rhd \\ x \xmapsto{\pi_3} (v, x_1, x_2) \star \pi_3 \cdot \mathsf{tree}(x_1) \star [x \xmapsto{(\pi_1 \ominus \pi_3)} (v, x_1, x_2) \star (\pi_2 \ominus \pi_3) \cdot \mathsf{tree}(x_1)] \end{array}}\;\textsc{Psub}}{\begin{array}{c} x \xmapsto{\pi_1 \oplus (\pi_3 \ominus \pi_1)} (v, x_1, x_2) \star (\pi_2 \oplus (\pi_3 \ominus \pi_2)) \cdot \mathsf{tree}(x_1) \star (x_2 = 0 \land \mathsf{emp}) \rhd \\ x \xmapsto{\pi_3} (v, x_1, x_2) \star \pi_3 \cdot \mathsf{tree}(x_1) \star \pi_3 \cdot \mathsf{tree}(x_2) \star [x \xmapsto{(\pi_1 \ominus \pi_3)} (v, x_1, x_2) \star (\pi_2 \ominus \pi_3) \cdot \mathsf{tree}(x_1)] \end{array}}\;\mathrm{F}_1'$$

Frame inference

Fig. 5. An example of biabduction with fractional permissions

where Cond is the side condition, together with consequents (H, H'), heap formulas $(\varDelta, \varDelta')$ and antiframes (M, M'). In principle, the abduction algorithm gradually matches fragments of consequent with antecedent, derives sound equalities among variables while applying various folding and unfolding rules for recursive predicates in both sides of the entailment. Ideally, the remaining unmatched fragments of the antecedent are returned to form the antiframe. During the process, certain conditions need to be maintained, *e.g.*, satisfiability of the antecedent or minimal choice for antiframe. After finding the antiframe, the inference process is invoked to construct the inference frame. In principle, the old antecedent is first combined with the antiframe to form a new antecedent whose fragments are matched with the consequent. Eventually, the remaining unmatched fragments of the antecedent are returned to construct the inference frame.

The discussion of fractional residue computation in §4.1 and extension of recursive predicate rules in §4.2 ensure a smooth upgrade of the biabduction algorithm to fractional form. We demonstrate this intuition using the example in Fig. 5. The partial consequent is a fractional $\mathsf{tree}(x)$ predicate with permission π_3 while the partial antecedent is star conjunction of a fractional maps-to predicate of address x with permission π_1, a fractional $\mathsf{tree}(x_1)$ predicate with permission π_2 and a null pointer x_2. Following the spirit of Calcagno *et al.* [10], the steps in both sub-routines include applying the folding and unfolding rules for predicate tree and then matching the corresponding pair of fragments from antecedent and consequent. On the other hand, the upgraded part is reflected through the use of the two new rules MSUB and PSUB to compute the fractional residues as well as a more general system of folding and unfolding rules for predicate tree. We are then able to compute the antiframe $a = x_1 \wedge (\pi_3 \ominus \pi_2) \cdot \mathsf{tree}(x_1) \star x \xmapsto{\pi_3 \ominus \pi_2} (v, a, x_2)$ and the inference frame $x \xmapsto{\pi_1 \ominus \pi_3} (v, x_1, x_2) \star (\pi_2 \ominus \pi_3) \cdot \mathsf{tree}(x_1)$ respectively.

Antiframe inference and disjointness. Consider the following abduction problem:

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

Using the folding rule F_2, we can identify the antiframe as $\mathsf{tree}(x_2)$. Now suppose we have a rational permission $\pi \in \mathbb{Q}$ distributed everywhere, *i.e.*:

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

A naïve solution is to let the antiframe be $\pi \cdot \mathsf{tree}(x_2)$. However, in $\mathbb{Q}$ this choice is unsound due to the deformation of recursive structures issue illustrated in Fig. 1: if the antiframe is $\pi \cdot \mathsf{tree}(x_2)$, the left hand side can be a DAG, even though the right hand side must be a tree. However, in disjoint share models like $\mathbb{T}$, choosing $\pi \cdot \mathsf{tree}(x_2)$ for the antiframe is correct and the entailment holds. As is often the case, things are straightforward once the definitions are correct.

5 A proof theory for fractional permissions

Our main objective in this section is to show how to discharge the uniformity and precision side conditions required by the DOTSTAR and DOTPLUS rules. To handle recursive predicates like $\mathsf{tree}(x)$ we develop set of novel modal-logic based proof rules to carry out induction in the heap. To allow tools to leverage existing entailment checkers, all of these techniques are done **in the object logic itself**, rather than in the metalogic. Thus, in §5, we do not assume a concrete model for our object logic (in §7 we will develop a model).

First we discuss new proof rules for predicate multiplication and fractional maps-to (§5.1), precision (§5.2), and induction over fractional heaps (§5.3). We then conclude (§5.4) with two examples of proving real properties using our proof theory: that $\mathsf{tree}(x)$ is $\mathcal{F}$-uniform and that $\mathsf{list}(x)$ is precise. Some of the theorems have delicate proofs, so all of them have been verified in Coq [1].

5.1 Proof theory for predicate multiplication and fractional maps-to

In §3 we presented the key rules that someone who wants to verify programs using predicate multiplication is likely to find convenient. On page 13 we present a series of additional rules, mostly used to establish the "uniform" and "precise" side conditions necessary in our proofs.

Figure 6 is the simplest group, giving basic facts about the fractional points-to predicate. Only $\mapsto$ INVERSION is not immediate from the nonfractional case. It says that it is impossible to have two fractional maps-tos of the same address and with two different values. We need this fact to *e.g.* prove that predicates with existentials such as tree are precise.

$$\frac{}{(x \overset{\pi}{\mapsto} y_1 \star \top) \wedge (x \overset{\pi'}{\mapsto} y_2 \star \top) \vdash |y_1 = y_2|}\ \overset{\mapsto}{\textsc{inversion}} \qquad \frac{}{x \overset{\pi}{\mapsto} y \vdash \neg\mathsf{emp}}\ \overset{\mapsto}{\mathsf{emp}} \qquad \frac{}{x \overset{\pi}{\mapsto} y \vdash |x \neq \mathtt{null}|}\ \overset{\mapsto}{\mathtt{null}}$$

Fig. 6. Proof theory for fractional maps-to

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

$$\frac{P \vdash \mathsf{uniform}(\pi)}{\pi' \cdot P \vdash \mathsf{uniform}(\pi' \otimes \pi)}\ \mathsf{uniform}\textsc{Dot} \qquad \frac{}{\mathrm{precise}(x \overset{\pi}{\mapsto} y)}\ \overset{\mapsto}{\textsc{precise}}$$

$$\frac{}{x \overset{\pi}{\mapsto} y \vdash \mathsf{uniform}(\pi)}\ \overset{\mapsto}{\mathsf{uniform}} \qquad \frac{\mathrm{precise}(P)}{\mathrm{precise}(\pi \cdot P)}\ \begin{matrix}\textsc{Dot}\\ \textsc{precise}\end{matrix}$$

Fig. 7. Uniformity and precision for predicate multiplication

$$\frac{G \vdash \mathsf{precisely}(P) \quad G \vdash \mathsf{precisely}(Q)}{G \vdash \mathsf{precisely}(P \star Q)}\ \mathsf{precisely}\star \qquad \frac{\top \vdash \mathsf{precisely}(P)}{\mathrm{precise}(P)}\ \begin{matrix}\mathsf{precisely}\\ \textsc{Precise}\end{matrix}$$

$$\frac{}{\mathsf{precisely}(P) \vdash \big((P \star Q) \wedge (P \star R)\big) \Rightarrow \big(P \star (Q \wedge R)\big)}\ \begin{matrix}\mathsf{precisely}\\ \textsc{Left}\end{matrix} \qquad \frac{\exists x. \Big(G \vdash \mathsf{precisely}\big(P(x)\big)\Big)}{G \vdash \mathsf{precisely}\big(\forall x. P(x)\big)}\ \mathsf{precisely}\forall$$

$$\frac{\forall Q, R. \Big(G \vdash \big((P \star Q) \wedge (P \star R)\big) \Rightarrow \big(P \star (Q \wedge R)\big)\Big)}{G \vdash \mathsf{precisely}(P)}\ \begin{matrix}\mathsf{precisely}\\ \textsc{Right}\end{matrix} \qquad \frac{G \vdash \mathsf{precisely}(P)}{G \vdash \mathsf{precisely}(P \wedge Q)}\ \mathsf{precisely}\wedge$$

$$\frac{\begin{matrix}\forall x. \Big(G \vdash \mathsf{precisely}\big(P(x)\big)\Big) \\ \forall x, y. \Big(G \wedge \big(P(x) \star \top\big) \wedge \big(P(y) \star \top\big) \vdash |x = y|\Big)\end{matrix}}{G \vdash \mathsf{precisely}\big(\exists x. P(x)\big)}\ \mathsf{precisely}\exists \qquad \frac{\begin{matrix}G \vdash \mathsf{precisely}(P) \\ G \vdash \mathsf{precisely}(Q) \\ G \wedge (P \star \top) \wedge (Q \star \top) \vdash \bot\end{matrix}}{G \vdash \mathsf{precisely}(P \vee Q)}\ \mathsf{precisely}\vee$$

Fig. 8. Proof theory for precision

$$\frac{}{\circledcirc P \vdash P}\ \mathrm{T} \qquad \frac{}{\circledcirc P \vdash \circledcirc\circledcirc\, P}\ \circledcirc\circledcirc \qquad \frac{}{\rhd_\pi P \vdash \rhd_\pi \rhd_\pi\, P}\ \rhd_\pi \rhd_\pi$$

$$\frac{\rhd_\pi P \vdash P}{\top \vdash P}\ \mathrm{W} \qquad \frac{}{\rhd_\pi P \dashv\vdash \rhd_\pi \circledcirc P}\ \rhd_\pi \circledcirc \qquad \frac{}{\rhd_\pi P \dashv\vdash \circledcirc \rhd_\pi P}\ \circledcirc \rhd_\pi$$

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

Fig. 9. Proof theory for substructural induction

Proving the side conditions for DotPlus *and* DotStar. Figure 7 contains some rules for establishing that P is π-uniform (*i.e.* $P \vdash \mathsf{uniform}(\pi)$) and that P is precise. Since uniformity is a simple property, the rules are easy to state:

To use predicate multiplication we will need to prove two kinds of side conditions: $\mathsf{uniform/emp}$ tells us that emp is π-uniform for all π; the conclusion (all defined heap locations are held with share π) is vacuously true. The $\mathsf{uniform}$Dot rule tells us that if P is π-uniform then when we multiply P by a fraction π' the result is $(\pi' \otimes \pi)$-uniform. The $\mapsto$ $\mathsf{uniform}$ rule tells us that points-to is uniform. The $\mathsf{uniform}\star$ rule possesses interesting characteristics. The $\dashv$ direction follows from $\mathsf{uniform/emp}$ and the $\star\mathsf{emp}$ rule ($P \star \mathsf{emp} \dashv\vdash P$). The $\vdash$ direction is not automatic but very useful. One consequence is that from $P \vdash \mathsf{uniform}(\pi)$ and $Q \vdash \mathsf{uniform}(\pi)$ we can prove $P \star Q \vdash \mathsf{uniform}(\pi)$. The $\vdash$ direction follows from disjointness but fails over non-disjoint models such as rationals $\mathbb{Q}$.

The $\mapsto$ precise rule tells us that points-tos are precise. The DotPrecise rule is a partial solution to proving precision. It states that $\pi \cdot P$ is precise if and only if P is precise. We will next show how to prove that P itself is precise.

5.2 Proof theory for proving that predicates are precise

Proving that a predicate is π-uniform is relatively straightforward using the proof rules presented so far. However, proving that a predicate is precise is not as pleasant. Traditionally precision is defined (and checked for concrete predicates) in the metalogic [40] using the following definition:

$$\text{precise}(P) \overset{\text{def}}{=} \forall h, h_1, h_2.\ h_1 \subseteq h \Rightarrow h_2 \subseteq h \Rightarrow (h_1 \models P) \Rightarrow (h_2 \models P) \Rightarrow h_1 = h_2 \quad (3)$$

Here we write $h_1 \subseteq h_2$ to mean that h_1 is a subheap of h_2, *i.e.* $\exists h'.h_1 \oplus h' = h_2$, where $\oplus$ is the joining operation on the underlying separation algebra [21]. Essentially precision is a kind of uniqueness property: if a predicate P is precise then it can only be true on a single subheap.

Rather than checking precision in the metalogic, we wish to do so in the object logic. We give a proof theory that lets us do so in Figure 8. Among other advantages, proving precision in the object logic lets tools build on existing separation logic entailment checkers to prove the precision of recursive predicates. The core idea is simple: we define a new object logic operator "$\mathsf{precisely}(P)$" that captures the notion of precision relativized to the current heap; essentially it is a partially applied version of the definition of precise(P) in equation (3):

$$h \models \mathsf{precisely}(P) \overset{\text{def}}{=} \forall h_1, h_2.h_1 \subseteq h \Rightarrow h_2 \subseteq h \Rightarrow (h_1 \models P) \Rightarrow (h_2 \models P) \Rightarrow h_1 = h_2 \quad (4)$$

Although we have given $\mathsf{precisely}$'s model to aid intuition, we emphasize that in §5 all of our proofs take place in the object logic; we never unfold $\mathsf{precisely}$'s definition. Note that $\mathsf{precisely}$ is also generally weaker than the typical notion of precision. For example, the predicate $x \mapsto 7 \vee y \mapsto 7$ is not precise; however the entailment $z \mapsto 8 \vdash \mathsf{precisely}(x \mapsto 7 \vee y \mapsto 7)$ is provable from Figure 8.

That said, two notions are closely connected as given in the $\mathsf{precisely}$Precise rule. We also give introduction $\mathsf{precisely}$Right and elimination rules $\mathsf{precisely}$Left that make a connection between precision and an "antidistribution" of $\star$ over $\wedge$.

We also give a number of rules for showing how $\mathsf{precisely}$ combines with the connectives of our logic. The rules for propositional $\wedge$ and separating $\star$ conjunction follow well-understood patterns, with the addition of an arbitrary premise context G being the key feature. The rule for disjunction $\vee$ is a little trickier, with an additional premise that forces the disjunction to be exclusive rather than inclusive. An example of such an exclusive disjunction is in the standard definition of the tree predicate, where the first disjunct $\langle x = \mathtt{null}\rangle$ is fundamentally incompatible with the second disjunct $\exists d, l, r.x \mapsto d, l, r \star \ldots$ since $\mapsto$ does not allow the address to be $\mathtt{null}$ (by rule $\mapsto \mathtt{null}$ from Figure 6). The rules for universal quantification $\forall$ existential quantification $\exists$ are essentially generalizations of the rules for the traditional conjunction $\wedge$ and disjunction $\vee$.

It is now straightforward to prove the precision of simple predicates such as $\langle x = \mathtt{null}\rangle \vee (\exists y.x \mapsto y \star y \mapsto 0)$. Finding and proving the key lemmas that enable the proof of the precision of recursive predicates remains a little subtle.

5.3 Proof theory for induction over the finiteness of the heap

Recursive predicates such as $\mathsf{list}(x)$ and $\mathsf{tree}(x)$ are common in SL. However, proving properties of such predicates, such as proving that $\mathsf{list}(x)$ is precise, is a little tricky since the μFoldUnfold rule provided by the Tarski fixed point does not automatically provide an induction principle. Generally speaking such properties follow by some kind of induction argument, either over auxiliary parameters (*e.g.* if we augment trees to have the form $\mathsf{tree}(x, \tau)$, where τ is an inductively-defined type in the metalogic) or over the finiteness of the heap itself. Both arguments usually occur in the metalogic rather than the object logic.

We have two contributions to make for proving inductive properties. First, we show how to do induction over the heap in a fractional setting. Intuitively this is more complicated than in the non-fractional case because there are infinite sequences of strictly smaller subheaps. That is, for a given initial heap h_0, there are infinite sequences h_1, h_2, ... such that $h_0 \supsetneq h_1 \supsetneq h_2 \supsetneq \ldots$. The disjointness property does not fundamentally change this issue, so we illustrate with an example with the shares in $\mathbb{Q}$. The heap h_0 satisfying $x \overset{1}{\mapsto} y$ is strictly larger than the heap h_1 satisfying $x \overset{\frac{1}{2}}{\mapsto} y$, which is strictly larger than the heap h_2 satisfying $x \overset{\frac{1}{4}}{\mapsto} y$; in general h_i satisfies $x \overset{\frac{1}{2^i}}{\mapsto} y$. Since our sequence is infinite, we cannot use it as the basis for an induction argument. The solution is that we require that the heaps decrease by at least some constant size c. If each heap subsequent heap must shrink by at least *e.g.* $c = 0.25$ of a memory cell then the sequence must be finite just as in the non-fractional case, *i.e.* $c = \mathcal{F}$. More sophisticated approaches are conceivable (*e.g.* limits) but they are not easy to automate and we did not find any practical examples that require such methods.

Our second contribution is the development of a proof theory in the object logic that can carry out these kinds of induction proofs in a relatively straightforward way. The proof rules that let us do so are given in Figure 9. Once good lemmas are identified, we find doing induction proofs over the finite heap formally in the object logic simpler than doing the same proofs in the metalogic.

The key to our induction rules is two new operators: “within” $\circledcirc$ and “shrinking” $\rhd_\pi$. Essentially $\rhd_\pi P$ is used as an induction guard, preventing us from applying our induction hypothesis P until we are on a π-smaller subheap. When $\pi = \mathcal{F}$ we sometimes write just $\rhd P$. Semantically, if h satisfies $\rhd_\pi P$ then P is true **on all strict subheaps of h that are smaller by at least a π-piece**. Accordingly, the key elimination rule $\rhd_\pi\star$ may seem natural: it verifies that the induction guard is satisfied and unlocks the underlying hypothesis. To start an induction proof to prove an arbitrary goal $\top \models P$, we use the rule W to introduce an induction hypothesis, resulting in the new entailment goal of $\rhd_\pi P \vdash P$.

Some definitions, such as $\mathsf{list}(x)$, have only one “recursive call”; others, such as $\mathsf{tree}(x)$ have more than one. Moreover, sometimes we wish to apply our inductive hypothesis immediately after satisfying the guard, whereas other times it is convenient to satisfy the guard somewhat before we need the inductive hypothesis. To handle both of these issues we use the “within” operator $\circledcirc$ such that $h \models \circledcirc P$ means P is true on all subheaps of h, which is the intuition behind the rule $\circledcirc\star$. To apply our induction hypothesis somewhat after meeting its guard (or if we wish to apply it more than once) we use the $\rhd_\pi\circledcirc$ rule to add the $\circledcirc$ modality before eliminating the guard. We will see an example of this shortly.

5.4 Using our proof theory

We now turn to two examples of using our proof theory from page 13 to demonstrate that the rule set is strong and flexible enough to prove real properties.

Proving that $\mathsf{tree}(x)$ *is $\mathcal{F}$-uniform.* Our logical rules for induction and uniformity are able to establish the uniformity of predicates in a fairly simple way. Here we focus on the $\mathsf{tree}(x)$ predicate because it is a little harder due to the two recursive “calls” in its unfolding. For convenience, we will write $\mathsf{u}(\pi)$ instead of $\mathsf{uniform}(\pi)$.

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

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

The proof breaks into two cases. The first reduces to $\langle x = \mathtt{null}\rangle \wedge (\rhd \cdots) \vdash \mathsf{u}(\mathcal{F})$, which follows from $\mathsf{uniform}/\mathsf{emp}$ rule. The second case reduces to $\big(x \mapsto (d,l,r) \star \mathsf{tree}(l) \star \mathsf{tree}(r)\big) \wedge \big(\rhd \circledcirc \forall x.\mathsf{tree}(x) \Rightarrow \mathsf{u}(\mathcal{F})\big) \vdash \mathsf{u}(\mathcal{F})$. Then the $\mathsf{uniform}\star$ rule gives

$$\big(x \mapsto (d,l,r) \star (\mathsf{tree}(l) \star \mathsf{tree}(r))\big) \wedge \big(\rhd \circledcirc \forall x.\mathsf{tree}(x) \Rightarrow \mathsf{u}(\mathcal{F})\big) \vdash \mathsf{u}(\mathcal{F}) \star \mathsf{u}(\mathcal{F})$$

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

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

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

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

Fig. 10. Key lemmas we use to prove recursive predicates precise

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

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

We again split over $\star$ to reach two essentially identical cases. We apply rule T to remove the $\circledcirc$ and then reach *e.g.* $\forall x.\mathsf{tree}(x) \Rightarrow \mathsf{u}(\mathcal{F}) \vdash \mathsf{tree}(l) \Rightarrow \mathsf{u}(\mathcal{F})$, which is immediate. Further details on this proof can be found in the full paper [2].

Proving that $\mathsf{list}(x)$ *is precise.* Precision is more complex than π-uniformity, so it is harder to prove. We will use the simpler $\mathsf{list}(x)$ as an example; the additional trick we need to prove that $\mathsf{tree}(x)$ is precise are applications of the $\triangleright_\pi\circledcirc$ and $\circledcirc\star$ rules in the same manner as the proof that $\mathsf{tree}(x)$ is $\mathcal{F}$-uniform. We have proved that both $\mathsf{list}(x)$ and $\mathsf{tree}(x)$ are precise using our proof rules in Coq [1].

In Figure 10 we give four key lemmas used in our proof[2]. All four are derived (with a little cleverness) from the proof rules given in Figure 8. We sketch the proof as follows. To prove precise($\mathsf{list}(x)$) we first use the preciselyPrecise rule to transform the goal into $\top \vdash \mathsf{precisely}(\mathsf{list}(x))$. We cannot immediately apply rule W, however, since without a concrete $\star$-separated conjunct **outside** the precisely, we cannot dismiss the inductive guard with the $\triangleright_\pi\star$ rule. Accordingly, we next use lemma (A) and standard natural deduction to reach the goal $\top \vdash \forall x.(\mathsf{list}(x) \star \top) \Rightarrow \mathsf{precisely}(\mathsf{list}(x))$, after which we apply rule W with $\pi = \mathcal{F}$.

Afterwards we do some standard natural deduction steps yielding the goal

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

We are now in a position to apply lemma (B) to break up the conjunction. We now have three goals. The first goal is that $\langle x = \texttt{null}\rangle$ is precise, which follows from the fact that emp is precise, which in turn can be proved using the rule preciselyRight. The third goal is that the two branches of the disjunction are mutually incompatible, which follows from $\langle x = \texttt{null}\rangle$ being incompatible with

[2] We abuse notation by reusing the inference rule format to present derived lemmas.

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

Fig. 11. Evaluation of our proof systems using ShareInfer

maps-to using rule $\mapsto\texttt{null}$. The second (and last remaining) goal needs to use lemma (C) twice to break up the existentials. Two of the three new goals are to show that the two existentials are uniquely determined, which follow from $\mapsto$ INVERSION, leaving the goal

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

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

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

We now use $\triangleright_\pi \star$ rule to defeat the inductive guard. The rest is straightforward. Further details on this proof can be found in the full paper [2].

6 The ShareInfer fractional biabduction engine

Having described our logical machinery in §3–§5, we now demonstrate that our techniques are well-suited to automation by documenting our ShareInfer prototype [1]. Our tool is capable of checking whether a user-defined recursive predicate such as list or tree is uniform and/or precise and then conducting biabductive inference over a separation logic entailment containing said predicates.

To check uniformity, the tool first uses heuristics to guess a potential tree share candidate π and then applies proof rules in Fig. 7 and 6 to derive the goal $\mathsf{uniform}(\pi)$. To support more flexibility, our tool also allows users to specify the candidate share π manually. To check precision, the tool maneuvers over the proof rules in Fig. 6 and 8 to achieve the desired goal. In both cases, recursive predicates are handled with the rules in Fig. 9. ShareInfer returns either Yes, No or Unknown together with a human-readable proof of its claim.

For bi-abduction, ShareInfer automatically checks precision and uniformity whenever it encounters a new recursive predicate. If the check returns Yes, the tool will unlock the corresponding rule, *i.e.*, DotPlus for precision and DotStar for uniformity. ShareInfer then matches fragments between the consequent and antecedent while applying folding and unfolding rules for recursive predicates to construct the antiframe and inference frame respectively. For instance, here is the biabduction problem contained in file $\mathsf{bi\_tree2}$ (see Fig. 11):

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

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

$\mathsf{ShareInfer}$ is around 2.5k LOC of Java. We benchmarked it with 27 selective examples from three categories: precision, uniformity and bi-abduction. The benchmark was conducted with a 3.4 GHz processor and 16 GB of memory. Our results are given in Fig. 11. Despite the complexity of our proof rules our performance is reasonable: $\mathsf{ShareInfer}$ only took 75.9 milliseconds to run the entire example set, or around 2.8 milliseconds per example. Our benchmark is small, but this performance indicates that more sophisticated separation logic verifiers such as HIP/SLEEK [14] or Infer [9] may be able to use our techniques at scale.

7 Building a model for our logic

Our task now is to provide a model for our proof theories. We present our models in several parts. In §7.1 we begin with a brief review of Cancellative Separation Algebras (CSA). In §7.2 we explain what we need from our fractional share models. In §7.3 we develop an extension to CSAs called "Scaling Separation Algebras" (SSA). In §7.5 we develop the machinery necessary to support our rules for object-level induction over the heap. We have verified in Coq [1] that the models in §7.1 support the rules in Figure 8, the models in §7.3 support the rules Figures 3 and 7, and the models in §7.5 support the rules in Figure 9.

7.1 Cancellative separation algebras

A Separation Algebra (SA) is a set H with an associative, commutative partial operation $\oplus$. Separation algebras can have a single unit or multiple units; we use $\mathit{identity}(x)$ to indicate that x is a unit. A Cancellative SA $\langle H, \oplus\rangle$ further requires that $a \oplus b_1 = c \Rightarrow a \oplus b_2 = c \Rightarrow b_1 = b_2$. We can define a partial order on H using $\oplus$ by $h_1 \subseteq h_2 \overset{\text{def}}{=} \exists h'. h_1 \oplus h' = h_2$. Calcagno *et al.* [12] showed that CSAs can model separation logic with the definitions

$$h \models P \star Q \overset{\text{def}}{=} \exists h_1, h_2.\ h_1 \oplus h_2 = h \wedge (h_1 \models P) \wedge (h_2 \models Q) \;\text{ and }\; h \models \mathsf{emp} \overset{\text{def}}{=} \mathit{identity}(h).$$

The standard definition of precise(P) was given as equation (3) in §5.2, together with the definition for our new $\mathsf{precisely}(P)$ operator in equation (4). What is difficult here is finding a set of axioms (Figure 8) and derivable lemmas (*e.g.* Figure 10) that are strong enough to be useful in the object-level inductive proofs. Once the axioms are found, proving them from the model given is straightforward. Cancellation is not necessary to model basic separation logic [18], but we need it to prove the introduction $\mathsf{precisely}\textsc{Right}$ and elimination rules $\mathsf{precisely}\textsc{Left}$ for our new operator.

7.2 Fractional share algebras

A fractional share algebra $\langle S, \oplus, \otimes, \mathcal{E}, \mathcal{F}\rangle$ (FSA) is a set S with two operations: partial addition $\oplus$ and total multiplication $\otimes$. The substructure $\langle S, \oplus\rangle$ is a CSA with the single unit $\mathcal{E}$. For the reasons discussed in §2 we require that $\oplus$ satisfies the disjointness axiom $a \oplus a = b \Rightarrow a = \mathcal{E}$. Furthermore, we require that the existence of a top element $\mathcal{F}$, representing complete ownership, and assume that each element $s \in S$ has a complement $\overline{s}$ such that $s \oplus \overline{s} = \mathcal{F}$.

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

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

The tree share model we present in §2 satisfies all of the above axioms, so we have a nontrivial model. As we will see shortly, it would be very convenient if we could assume that $\otimes$ also distributed on the left, or if we had multiplicative inverses on the left rather than merely cancelation on the right. However, we will see in §8.2 that both assumptions are untenable.

7.3 Scaling separation algebra

A scaling separation algebra (SSA) is $\langle H, S, \oplus_H, \oplus_S, \otimes_S, \mathcal{E}, \mathcal{F}, \mathit{mul}, \mathit{force}\rangle$, where $\langle H, \oplus_H\rangle$ is a CSA for heaps and $\langle S, \oplus_S, \otimes_S, \mathcal{E}, \mathcal{F}\rangle$ is a FSA for shares. Intuitively, $\mathit{mul}(\pi, h_1)$ multiplies every share inside h_1 by π and returns the result h_2. The multiplication is on the left, so for each original share π' in h_1, the resulting share in h_2 is $\pi \otimes_S \pi'$. Recall that the informal meaning of $\pi \cdot P$ is that we have a π-fraction of predicate P. Formally this notion relies on a little trick:

$$h \models \pi \cdot P \quad \overset{\text{def}}{=} \quad \exists h'.\ \mathit{mul}(\pi, h') = \pi \wedge h' \models P \tag{5}$$

A heap h contains a π-fraction of P if there is a **bigger** heap h' satisfying P, and multiplying that bigger heap h' by the scalar π gets back to the smaller heap h.

The simpler $\mathit{force}(\pi, h_1)$ overwrites all shares in h_1 with the constant share π to reach the resulting heap h_2. We use force to define the $\mathsf{uniform}$ predicate as $h \models \mathsf{uniform}(\pi) \overset{\text{def}}{=} \mathit{force}(\pi, h) = h$. A heap h is π-uniform when setting all the shares in h to π gets you back to h—*i.e.*, they must have been π to begin with.

We need to understand how all of the ingredients in an SSA relate to each other to prove the core logical rules on page 13. We distill the various relationships we need to model our logic in Figure 12. Although there are a goodly number of them, most are reasonably intuitive.

$S_1.\ \mathit{force}(\pi, \mathit{force}(\pi', a)) = \mathit{force}(\pi, a)$	$S_2.\ \mathit{force}(\pi, \mathit{mul}(\pi', a)) = \mathit{force}(\pi, a)$
$S_3.\ \mathit{mul}(\pi, \mathit{force}(\pi', a)) = \mathit{force}(\pi \otimes_S \pi', a)$	$S_4.\ \mathit{mul}(\pi, \mathit{mul}(\pi', a)) = \mathit{mul}(\pi \otimes_S \pi', a)$
$S_5.\ \mathit{identity}(a) \Rightarrow \mathit{force}(\pi, a) = a$	$S_6.\ a \subseteq_H \mathit{force}(\mathcal{F}, a)$
$S_7.\ \pi_1 \subseteq_S \pi_2 \Rightarrow \mathit{force}(\pi_1, a) \subseteq_H \mathit{force}(\pi_2, a)$	$S_8.\ \mathit{force}(\pi, a) \oplus_H \mathit{force}(\pi, b) = c \Rightarrow \mathit{force}(\pi, c) = c$
$S_9.\ \mathit{identity}(a) \Rightarrow \mathit{mul}(\pi, a) = a$	$S_{10}.\ \mathit{mul}(\mathcal{F}, a) = a$
$S_{11}.\ \mathit{mul}(\pi, a_1) = \mathit{mul}(\pi, a_2) \Rightarrow a_1 = a_2$	$S_{12}.\ \mathit{mul}(\pi, a) \subseteq_H a$

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

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

Fig. 12. The 14 additional axioms for scaling separation algebras beyond those inherited from cancellative separation algebras

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

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

S'_{14} is a kind of dual for S_{13}, *i.e.* it would correspond to a **left** distributivity property of $\oplus_S$ over $\otimes_S$ in the share model into heaps. Unfortunately, as we will see in §8.2, the disjointness of $\oplus_S$ is incompatible with simultaneously supporting both left and right distributivity. Accordingly, S_{14} weakens S'_{14} so that it only holds when a and b are π'-uniform (which by S_8 forces c to be π'-uniform). We also wish we could satisfy S'_{15}: $\forall \pi, a. \exists b. \mathit{mul}(\pi, b) = a$, which corresponds to left multiplicative inverses, but again (§8.2) disjointness is incompatible.

7.4 Compositionality of scaling separation algebras

Despite their complex axiomatization, we gain two advantages from developing SSAs rather than directly proving our logical axioms on a concrete model. First, they give us a precise understanding of exactly which operations and properties (S_1–S_{14}) are used to prove the logical axioms. Second, following Dockins *et al.* [21] we can build up large SSAs compositionally from smaller SSAs.

To do so cleanly it will be convenient to consider a slight variant of SSAs, "Weak SSAs" that allow, but do not require, the existence of identity elements in the underlying CSA model. A WSSA satisfies exactly the same axioms as an SSA, except that we use the weaker $\subseteq_H$ definition we defined for permission algebras, *i.e.* $a_1 \subseteq_H a_2 \overset{\text{def}}{=} (\exists a'. a_1 \oplus_H a' = a_2) \vee (a_1 = a_2)$. Note that S_5 and

S_9 are vacuously true when the CSA does not have identity elements. We need identity elements to prove the logical axioms from the model; we only use WSSAs to gain compositionality as we construct a suitable final SSA. Keeping the share components $\langle S, \oplus_S, \otimes_S, \mathcal{E}, \mathcal{F}\rangle$ constant, we give three SSA constructors to get a flavor for what we can do with the remaining components $\langle H, \oplus_H, \mathit{force}, \mathit{mul}\rangle$.

Example 1 (Shares). The share model $\langle S, \oplus_S\rangle$ is an SSA, and the positive (non-$\mathcal{E}$) shares $\langle S^+, \oplus\rangle$ are a WSSA, with $\mathit{force}_S(\pi, \pi') \overset{\text{def}}{=} \pi$ and $\mathit{mul}_S(\pi, \pi') \overset{\text{def}}{=} \pi \otimes \pi'$.

Example 2 (Semiproduct). Let $\langle A, \oplus_A, \mathit{force}_A, \mathit{mul}_A\rangle$ be an SSA/WSSA, and B be a set. Define $(a_1, b_1) \oplus_{A\times B} (a_2, b_2) = (a_3, b_3) \overset{\text{def}}{=} a_1 \oplus_A a_2 = a_3 \wedge b_1 = b_2 = b_3$, $\mathit{force}_{A\times B}(\pi, (a, b)) \overset{\text{def}}{=} (\mathit{force}_A(\pi, a), b)$, and $\mathit{mul}_{A\times B}(\pi, (a, b)) \overset{\text{def}}{=} (\mathit{mul}_A(\pi, a), b)$. Then $\langle A \times B, \oplus_{A\times B}, \mathit{force}_{A\times B}, \mathit{mul}_{A\times B}\rangle$ is an SSA/WSSA.

Example 3 (Finite partial map). Let A be a set and $\langle B, \oplus_B, \mathit{force}_B, \mathit{mul}_B\rangle$ be an SSA/WSSA. Define $f \oplus_{A \overset{\mathsf{fin}}{\rightharpoonup} B} g = h$ pointwise [21]. Define $\mathit{force}_{A \overset{\mathsf{fin}}{\rightharpoonup} B}(\pi, f) \overset{\text{def}}{=} \lambda x.\mathit{force}_B(\pi, f(x))$ and likewise define $\mathit{mul}_{A \overset{\mathsf{fin}}{\rightharpoonup} B}(\pi, f) \overset{\text{def}}{=} \lambda x.\mathit{mul}_B(\pi, f(x))$. The structure $\langle A \overset{\mathsf{fin}}{\rightharpoonup} B, \oplus_{A \overset{\mathsf{fin}}{\rightharpoonup} B}, \mathit{force}_{A \overset{\mathsf{fin}}{\rightharpoonup} B}, \mathit{mul}_{A \overset{\mathsf{fin}}{\rightharpoonup} B}\rangle$ is an SSA.

Using these constructors, $A \overset{\mathsf{fin}}{\rightharpoonup} (S^+, V)$, *i.e.* finite partial maps from addresses to pairs of positive shares and values, is an SSA and thus can support a model for our logic. We also support other standard constructions *e.g.* sum types $+$.

7.5 Model for inductive logic

What remains is to give the model that yields the inductive logic in Figure 9. The key induction guard modal $\rhd_\pi$ operator is defined as follows:

$$\begin{aligned} h_1\ S_\pi\ h_4 &\overset{\text{def}}{=} \exists h_2, h_3.\ h_1 \supseteq_H h_2 \wedge h_3 \oplus_H h_4 = h_2 \wedge (h_3 \models \mathsf{uniform}(\pi) \wedge \neg\mathsf{emp}) \\ h \models \rhd_\pi P &\overset{\text{def}}{=} \forall h'.\ (h\ S_\pi\ h') \Rightarrow (h' \models P) \end{aligned}$$

In other words, $\rhd_\pi$ is a (boxy) modal operator over the relation S_π, which relates a heap h_1 with all heaps that are strict subheaps that are smaller by at least a π-piece. The model is a little subtle to enable the rules $\rhd_\pi\circledcirc$ and $\circledcirc\rhd_\pi$ that let us handle multiple recursive calls and simplify the engineering. The within operator $\circledcirc$ is much simpler to model:

$$h_1\ W\ h_2 \overset{\text{def}}{=} h_1 \supseteq_H h_2 \qquad\qquad h \models \circledcirc P \overset{\text{def}}{=} \forall h'.\ (h\ W\ h') \Rightarrow (h' \models P)$$

All of the rules in Figure 9 follow from these definitions except for rule W. To prove this rule, we require that the heap model have an additional operator. The "π-quantum", written $|h|_\pi$, gives the number of times a non-empty π-sized piece can be taken out of h. For disjoint shares, the number of times is no more than the number of defined memory locations in h. We require two facts for $|h|_\pi$. First, that $h_1 \subseteq_H h_2 \Rightarrow |h_1|_\pi \leq |h_2|_\pi$, *i.e.* that subheaps do not have larger π-quanta than their parent. Second, that $h_1 \oplus_H h_2 = h_3 \Rightarrow (h_2 \models \mathsf{uniform}(\pi) \wedge \neg\mathsf{emp}) \Rightarrow |h_3|_\pi > |h_1|_\pi$, *i.e.* that taking out a π-piece strictly decreases the number of π-quanta. Given this setup, rule W follows immediately by induction on $|h|_\pi$. The rules that require the longest proofs in the model are $\rhd_\pi\circledcirc$ and $\circledcirc\rhd_\pi$.

8 Lower bounds on predicate multiplication

In §7 we gave a model for the logical axioms we presented in Figure 3 and on page 13. Our goal here is to show that it is difficult to do better, *e.g.* by having a premise-free DotStar rules or a bidirectional DotImpl rule. In §8.1 we show that these logical rules force properties on the share model. In §8.2 we show that disjointness puts restrictions on the class of share models. There are no non-trivial models that have left inverses or satisfy both left and right distributivity.

8.1 Predicate multiplication's axioms force share model properties

The SSA structures we gave in §7.3 are good for building models that enable the rules for predicate multiplication from Figure 3. However, since they impose intermediate algebraic and logical signatures between the concrete model and rules for predicate multiplication, they are not good for showing that we cannot do better. Accordingly here we disintermediate and focus on the concrete model $A \overset{\mathsf{fin}}{\rightharpoonup} (S^+, V)$, that is finite partial maps from addresses to pairs of positive shares and values. The join operations on heaps operates pointwise [21], with $(\pi_1, v_1) \oplus (\pi_2, v_2) = (\pi_3, v_3) \overset{\text{def}}{=} \pi_1 \oplus_S \pi_2 = \pi_3 \wedge v_1 = v_2 = v_3$, from which we derive the usual SA model for $\star$ and emp (§7.1). We define $h \models x \overset{\pi}{\mapsto} y \overset{\text{def}}{=} \mathit{dom}(h) = \{x\} \wedge h(x) = (\pi, y)$. We define scalar multiplication over heaps $\otimes_H$ pointwise as well, with $\pi_1 \otimes (\pi_2, v) \overset{\text{def}}{=} (\pi_1 \otimes_S \pi_2, v)$, and then define predicate multiplication by $h \models \pi \cdot P \overset{\text{def}}{=} \exists h'.\ h' = \pi \otimes_H h' = h \wedge h' \models P$. All of the above definitions are standard except for $\otimes_H$, which strikes us as the only choice (up to commutativity), and predicate multiplication itself.

By §7 we already know that this model satisfies the rules for predicate multiplication, given the assumptions on the share model from §7.2. What is interesting is that we can prove the other direction: if we assume that the key logical rules from Figure 3 hold, they force axioms on the share model. The key correspondences are: DotFull forces that $\mathcal{F}$ is the left identity of $\otimes_S$; DotMapsTo forces that $\mathcal{F}$ is the right identity of $\otimes_S$; DotMapsTo forces the associativity of $\otimes_S$; the $\dashv$ direction of DotConj forces the right cancellativity of $\otimes_S$ (as does DotImpl and the $\dashv$ direction of DotUniv); and DotPlus, which forces right distributivity of $\otimes_S$ over $\oplus_S$.

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

$$\frac{}{\pi \cdot (P \star Q) \dashv\vdash (\pi \cdot P) \star (\pi \cdot Q)}\ \textsc{Dot Star}' \qquad \frac{}{\pi \cdot (P \Rightarrow Q) \dashv (\pi \cdot P) \Rightarrow (\pi \cdot Q)}\ \textsc{Dot Impl}'$$

The $\dashv$ direction of DotStar′ also forces that $\oplus_S$ satisfies disjointness; this is the key reason that we cannot use rationals $\langle (0,1], +, \times \rangle$. Clearly the side-condition-free DotStar′ rule is preferable to the DotStar in Figure 3, and it would also be preferable to have bidirectionality for predicate multiplication over implication and negation. Unfortunately, as we will see shortly, the disjointness of $\oplus_S$ places strong multiplicative algebraic constraints on the share model. These constraints are the reason we cannot support the DotImpl′ rule and why we require the π'-uniformity side condition in our DotStar rule.

8.2 Disjointness in a multiplicative setting

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

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

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

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

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

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

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

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

$$\begin{aligned} \mathcal{F} = \mathcal{F} \otimes \mathcal{F} = (s \oplus \overline{s}) \otimes (s \oplus \overline{s}) &= (s \otimes s) \oplus (s \otimes \overline{s}) \oplus (\overline{s} \otimes s) \oplus (\overline{s} \otimes \overline{s}) \\ &= (s \otimes s) \oplus \underline{(s \otimes \overline{s}) \oplus (s \otimes \overline{s})} \oplus (\overline{s} \otimes \overline{s}) \end{aligned}$$

The final equality is by lemma 1. The underlined portion implies $s \otimes \overline{s} = \mathcal{E}$ by disjointness. The upshot of lemma 2, together with our requirement that the product of two positive shares be positive, is that we can have no more than the two elements $\mathcal{E}$ and $\mathcal{F}$ in our share model. Since the entire motivation for fractional share models is to allow ownership between $\mathcal{E}$ and $\mathcal{F}$, we must choose either left or right distributivity; we choose right since we are able to prove that the π'-uniformity side condition enables the bidirectional DotStar.

9 Related Work

Fractional permissions are essentially used to reason about resource ownership in concurrent programming. The well-known rational model $\langle [0,1], + \rangle$ by Boyland *et al.* [5] is used to reason about join-fork programs. This structure has the disjointness problem mentioned in §2, first noticed by Bornat *et al.* [4], as well as other problems discussed in §3, §4, and [2]. Boyland [6] extended the framework to scale permissions uniformly over arbitrary predicates with multiplication, *e.g.*, he defined $\pi \cdot P$ as "multiply each permission π' in P with π". However, his framework cannot fit into SL and his scaling rules are not bi-directional. Jacobs and Piessens [28] also used rationals for scaling permissions $\pi \cdot P$ in SL but only obtained one direction for DotStar and DotPlus. A different kind of scaling permission was used by Dinsdale-Young *et al.* [20] in which they used rationals to define permission assertions $[A]^r_\pi$ to indicate a thread with permission π can execute the action A over the shared region r.

There are other flavors of permission besides rationals. Bornat *et al.* [4] introduced integer counting permissions $\langle \mathbb{Z}, +, 0 \rangle$ to reason about semaphores and combined rationals and integers into a hybrid permission model. Heule *et al.* [23] flexibly allowed permissions to be either concretely rational or abstractly read-only to lower the nuisance of detailed accounting. A more general read-only permissions was proposed by Charguéraud and Pottier [13] that transforms a predicate P into read-only mode $\mathsf{RO}(P)$ which can duplicated/merged with the bi-entailment $\mathsf{RO}(P) \dashv\vdash \mathsf{RO}(P) \star \mathsf{RO}(P)$. Their permissions distribute pleasantly over disjunction and existential quantifier but only work one way for $\star$, *i.e.*, $\mathsf{RO}(H_1 \star H_2) \vdash \mathsf{RO}(H_1) \star \mathsf{RO}(H_2)$. Parkinson [41] proposed subsets of the natural numbers for shares $\langle \mathcal{P}(\mathbb{N}), \uplus \rangle$ to fix the disjointness problem. Compared to tree shares, Parkinson's model is less practical computationally and does not have an obvious multiplicative structure.

Protocol-based logics like FCSL [38] and Iris [30] have been very successful in reasoning about fine-grained concurrent programs, but their high expressivity results in a heavyweight logic. Automation (*e.g.* inference such as we do in §4) has been hard to come by. We believe that fractional permissions and protocol-based logics are in a meaningful sense complementary rather than competitors.

Verification tools often implement rational permissions because of its simplicity. For example, VeriFast [29] uses rationals to verify programs with locks and semaphores. It also allows simple and restrictive forms of scaling permissions which can be applied uniformly over standard predicates. On the other hand, HIP/SLEEK [31] uses rationals to model "thread as resource" so that the ownership of a thread and its resources can be transferred. Chalice [36] has rational permissions to verify properties of multi-threaded, objected-based programs such as data races and dead-locks. Viper [37] has an expressive intermediate language that supports both rational and abstract permissions. However, a number of verification tools have chosen tree shares due to their better metatheorical properties. VST [3] is equipped with tree share permissions and an extensive tree share library. HIP/SLEEK uses tree shares to verify the barrier structure [26] and has its own complete share solver [33, 35] that reduces tree formulae to Boolean formulae handled by Z3 [17]. Lastly, tree share permissions are featured in Heap-Hop [47] to reason over asynchronous communications.

10 Conclusion

We presented a separation logic proof framework to reason about resource sharing using fractional permissions in concurrent verification. We support sophisticated verification tasks such as inductive predicates, proving predicates precise, and biabduction. We wrote ShareInfer to gauge how our theories could be automated. We developed scaling separation algebras as compositional models for our logic. We investigated why our logic cannot support certain desirable properties.

References

1. http://www.comp.nus.edu.sg/~lxbach/tools/share_infer/.
2. http://www.comp.nus.edu.sg/~lxbach/publication/permission_full.pdf.
3. Andrew W. Appel. Verified software toolchain. In *ESOP*, 2011.
4. Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, 2005.
5. John Boyland. Checking interference with fractional permissions. In *SAS*, 2003.
6. John Tang Boyland. Semantics of fractional permissions with nesting. *ACM Trans. Program. Lang. Syst.*, 32(6), August 2010.
7. James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *POPL*, 2008.
8. James Brotherston, Nikos Gorogiannis, and Max Kanovich. Biabduction (and related problems) in array separation logic. In *CADE*, 2017.
9. Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NFM*, 2015.
10. Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.
11. Cristiano Calcagno, Dino Distefano, and Viktor Vafeiadis. Bi-abductive resource invariant synthesis. In *APLAS*, 2009.
12. Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS*, pages 366–378, 2007.
13. Arthur Charguéraud and François Pottier. Temporary read-only permissions for separation logic. In *ESOP*, 2017.
14. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, August 2012.
15. Wei Ngan Chin, Ton Chanh Le, and Shengchao Qin. Automated verification of countdownlatch, 2017.
16. Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In *ECOOP*, 2014.
17. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
18. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional reasoning for concurrent programs. In *POPL*, 2013.
19. T. Dinsdale-Young, P. D. R. Pinto, K. J. Andersen, and L. Birkedal. Caper: Automatic verification for fine-grained concurrency. In *ESOP*, 2017.
20. Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.
21. Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, 2009.
22. Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.
23. Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. Fractional permissions without the fractions. In *FTfJP*, pages 1:1–1:6, 2011.
24. Aquinas Hobor. *Oracle Semantics*. PhD thesis, Princeton University, Department of Computer Science, Princeton, NJ, October 2008.
25. Aquinas Hobor and Cristian Gherghina. Barriers in concurrent separation logic. In *ESOP*, 2011.

26. Aquinas Hobor and Cristian Gherghina. Barriers in concurrent separation logic: Now with tool support! *Logical Methods in Computer Science*, 8(2), 2012.
27. Jochen Hoenicke, Rupak Majumdar, and Andreas Podelski. Thread modularity at many levels: A pearl in compositional verification. In *POPL*, 2017.
28. Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, 2011.
29. Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In *APLAS*, 2010.
30. 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*, 2015.
31. Duy-Khanh Le, Wei-Ngan Chin, and Yong Meng Teo. Threads as resource for concurrency verification. In *PEPM*, 2015.
32. Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In *CAV*, 2014.
33. Xuan-Bach Le, Cristian Gherghina, and Aquinas Hobor. Decision procedures over sophisticated fractional permissions. In *APLAS*, 2012.
34. Xuan-Bach Le, Aquinas Hobor, and Anthony W. Lin. Decidability and complexity of tree shares formulas. In *FSTTCS*, 2016.
35. Xuan-Bach Le, Thanh Toan Nguyen, Wei Ngan Chin, and Aquinas Hobor. A certified decision procedure for tree shares. In *ICFEM*, 2017.
36. K. Rustan Leino and Peter Müller. A basis for verifying multi-threaded programs. In *ESOP*, 2009.
37. Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, 2016.
38. Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, 2014.
39. Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei Ngan Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, 2007.
40. Peter W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, April 2007.
41. Matthew Parkinson. *Local Reasoning for Java.* PhD thesis, University of Cambridge, 2005.
42. Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. Modular verification of a non-blocking stack. In *POPL*, 2007.
43. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
44. Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, 2015.
45. Viktor Vafeiadis. *Modular fine-grained concurrency verification.* PhD thesis, 2008.
46. Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.
47. Jules Villard. *Heaps and Hops.* Ph.D. thesis, Laboratoire Spécification et Vérification, École Normale Supérieure de Cachan, France, February 2011.
48. Matt Windsor, Mike Dodds, Ben Simner, and Matthew J. Parkinson. Starling: Lightweight concurrency verification with views. In *CAV*, 2017.
49. Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.