A Fresh Look at Separation Algebras and Share Accounting*

Robert Dockins[1], Aquinas Hobor[2], and Andrew W. Appel[1]

[1] Princeton University
[2] National University of Singapore

Abstract. *Separation Algebras* serve as models of Separation Logics; *Share Accounting* allows reasoning about concurrent-read/exclusive-write resources in Separation Logic. In designing a Concurrent Separation Logic and in mechanizing proofs of its soundness, we found previous axiomatizations of separation algebras and previous systems of share accounting to be useful but imperfect. We adjust the axioms of separation algebras; we demonstrate an operator calculus for constructing new separation algebras; we present a more powerful system of share accounting with a new, simple model; and we provide a reusable Coq development.

1 Introduction

Separation logic is an elegant solution to the pointer aliasing problem of Hoare logic. We have been using separation logic to examine the metatheory of C minor enhanced with primitives for shared-memory concurrency [6, 5]. Along the way, we developed a generic library of constructions and proof techniques for separation logic. Here we explain two related parts of our toolkit: separation algebras and share models. **Contribution 0:** Our proofs are machine checked in Coq.[3]

Calcagno, O'Hearn and Yang [4] present a semantics of separation logic based on structures they call "separation algebras." Although Calcagno's definition is adequate for their purposes, we found it too limiting in some ways and too permissive in others. **Contribution 1:** We make several alterations to the definition of separation algebras to produce a class of objects that have more pleasing mathematical properties and that are better suited to the task of generating useful separation logics.

Different separation logics often require different separation algebra models, but verifying that a complex object is a separation algebra can be both tedious and surprisingly difficult. **Contribution 2:** We demonstrate an operator calculus for rapidly constructing a wide variety of new separation algebras.

We also revisit share accounting, which is used to reason about read-sharing concurrent protocols. Share accounting allows a process to "own" some share of

* Supported in part by National Science Foundation grant CNS-0627650 and a Lee Kuan Yew Postdoctoral Fellowship.

[3] The Coq development corresponding to this paper is part of the Mechanized Semantic Library, available at `http://msl.cs.princeton.edu/`.

a memory location: the full share gives read/write/deallocate permissions, while a partial share gives only the read permission. **Contribution 3:** We present a new share model that is superior to those by Bornat et al. [1] and Parkinson [8].

2 Separation Algebras

Calcagno, O'Hearn, and Yang [4] introduced the notion of a *separation algebra*, which they defined as "a cancellative, partial commutive monoid." That is, a separation algebra (SA) is a tuple $\langle A, \oplus, u\rangle$ where A is a set, $\oplus$ is a partial binary operation on A and u is an element of A satisfying the following axioms:

$$x \oplus y = y \oplus x \tag{1}$$
$$x \oplus (y \oplus z) = (x \oplus y) \oplus z \tag{2}$$
$$u \oplus x = x \tag{3}$$
$$x_1 \oplus y = x_2 \oplus y \ \rightarrow \ x_1 = x_2 \tag{4}$$

The primary interest of a separation algebra is that it can be used to build a separation logic [11]. However, for most of this paper, we are going to consider separation algebras as first-class objects and investigate their properties.

We wish to construct our models in Coq. Dealing with partial functions in Coq can be tricky, since the function space of Coq's metatheory contains only computable total functions.[4] We would like to be able to construct models whose combining operation is not computable. Therefore, we adopt a convention from philosophical logic, where it is common to give the semantics of substructural connectives in terms of 3-place relations rather than binary functions [10].

We recast the separation algebra ideas of Calcagno et al. in this relational setting by considering the *join relation* $J(x, y, z)$, usually written suggestively as $x \oplus y = z$. The partiality of the $\oplus$ operation follows from the fact that for a given x and y we are not guaranteed that there is some z such that $x \oplus y = z$. Reinterpreted in this setting, we say that $\langle A, J\rangle$, (where A is a carrier set and J is a three-place relation on A) is a separation algebra provided that:

$$x \oplus y = z_1 \ \rightarrow \ x \oplus y = z_2 \ \rightarrow \ z_1 = z_2 \tag{5}$$
$$x_1 \oplus y = z \ \rightarrow \ x_2 \oplus y = z \ \rightarrow \ x_1 = x_2 \tag{6}$$
$$x \oplus y = z \ \rightarrow \ y \oplus x = z \tag{7}$$
$$x \oplus y = a \ \rightarrow \ a \oplus z = b \ \rightarrow \ \exists c.\ y \oplus z = c \wedge x \oplus c = b \tag{8}$$
$$\exists u.\ \forall x.\ u \oplus x = x \tag{9}$$

That is, $\oplus$ is a functional relation (5), it is cancellative (6), commutative (7), associative (8), and has a unit u (9).

We are justified in calling this object "the" unit because the cancellation axiom guarantees that it must be unique. In addition, the unit is exactly the unique element u satisfying $u \oplus u = u$, again by the cancellation axiom.

[4] Uncomputable functions can be built if one assumes the axiom of description.

3 The algebra of separation algebras

A new separation algebra can be built by applying operators to preexisting SAs.

Definition 1 (SA product operator). *Let $\langle A, J_A\rangle$ and $\langle B, J_B\rangle$ be SAs. Then the product SA is $\langle A \times B, J\rangle$, where J is defined componentwise:*

$$J((x_a, x_b), (y_a, y_b), (z_a, z_b)) \equiv J_A(x_a, y_a, z_a) \wedge J_B(x_b, y_b, z_b) \tag{10}$$

Definition 2 (SA function operator). *Let A be a set and let $\langle B, J_B\rangle$ be a SA. Then the function SA is $\langle A \to B, J\rangle$, where J is defined pointwise:*

$$J(f, g, h) \equiv \forall a \in A.\ J_B(f(a), g(a), h(a)) \tag{11}$$

The SA product and the SA function operators are isomorphic to special cases of the general indexed product (*i.e.*, dependent function space) operator.

Definition 3 (SA indexed product operator). *Let I be a set, called the index set, and let P be a mapping from I to separation algebras. Then the indexed product SA is $\langle \Pi x : I.\ P(x), J\rangle$ where J is defined pointwise:*

$$J(f, g, h) \equiv \forall i \in I.\ J_{P(i)}(f(i), g(i), h(i)) \tag{12}$$

What about the disjoint union operator? Can it be constructed? Unfortunately, it cannot. Suppose we have two SAs $\langle A, J_A\rangle$, $\langle B, J_B\rangle$. We would like to define $\langle A + B, J\rangle$ such that J is the smallest relation satisfying:

$$J_A(x, y, z) \to J(inl\ x, inl\ y, inl\ z) \quad \text{for all } x, y, z \in A \tag{13}$$

$$J_B(x, y, z) \to J(inr\ x, inr\ y, inr\ z) \quad \text{for all } x, y, z \in B \tag{14}$$

Here $A + B$ is the disjoint union of A and B with *inl* and *inr* as the left and right injections. This structure cannot be a separation algebra under the original axioms: if u_A and u_B are the units of A and B, then both $inl\ u_A$ and $inr\ u_B$ satisfy $u \oplus u = u$. We noted the unit u is the unique element satisfying this equation, so $inl\ u_A = u = inr\ u_B$. However, this is a contradiction as the injection functions for disjoint union always produce unequal elements. The generalization of disjoint union to the indexed sum fails for the same reason.

One option for dealing with this problem is to use the "almost" disjoint sum, which is the disjoint sum of the nonunit elements together with a single unit. This solution is adequate, but not entirely satisfactory; it combines two distinct operations (the disjoint sum and the combining of units) which are better understood separately. In other words, the construction is not *compositional*.

There is good news, however: we can slightly relax the SA axioms to allow the desired construction. Recall the SA axiom for the existence of a unit:

$$\exists u.\ \forall x.\ u \oplus x = x \tag{9}$$

If we simply swap the order of the quantifiers in this axiom, we get a weaker statement which says that every element of the SA has an associated unit:

$$\forall x.\ \exists u_x.\ u_x \oplus x = x \tag{15}$$

To distinguish these species of separation algebras, we call SAs defined by axioms 5, 6, 7, 8, and 9 *Single-unit Separation Algebras* (SSA), we call SAs defined by axioms 5, 6, 7, 8, and 15 *Multi-unit Separation Algebras* (MSA). Note that axiom 9 implies axiom 15, and thus the SSAs are a strict subset of the MSAs.

Although the relaxation to multiple units is common in the tradition of relevant logic [10], it is significant here because it enables a number of additional operators, including naïve indexed sums and the *discrete* separation algebra.

Definition 4 (MSA indexed sum operator). *Let I be a set, called the index set. Let S be a mapping from I to separation algebras. Then the indexed sum MSA is $\langle \Sigma i : I.\ S(i), J\rangle$ such that J is the least relation satisfying:*

$$J_{S(i)}(x,y,z) \to J(inj_i(x), inj_i(y), inj_i(z)) \quad \text{for all } i \in I; x,y,z \in S(i) \tag{16}$$

Here inj_i is the injection function associated with i. As with products, if $|I| = 2$, the indexed sum is isomorphic to the disjoint union operator.

Definition 5 (discrete MSA). *Let A be a set. Then the discrete MSA is $\langle A, J\rangle$ where J is defined as the smallest relation satisfying:*

$$J(x,x,x) \quad \text{for all } x \in A \tag{17}$$

The discrete MSA has a join relation that holds only when all three arguments are equal: *every* element of the discrete MSA is a unit. The discrete MSA is useful for constructing MSAs over tuples where only some of the components have interesting joins; the other components can be turned into discrete MSAs. Note the *compositionality* of this construction using the discrete and product operators together. With SSAs, one would instead have to "manually" construct an appropriate tupling operator because the discrete SA is not available.

Sometimes we wish to coerce an MSA into an SSA; the *lifting operator* removes all the units from an MSA and replaces them with a new unique unit.[5]

Definition 6 (MSA lifting operator). *Let $\langle A, J_A\rangle$ be a multi-unit separation algebra. Define A^+ to be the subset of A containing all the nonunit elements, $A^+ = \{x \in A | \neg(x \oplus x = x)\}$. Let $\bot$ be a distinguished element such that $\bot \notin A$. Then the lifting SA is $\langle A^+ \cup \{\bot\}, J\rangle$ where J is the least relation satisfying:*

$$J(\bot, x, x) \quad \text{for all } x \in A^+ \cup \{\bot\} \tag{18}$$

$$J(x, \bot, x) \quad \text{for all } x \in A^+ \cup \{\bot\} \tag{19}$$

$$J_A(x,y,z) \to J(x,y,z) \quad \text{for all } x,y,z \in A^+ \tag{20}$$

Note that the "almost" disjoint union described above can be constructed using the lifting operator applied to the naïve disjoint sum operator on MSAs. Again compositionality is increased by working with MSAs.

The above operators can construct many kinds of separation algebras. The associated Coq development includes a number of additional operators (*e.g.*, lists, subsets, bijections, etc.) that we have also found useful.

[5] Considered as a functor from **MSA** to **SSA**, the lifting operator is left adjoint to the inclusion functor from **SSA** to **MSA**.

Assume we have a MSA $\langle A, J \rangle$. The following is a model of HBI (a Hilbert-style axiomatization of the logic of bunched implications). `Prop` is the type of Coq propositions, and the right-hand sides are stated in Coq's metalogic.

$$
\begin{array}{lcl}
\mathit{formula} & \equiv & A \to \texttt{Prop} \\
a \models p & \equiv & p(a) \\
p \vdash q & \equiv & \forall a.\ a \models p \to a \models q
\end{array}
$$

$$
\begin{array}{lcl}
\top & \equiv & \lambda a.\ \texttt{True} \\
\bot & \equiv & \lambda a.\ \texttt{False} \\
p \wedge q & \equiv & \lambda a.\ a \models p \wedge a \models q \\
p \vee q & \equiv & \lambda a.\ a \models p \vee a \models q \\
p \to q & \equiv & \lambda a.\ a \models p \to a \models q \\
\mathbf{emp} & \equiv & \lambda a.\ a \oplus a = a \\
p * q & \equiv & \lambda a.\ \exists a_1 . \exists a_2.\ a_1 \oplus a_2 = a \wedge a_1 \models p \wedge a_2 \models q \\
p \mathrel{-\!\!*} q & \equiv & \lambda a.\ \forall a_1 . \forall a'.\ a_1 \oplus a = a' \to a_1 \models p \to a' \models q
\end{array}
$$

Table 1. A model of HBI given a SA

4 Inducing a separation logic

The purpose of a separation algebra is to generate a separation logic, that is, a Hoare-style program logic where the assertion language is the logic of bunched implications (BI). Calcagno et al. demonstrated that their interpretation leads to a Boolean BI algebra, a model of BI. Here we demonstrate that we are still generating models for the desired class of logics despite relaxing the unit axiom.

We too will interpret formulae of separation logic as predicates on the elements of a separation algebra (equivalently, members of the powerset). In Coq we simply define the formulae as `A -> Prop`, where `A` is the type of elements in the MSA, and `Prop` is the type of propositions in Coq's metatheory.

We have chosen to directly link our models to the proof theory of BI by showing[6] a soundness proof with respect to the system HBI, a Hilbert-style axiomatic system for the (propositional) logic of bunched implications [9, Table 2]. The definitions which give rise to HBI are summarized in table 1; they are quite standard, except for the definition of the empty proposition **emp**.

Ordinarily, one defines **emp** as the predicate which accepts only the unit of the SSA. However, by relaxing the unit axiom we allow multiple units, each of which must be characterized. Recall that an MSA is a set of equivalence classes distinguished by unique units, each of which satisfies the equation $u \oplus u = u$. In fact, only units satisfy that equation, so we define **emp** as the predicate that accepts any element x provided that $x \oplus x = x$. This subsumes the ordinary definition in the event that the unit is unique. More importantly, however, from this definition we can prove that **emp** is the unit for separating conjunction.

Thus we see that relaxing the unit axiom does not take us outside the class of models of the logic of bunched implications.

[6] The accompanying Coq development contains the full set of definitions and proofs.

5 Useful restrictions of SAs

Positivity. Calcagno et al.'s definition of separation algebras [4] permits very strange logics that do not correspond well to the common view that the formulae in separation logic describe resources.

Consider the structure $\langle \{0, 1, 2\}, +_3 \rangle$, of the integers with addition modulo 3. This structure satisfies the separation algebra axioms given in the previous section and the integer 0 is the unique unit. The problem is that the following holds: $1 +_3 2 = 0$. The resource 1 combines with the resource 2 to give the "empty" resource 0. Stated another way, we can *split* the empty resource 0 to get two *nonempty* resources 1 and 2. By analogy to physics, 1 and 2 act as a resource/antiresource pair that annihilate each other when combined.

This is not at all how one expects resources to behave. If one has, for example, an empty pile of bricks and splits it into two piles, one expects to have two empty piles. One does *not* expect to get one pile of bricks and another pile of antibricks.

The existence of antiresources is particularly troublesome because it interacts badly with the frame rule, a ubiquitous feature of separation logic. A program with no resources can write to memory! Proof: it splits the empty permission, obtaining a write and an anti-write permission. Using the frame rule, it "frames out" the antipermission, giving it the permission to perform a write.

Calcagno et al. resolve this problem by requiring that all actions be "local." The locality condition captures the requirement that actions must be compatible with the frame rule; as a side-effect, the locality condition ensures that any "negative" resources that may exist cannot be used for any interesting purpose. The ultimate consequence is that proving the required locality properties is more difficult (if not impossible) in SAs with negative resources.

We prefer to directly rule out this troublesome class of separation algebras by disallowing negative resources. We require that SAs be *positive* by adding the following positivity axiom:

$$a \oplus b = c \;\rightarrow\; c \oplus c = c \;\rightarrow\; a \oplus a = a \tag{21}$$

That is, whenever two elements join to create a unit element, these joined elements must themselves be units (and hence the same element). This axiom rules out separation algebras such as the addition-modulo-3 example above. Furthermore, this axiom is preserved by the all the SA operators we examined above, which means we have not lost the ground we gained by relaxing the unit axiom.

One of most compelling reasons for including (21) in the axiom base is that all the nontoy SAs known to the authors (that is, those which can be used to reason about some computational system), including all five examples listed by Calcagno et al. [4], satisfy this axiom.[7]

[7] Brotherson and Calcagno [3] investigate *Classical* BI, whose models have *negative*, or *dual*, elements. However, the combination of an element with its dual gives the distinguished element ∞, which is not necessarily the unit, so (21) may still hold.

The positivity axiom also allows us to make a connection to order theory. We define an ordering relation on the elements of a separation algebra:

$$a \preceq b \quad \equiv \quad \exists x.\ a \oplus x = b \tag{22}$$

This relation is reflexive and transitive, which makes it a preorder. The preorder $\preceq$ is antisymmetric, and thus a partial order, if and only if axiom (21) holds.

Disjointness. The *disjointness* property is that no nonempty share joins with itself. If a separation logic over program heaps lacks disjointness then unusual things can happen when defining predicates about inductive data in a program heap. For example, without disjointness, the "obvious" definition of a formula to describe binary trees in fact describes directed acyclic graphs [1].

Disjointness is easy to axiomatize:

$$a \oplus a = b \ \rightarrow \ a = b \tag{23}$$

The disjointness axiom requires that any SA element that joins with itself be a unit. Equivalently, it says that any nonunit element cannot join with itself (is disjoint). This axiom captures the same idea as Parkinson's "disjoint" axiom [8]:

$$\ell \mapsto_s v * \ell \mapsto_s v \ \leftrightarrow \ \text{false} \tag{24}$$

The primary difference is that our axiom is on separation *algebras*, whereas Parkinson's axiom is on separation *logic*.

As with the positivity axiom, the disjointness axiom is preserved by the SA operators presented in the previous section. It also implies positivity. The converse does not hold: disjointness is strictly stronger than positivity.

To see why disjointness (23) implies positivity (21), consider the following proof sketch. Assume $a \oplus b = c$ and $c \oplus c = c$ for some a, b, and c; we wish to show $a \oplus a = a$. Then the following hold (modulo some abuse of notation):

$c \oplus c$	$= c$	assumed
$(a \oplus b) \oplus (a \oplus b)$	$= a \oplus b$	subst. $a \oplus b = c$
$(a \oplus a) \oplus (b \oplus b)$	$= a \oplus b$	comm. and assoc.
$(a \oplus a) \oplus b$	$= a \oplus b$	disjointness
$a \oplus a$	$= a$	cancellation

The converse fails: consider the structure $\langle \mathbb{N}, + \rangle$ of natural numbers with addition. This fulfills the SA axioms, including positivity. However, every natural number $i > 0$ falsifies the disjointness axiom.

Two alternative ways of formulating the disjointness property are inspired by order theory and provide additional insight. First, if $a \oplus b = c$, then any lower bound of a and b is a unit:

$$a \oplus b = c \ \rightarrow \ d \preceq a \ \rightarrow \ d \preceq b \ \rightarrow \ d \oplus d = d \tag{25}$$

Second, $a \oplus b$ is minimal in the following sense:

$$a \oplus b = c \ \rightarrow \ a \preceq d \ \rightarrow \ b \preceq d \ \rightarrow \ d \preceq c \ \rightarrow \ c = d \tag{26}$$

This implies that if a and b join and have a least upper bound, then it is $a \oplus b$.

Cross-split. The alternative formulations of disjointness bring up an interesting point about separation algebras: even for two elements in the same equivalence class, there is no guarantee that either least upper bounds or greatest lower bounds exist. The lack of greatest lower bounds (*i.e.*, intersections), in particular, proved to be troublesome in Hobor et al.'s proof of soundness of a concurrent separation logic for Concurrent C minor [6, 5], when they needed to track permissions being transferred between threads. At the time they were using a modified version of Parkinson's share model (discussed below) in which intersections did not always exist. This failure resulted in an unpleasant workaround and spurred development of the alternate model discussed in section 7.

The particular property required was as follows: suppose a single resource can be split in two different ways; then one should be able to divide the original resource into *four* pieces that respect the original splittings.

$$\begin{array}{l} a \oplus b = z \ \wedge \ c \oplus d = z \rightarrow \\ \exists\ ac,\ ad,\ bc,\ bd. \\ \quad ac \oplus ad = a \ \wedge \ bc \oplus bd = b \ \wedge \\ \quad ac \oplus bc = c \ \wedge \ ad \oplus bd = d \end{array} \qquad \forall \begin{array}{|c|c|}\hline a & b \\ \hline \end{array} \quad \begin{array}{|c|}\hline c \\ \hline d \\ \hline \end{array} \quad \exists \begin{array}{|c|c|}\hline ac & bc \\ \hline ad & bd \\ \hline \end{array} \tag{27}$$

That is, if an element can be split in two different ways, then there should be four subelements that partition the original element and respect the splittings. We call this property the cross-split axiom and, as with positivity and disjointness, this property is preserved by the separation algebra operators.[8]

Splittability. Another frequently desirable property of SAs is *infinite splittability*, which is a useful property for reasoning about the kinds of resource sharing that occur in divide-and-conquer style computations. Splittability means that we can take any element of the SA and split it into two pieces that recombine into the original. To avoid degenerate splittings, both the split pieces must be nonempty if the original was nonempty. Thus, a SA is infinitely splittable if there exists a function `split` that calculates such a splitting.

$$\texttt{split}\ x = (x_1, x_2) \ \rightarrow \ x_1 \oplus x_2 = x \tag{28}$$

$$\texttt{split}\ x = (x_1, x_2) \ \rightarrow \ x_1 \oplus x_1 = x_1 \rightarrow x \oplus x = x \tag{29}$$

$$\texttt{split}\ x = (x_1, x_2) \ \rightarrow \ x_2 \oplus x_2 = x_2 \rightarrow x \oplus x = x \tag{30}$$

There are reasonable separation logics that have models where disjointness, cross-split, and/or infinite splittability are false. We present these axioms because we (and others such as Parkinson) found them useful in separation logic proofs and metaproofs, and because any separation algebra built with our operator calculus can inherit them for free if desired. In contrast, every nontoy separation algebra used for reasoning about computational systems and known to the authors satisfies the positivity axiom (21). In our own work we find that adding the disjointness axiom (which implies positivity) to the SA axiom base to be quite convenient and suggest that "disjoint" MSAs, defined by axioms 5, 6, 7, 8, 15 and 23, are very natural structures for reasoning in this domain.

[8] To pull an intersection property through the lifting operator, one needs to apply lifts only where there is a computable test for units, or use the axiom of description.

6 Shares

An important application of separation algebras is to model Hoare logics of programming languages with mutable memory. We generate an appropriate separation logic by choosing the correct semantic model, that is, the correct separation algebra. A natural choice is to simply take the program heaps as the elements of the separation algebra together with some appropriate join relation.

In most of the early work in this direction, heaps were modeled as partial functions from addresses to values. In those models, two heaps join iff their domains are disjoint, the result being the union of the two heaps. However, this simple model is too restrictive, especially when one considers concurrency. It rules out useful and interesting protocols where two or more threads agree to share *read* permission to an area of memory.

There are a number of different ways to do the necessary permission accounting. Bornat et al. [1] present two different methods; one based on fractional permissions, and another based on token counting. Parkinson, in chapter 5 of his thesis [8], presents a more sophisticated system capable of handling both methods. However, this model has some drawbacks, which we shall address below.

Fractional permissions are used to handle the sorts of accounting situations that arise from concurrent divide-and-conquer algorithms. In such algorithms, a worker thread has read-only permission to the dataset and it needs to divide this permission among various child threads. When a child thread finishes, it returns its permission to its parent. Child threads, in turn, may need to split their permissions among their own children and so on. In order to handle any possible pattern of divide-and-conquer, splitting must be possible to an unbounded depth.

The token-counting method is intended to handle the accounting problem that arises from reader-writer locks. When a reader acquires a lock, it receives a "share token," which it will later return when it unlocks. The lock tracks the number of active readers with an integer counter that is incremented when a reader locks and decremented when a reader unlocks. When the reader count is positive there are outstanding read tokens; when it is zero there are no outstanding readers and a writer may acquire the lock.

Here we will show how each of the above accounting systems arises from the choice of a "share model," and we present our own share model which can handle both accounting methods and avoids a pitfall found in Parkinson's model.

Suppose we have a separation algebra $\langle S, J_S \rangle$ of *shares*. If L and V are sets of addresses and values, respectively, we can define a SA over heaps as follows:

$$H \quad \equiv \quad L \;\rightarrow\; (S \times V_{=})_{\perp} \tag{31}$$

This equation is quite concise but conceals some subtle points. The operators in this equation are the operators on SAs defined in § 3. We let $V_{=}$ be the "discrete" MSA over values (i.e., values V with the trivial join relation) and $S \times V_{=}$ is the MSA over pairs of shares and values.Next we construct the "lifted" SSA $(S \times V_{=})_{\perp}$, which removes the unit values and adds a new, distinguished unit $\perp$. This requires values to be paired only with nonunit shares. Finally,

$L \to (S \times V_=)_\perp$ builds the function space SSA. Thus, heaps are partial functions from locations to pairs of nonunit shares and values.[9]

Now we can define the points-to operator of separation logic as:

$$\ell \mapsto_s v \quad \equiv \quad \lambda h.\ h(\ell) = (s, v) \wedge (\forall \ell'.\ell \neq \ell' \to h(\ell') = \perp) \tag{32}$$

Here, $\ell \in L$ is an address, $v \in V$ is a value, and $s \in S^+$ is a nonunit share. In English, $\ell \mapsto_s v$ means "the memory location at address ℓ contains v, I have share s at this location, and I have no permission at any other locations." Now the exact behavior of the points-to operator depends only on the share model S.

An important property of this definition is that the separation algebra on shares lifts in a straightforward way through the separation logic:

$$s_1 \oplus s_2 = s \ \leftrightarrow \ (\ell \mapsto_s v \ \leftrightarrow \ \ell \mapsto_{s_1} v * \ell \mapsto_{s_2} v) \tag{33}$$

Thus we can use properties of our share model in the separation logic.

We can produce a separation logic very similar to the ones studied by Reynolds [11] and by Ishtiaq and O'Hearn [7] by choosing S to be the SA over Booleans with the smallest join relation such that "false" is the unique unit.

Definition 7 (Boolean shares). *The Boolean share model is $\langle \{\circ, \bullet\}, J \rangle$ where J is the least relation satisfying $J(\circ, x, x)$ and $J(x, \circ, x)$ for all $x \in \{\circ, \bullet\}$.*

Here $\circ$ and $\bullet$ stand for "false" and "true", respectively. This share model is unsophisticated: one either has unrestricted permission or no permission at all. Note that the lifting operator removes $\circ$, leaving $\bullet$ as the only legal annotation. This justifies omitting the annotation, resulting in the more familiar $\ell \mapsto v$.

Boyland proposed a model which takes shares as fractions in the interval $[0, 1]$ as shares [2]. Although Boyland works in the reals, the rationals suffice.

Definition 8 (Fractional shares). *The fractional share model is $\langle [0, 1] \cap \mathbb{Q}, + \rangle$ where $+$ is the restriction of addition to a partial operation on $[0, 1]$.*

The main advantage of the fractional share model is that it is infinitely splittable. The splitting function is simple: to split a share s, let $s_1 = s_2 = s/2$. The fractional share model satisfies the positivity axiom but not the disjointness axiom, which leads to the problems noticed by Bornat et al. [1, §13.1].

Bornat et al. also examined the *token factory* model, where a central authority starts with total ownership and then lends out permission tokens. The authority counts the outstanding tokens; when the count is zero, all have returned. A slight modification of Bornat's construction yields a suitable model:

Definition 9 (Counting shares). *The counting share model is $\langle \mathbb{Z} \cup \{\perp\}, J \rangle$ where J is defined as the least relation satisfying:*

$$J(\perp, x, x) \quad \text{for all } x \in \mathbb{Z} \cup \{\perp\} \tag{34}$$

$$J(x, \perp, x) \quad \text{for all } x \in \mathbb{Z} \cup \{\perp\} \tag{35}$$

$$(x < 0 \vee y < 0) \wedge ((x + y \geq 0) \vee (x < 0 \wedge y < 0)) \to J(x, y, x + y) \quad \text{for all } x, y \in \mathbb{Z} \tag{36}$$

[9] Our heaps are quite similar those defined by Bornat et al. [1, §10.1]. Their "partial commutative semigroup" of shares arises here from the nonunit elements of a SA.

This definition sets up the nonnegative integers as token factories and negative integers as tokens. To absorb a token back into a factory, the integers are simply added. The token factory has collected all its tokens when its share is zero. Like the fractional model, the counting model satisfies positivity but not disjointness.

This share model validates the following logical axioms:

$$\ell \mapsto_n v \quad \leftrightarrow \quad (\ell \mapsto_{n+m} v * \ell \mapsto_{-m} v) \quad \text{for } n \geq 0 \text{ and } m > 0 \tag{37}$$

$$\ell \mapsto_{-(n+m)} v \quad \leftrightarrow \quad (\ell \mapsto_{-n} v * \ell \mapsto_{-m} v) \quad \text{for } n, m > 0 \tag{38}$$

$$(\ell \mapsto_0 v * \ell \mapsto_n v) \quad \leftrightarrow \quad \text{false} \tag{39}$$

Equation (37) says that a token factory with n tokens outstanding can be split into a token (of size m) and a new factory, which has $n + m$ tokens outstanding. Furthermore the operation is reversable: a token and its factory can be recombined to get a factory with fewer outstanding tokens. Equation (38) says that the tokens themselves may be split and merged. Finally, equation (39) says that it is impossible to have both a full token factory (with no outstanding tokens) and any other share of the same location (whether a factory or a token).

If one only utilizes tokens of size one, then equations (37)–(39) describe the sorts of share manipulations required for a standard reader-writer lock. Other token sizes allow more subtle locking protocols where, for example, one thread may acquire the read tokens of several others and release them all at once.

In his thesis, Parkinson defines a more sophisticated share model that can support both the splitting and the token counting use cases.

Definition 10 (Parkinson's named shares). *Parkinson's named share model is given by* $\langle \mathcal{P}(\mathbb{N}), \uplus \rangle$*, where* $\mathcal{P}(\mathbb{N})$ *is the set of subsets of the natural numbers and* $\uplus$ *is disjoint union.*[10]

This model satisfies the disjointness axiom, and thus positivity. It also satisfies the cross-split axiom: the required subshares are calculated by set intersection.

In order to support the token-counting use case, Parkinson considers the finite and cofinite subsets of $\mathbb{N}$. These sets can be related to the counting model given above by considering the cardinality of the set (or set complement, for cofinite sets). We will see the details of this embedding later.

Unfortunately, this share model is not infinitely splittable, since there is no way to split a singleton set into two nonempty subsets. Therefore we cannot define a total function which calculates the splitting of a share in this model, and this makes it difficult to support the parallel divide-and-conquer use case.

We can fix this problem by restricting the model to include only the *infinite* subsets of $\mathbb{N}$ (and the empty set). We can split an infinite set s by enumerating its elements and generating s_1 from those in even positions and s_2 from the those in odd positions. Then s_1 and s_2 are infinite, disjoint, and partition s.

Unfortunately, restricting to infinite subsets means that we cannot use finite and cofinite sets to model token counting. This problem can be solved, at the cost of some complication, with an embedding into the infinite sets [8].

[10] That is, the union of disjoint sets rather than discriminated union.

The problem with *that* solution is that the infinite subsets of $\mathbb{N}$ are also not closed under set intersection, which means the share model no longer satisfies the cross split axiom. To see why this axiom fails, consider splitting $\mathbb{N}$ into the primes/nonprimes and the even/odd numbers. All four sets are infinite, but the set $\{2\}$ of even primes is finite and thus not in the share model.

Hobor suggested further restricting the model by reasoning about equivalence classes of subsets of $\mathbb{N}$, where two subsets are equivalent when their symmetric difference is finite; but developing this model in Coq was difficult [5].

We will present a new model with all the right properties: disjointness axiom, cross-split axiom, infinitely splittable, supports token counting, and is straightforward to represent in a theorem prover. As a bonus, we also achieve a decidable test for share equality.

7 Binary tree share model

Before giving the explicit construction of our share model, we shall take a short detour to show how we can induce a separation algebra from a lattice.

Definition 11 (Lattice SA). *Let $\langle A, \sqsubseteq, \sqcap, \sqcup, 0, 1\rangle$ be a bounded distributive lattice. Then, $\langle A, J\rangle$ is a separation algebra where J is defined as:*

$$J(x, y, z) \quad \equiv \quad x \sqcup y = z \ \wedge \ x \sqcap y = 0 \tag{40}$$

Disjointness follows from the right conjunct of the join relation; cross split follows from the existence of greatest lower bounds. It also has a unique unit, 0.

It is interesting to note that all of the share models we have examined thus far that satisfy the disjointness axiom are instances of this general construction.[11] The Boolean share model is just the lattice SA derived from the canonical 2-element Boolean algebra, and Parkinson's model (without the restriction to infinite subsets) is the separation algebra derived from the powerset Boolean algebra. Restricting Parkinson's model to infinite sets as described above buys the ability to do infinite splitting at the price of destroying part of the structure of the lattice. Below we show that paying this price is unnecessary.

If the structure is additionally a Boolean algebra, then we can make the following pleasant connection:

$$x \preceq y \ \leftrightarrow \ x \sqsubseteq y \tag{41}$$

That is, the lattice order coincides with the SA order. The forward direction holds for any bounded distributive lattice. The backward direction relies on the complement operator to construct the witness $(\neg x \sqcap y)$ for the existential quantifier in the definition of $\preceq$. Any bounded distributive lattice satisfying (41) is a Boolean algebra; the witness of $\preceq$ gives the complement for x when $y = 1$.

[11] This is not *necessarily* so. There exist disjoint SAs which are not distributive lattices.

Trees. Now we can restate our goal; we wish to construct a bounded distributive lattice which supports splitting and token counting. This means we must support a splitting function and we must be able to embed the finite and cofinite subsets of the naturals. We can build a model of shares supporting all these operations by starting with a very simple data structure: the humble binary tree. We consider binary trees with Boolean-valued leaves and unlabeled internal nodes.

$$\tau ::= \circ \mid \bullet \mid \tau\,\widehat{}\,\tau \tag{42}$$

We use an empty circle $\circ$ to represent a "false" leaf and the filled circle $\bullet$ to represent a "true" leaf. Thus $\bullet$ is a tree with a single leaf, $\circ\,\widehat{}\,\bullet$ is a tree with one internal node and two leaves, etc.

We define the ordering on trees as the least relation $\sqsubseteq$ satisfying:

$$\circ \sqsubseteq \circ \tag{43}$$

$$\circ \sqsubseteq \bullet \tag{44}$$

$$\bullet \sqsubseteq \bullet \tag{45}$$

$$\circ \cong \circ\,\widehat{}\,\circ \tag{46}$$

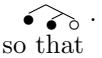
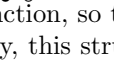
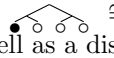
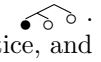
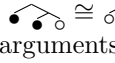
$$\bullet \cong \bullet\,\widehat{}\,\bullet \tag{47}$$

$$x_1 \sqsubseteq x_2 \quad \rightarrow \quad y_1 \sqsubseteq y_2 \quad \rightarrow \quad x_1\,\widehat{}\,y_1 \sqsubseteq x_2\,\widehat{}\,y_2 \tag{48}$$

Here, $x \cong y$ is defined as $x \sqsubseteq y \wedge y \sqsubseteq x$. The intuitive meaning is that $x \sqsubseteq y$ holds iff x has a $\circ$ in at least every position y does once we expand leaf nodes using the congruence rules until the trees are the same shape. The congruence rules allow us to "fold up" any subtree which has the same label on all its leaves.

This relation is reflexive and transitive; however it is not antisymmetric because of the structural congruence rules. We can get around this by working only with the "canonical" trees. A tree is canonical if it is the tree with the fewest nodes in the equivalence class generated by $\cong$. Canonical trees always exist and are unique, and the ordering relation is antisymmetric on the domain of canonical trees. Therefore we can build a partial order using the canonical Boolean-labeled binary trees with the above ordering relation.

The details of canonicalization are straightforward but tedious, so we will work informally up to congruence. In the formal Coq development, however, we give a full account of canonicalization and show all the required properties. The short story is that we normalize trees after every operation by finding and reducing all the subtrees which can be reduced by one of the congruence rules.

Our next task is to implement the lattice operations. The trees $\circ$ and $\bullet$ are the least and greatest element of the partial order, respectively. The least upper bound of two trees is calculated as the pointwise disjunction of Booleans (expanding the trees as necessary to make them the same shape). For example,  $\sqcup$  $\cong$  $\sqcup$  $\cong$  $\cong$  . Likewise, the greatest lower bound is found by pointwise conjunction, so that  $\sqcap$  $\cong$  $\sqcap$  $\cong$  $\cong$  . Finally, this structure is a Boolean algebra as well as a distributive lattice, and the complement operation is pointwise Boolean complement: $\neg$  $\cong$  . The Boolean algebra axioms can be verified by simple inductive arguments over the structure of the trees.

We can also define a decidable test for equality by simply checking structural equality of trees. Trees form a lattice, and thus a decision procedure for equality also yields a test for the lattice order. In contrast, Parkinson's model over arbitrary subsets of $\mathbb{N}$ lacks both decidable equality and decidable ordering.

In addition to the lattice operations, we require an operation to split trees. Given some tree s, we wish to find two trees s_1 and s_2 such that $s_1 \sqcup s_2 \cong s$ and $s_1 \sqcap s_2 \cong \circ$ and both $s_1 \ncong \circ$ and $s_2 \ncong \circ$ provided that $s \ncong \circ$. We can calculate s_1 and s_2 by recursively replacing each $\bullet$ leaf in s with $\overset{\frown}{\bullet}\circ$ and $\circ\overset{\frown}{\bullet}$ respectively.

We can usefully generalize this procedure by defining the "relativization" operator $x \bowtie y$, which replaces every $\bullet$ leaf in x with the tree y. This operator is associative with identity $\bullet$. It distributes over $\sqcup$ and $\sqcap$ on the left, and is injective for non-$\circ$ arguments.

$$x \bowtie \bullet = x = \bullet \bowtie x \tag{49}$$

$$x \bowtie \circ = \circ = \circ \bowtie x \tag{50}$$

$$x \bowtie (y \bowtie z) = (x \bowtie y) \bowtie z \tag{51}$$

$$x \bowtie (y \sqcup z) = (x \bowtie y) \sqcup (x \bowtie z) \tag{52}$$

$$x \bowtie (y \sqcap z) = (x \bowtie y) \sqcap (x \bowtie z) \tag{53}$$

$$x \bowtie y_1 = x \bowtie y_2 \;\rightarrow\; x = \circ \vee y_1 = y_2 \tag{54}$$

$$x_1 \bowtie y = x_2 \bowtie y \;\rightarrow\; x_1 = x_2 \vee y = \circ \tag{55}$$

Given this operator, we can more succinctly define the split of x as returning the pair containing $x \bowtie \overset{\frown}{\bullet}\circ$ and $x \bowtie \circ\overset{\frown}{\bullet}$. The required splitting properties follow easily from this definition and the above properties of $\bowtie$.

If this were the only use of the relativization, however, it would hardly be worthwhile to define it. Instead, the main purpose of this operator is to allow us to glue together arbitrary methods for partitioning permissions. In particular, we can split or perform token counting on any nonempty permission we obtain, no matter how it was originally generated. In addition, we only have to concentrate on how to perform accounting of the full permission $\bullet$ because we can let the $\bowtie$ operator handle relativizing to some other permission of interest.

Following Parkinson, we will consider finite and cofinite sets of the natural numbers to support token counting. This structure has several nice properties. First, it is closed under set intersection, set union and set complement and it contains $\mathbb{N}$ and $\emptyset$; in other words, it forms a sub-Boolean algebra of the power-set Boolean algebra over $\mathbb{N}$. Furthermore the cardinalities of these sets can be mapped to the integers in following way:

$$[\![p]\!]_{\mathbb{Z}} = \begin{cases} -|p| & \text{when } p \text{ is finite and nonempty} \\ |\mathbb{N} \backslash p| & \text{when } p \text{ is cofinite} \end{cases} \tag{56}$$

The cardinalities of disjoint (co)finite sets combine in exactly the way defined by the counting share model (equation 36).

We can embed the (co)finite subsets of $\mathbb{N}$ into our binary tree model by encoding the sets as right-biased trees[12] (trees where the left subtree of each

[12] We could just as well have used left-biased trees.

internal node is always a leaf). Such trees form a list of Booleans together with one extra Boolean, the rightmost leaf in the tree. Then the ith Boolean in the list encodes whether the natural number i is in the set. The final terminating Boolean stands for all the remaining naturals. If it is $\circ$, the set is finite and does not contain the remaining naturals, and if it is $\bullet$ the set is infinite and contains all the remaining naturals. This interpretation is consistent with the congruence rules that allow you to unfold the rightmost terminating Boolean into a arbitrarily long list of the same Boolean value.

For example, the finite set $\{0, 2\}$ is encoded in tree form as  . The coset $\mathbb{N}\backslash\{0, 2\}$ is encoded as  . And, of course,  $\oplus$  $=$ $\bullet$.

This encoding is in fact a Boolean algebra homomorphism; GLBs, LUBs, complements and the top and bottom elements are preserved. This homomorphism allows us to transport the token counting results on (co)finite sets to binary trees. We write $[\![p]\!]_\tau = s$ when s is the tree encoding the (co)finite set p.

Now we can define a more sophisticated points-to operator which allows us to incorporate token counting along with permission splitting.

$$\ell \mapsto_{s,n} v \equiv \lambda h.\, \exists p.\, h(\ell) = (s \bowtie [\![p]\!]_\tau, v) \wedge [\![p]\!]_{\mathbb{Z}} = n \wedge \forall \ell'.\ell \neq \ell' \rightarrow h(\ell') = \bot \quad (57)$$

Then $\ell \mapsto_{s,n} v$ means that ℓ contains value v and we have a portion of the permission s indexed by n. If n is zero, we have all of s. If n is positive, we have a token factory over s with n tokens missing, and if n is negative, we have a token of s (of size $-n$).

This points-to operator satisfies the following logical axioms:

$$(\ell \mapsto_{s,0} v * \ell \mapsto_{s,n} v) \quad \leftrightarrow \quad \text{false} \quad (58)$$

$$s_1 \oplus s_2 = s \quad \rightarrow \quad ((\ell \mapsto_{s_1,0} v * \ell \mapsto_{s_2,0} v) \quad \leftrightarrow \quad \ell \mapsto_{s,0} v) \quad (59)$$

$$n_1 \oplus n_2 = n \quad \rightarrow \quad ((\ell \mapsto_{s,n_1} v * \ell \mapsto_{s,n_2} v) \quad \leftrightarrow \quad \ell \mapsto_{s,n} v) \quad (60)$$

$$\ell \mapsto_{s,n} v \quad \rightarrow \quad \exists! s'.\, \ell \mapsto_{s,n} v \quad \leftrightarrow \quad \ell \mapsto_{s',0} v \quad (61)$$

Equation (58) generalizes both the disjointness axiom from Parkinson (24) and the disjointness axiom for token factories (39). Likewise, equation (59) generalizes the share axiom (33). Essentially, if we fix $n = 0$ we get back the simpler definition of the points-to operator from above as a special case. In equation (60), $n_1 \oplus n_2 = n$ refers to the token counting join relation on integers defined in equation 36, and this axiom generalizes the token factory axioms (37) and (38). Both of those axioms follow as a special case when we fix $s = \top$. Finally, equation (61) allows one to project a tokenized share into a nontokenized share (one where $n = 0$). This might be useful if one needs to perform share splitting on a share which was derived from a token factory, for example.

This collection of axioms allow fluid reasoning about both the token-counting and splitting use cases, which enables a unified way to do flexible and precise permission accounting.

8 Conclusion

We have presented a new formulation of multi-unit separation algebras which we find easier to use than the original definition by Calcagno et al. [4]. The original definition is both too restrictive (it rules out desirable constructions, including the naïve disjoint sum and the discrete SA) and too permissive (it allows badly-behaved "exotic" SAs). We examined a variety of operators over separation algebras that allow us to easily construct complicated separation algebras from simpler ones, and have shown an example of their utility.

We have also constructed a new solution to the share accounting problem. Our share model based on Boolean-labeled binary trees fully supports both the splitting and token counting use cases for read sharing, and yet still validates the cross split axiom; it also enjoys a decidable equality test. No previously published system for share accounting has all these properties. Parkinson's model [8] comes closest, but suffers from the inability to find splittings for some shares and lacks decidable equality.

We have implemented the constructions discussed in this paper and proved their relevant properties using the proof assistant Coq.[3]

References

1. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL '05: Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 259–270, 2005.
2. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symp.*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.
3. J. Brotherston and C. Calcagno. Classical BI: a logic for reasoning about dualising resources. In *POPL '09: Proc. of the 36th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 328–339, 2009.
4. C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS '07: Proceedings of the 22nd IEEE Symp. on Logic in Computer Science*, pages 366–378, 2007.
5. A. Hobor. *Oracle Semantics*. PhD thesis, Princeton University, 2008.
6. A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proc. European Symp. on Programming (ESOP 2008)*, volume 4960/2008, pages 353–367, 2008.
7. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL '01: Proc. of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–26, 2001.
8. M. Parkinson. *Local Reasoning for Java*. PhD thesis, Univ. of Cambridge, 2005.
9. D. J. Pym, P. W. O'Hearn, and H. Yang. Possible worlds and resources: the semantics of BI. *Theor. Comput. Sci.*, 315(1):257–305, 2004.
10. G. Restall. *An Introduction to Substructural Logics*. Routledge, London, 2000.
11. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proc. of the 17th Annual IEEE Symp. on Logic in Computer Science*, pages 55–74, 2002.