Improving the Compositionality of Separation Algebras

Aquinas Hobor*

National University of Singapore

Abstract. We show how to improve one of the constructors of separation algebras to increase modularity and expressibility. Our technique has advantages for both paper and mechanized proofs. Our results are implemented in Coq.

1 Introduction

Separation algebras are mathematical structures that track resource accounting and are most commonly used in semantic models of separation logic [1]. Dockins *et al*. showed that new separation algebras could be constructed from old ones by using some of the standard constructors of category theory [2]. This modular approach is extremely practical, especially in a machine-checked setting, and often provides theoretical insight.

However, one of the common constructors presented by Dockins, the so-called "lift" operator, is not as modular as one could hope for because it limits the expressibility of the composed objects in an inconvenient way. Moreover, its implementation in Coq leads to undue hassle due to an awkwardly-placed dependent type.

We present a new pair of constructors whose composition is the lift operator but that allow for greater modularity "in the middle". Our key insight is that while the intermediate structure is not a separation algebra, it is a well-behaved structure in its own right, and by utilizing this structure directly we can allow for greater expressibility than we can by skipping over it. In addition, the new structure puts the dependent type in a much more convenient location, allowing for a much smoother mechanization.

2 Separation Algebras

There are several related definitions of separation algebras; here we use a variant called a disjoint multi-unit separation algebra (DSA) [2]. Briefly, a DSA is a set S and an associated three-place **partial** *join relation* $\oplus$, written $x \oplus y = z$, such that:

(a) A function: $x \oplus y = z_1 \;\Rightarrow\; x \oplus y = z_2 \;\Rightarrow\; z_1 = z_2$
(b) Commutative: $x \oplus y \;=\; y \oplus x$
(c) Associative: $x \oplus (y \oplus z) \;=\; (x \oplus y) \oplus z$
(d) Cancellative: $x_1 \oplus y = z \;\Rightarrow\; x_2 \oplus y = z \;\Rightarrow\; x_1 = x_2$
(e) Multiple units: $\forall x.\, \exists u_x.\, x \oplus u_x = x$
(f) Disjointness: $x \oplus x = y \;\Rightarrow\; x = y$

We say that $x \in S$ is an *identity* if $\forall y, z.\, x \oplus y = z \;\Rightarrow\; y = z$. Observe that "identity" is much stronger than "unit" as given in axiom (e): a unit u_x need only behave as an

* Supported by a Lee Kuan Yew Postdoctoral Fellowship.

identity for the associated element x; the axiom says nothing about how u_x behaves with other elements. However, as it happens, the axioms together imply that all units are identities. Thus, x is an identity if and only if $x \oplus x = x$.

The elements in S can be partitioned into equivalence classes indexed by the identity elements. For each identity element i, two elements a and b are in the same equivalence class if $a \oplus i$ and $b \oplus i$ is defined. Within an equivalence class there is only a single identity; however, since the $\oplus$ operation is partial, multiple identities are possible for the set S as a whole. Elements from distinct equivalence classes never join together.

3 Constructors Over Separation Algebras

Dockins develops a series of standard constructions for building DSAs. One simple construction is the *discrete* DSA, where any set S is given DSA structure by defining

$$s_1 \oplus_= s_2 = s_3 \quad \equiv \quad s_1 = s_2 = s_3 \tag{1}$$

In the discrete DSA, every element is an identity, and no distinct elements join together.

More powerful is the ability to build complicated DSAs from simpler DSAs, including products, sums, functions, etc. Most of these constructions define the joining operation in the "obvious" way. For example, if $(A, \oplus_A)$ and $(B, \oplus_B)$ are two DSAs, then the join function for the product DSA on $A \times B$ is defined componentwise:

$$(a_1, b_1) \oplus_{A \times B} (a_2, b_2) = (a_3, b_3) \quad \equiv \quad (a_1 \oplus_A a_2 = a_3) \wedge (b_1 \oplus_B b_2 = b_3) \tag{2}$$

Similarly, the function DSA from a set A to a DSA $(B, \oplus_B)$ is defined pointwise:

$$f \oplus_{A \to B} g = h \quad \equiv \quad \forall a.\, f(a) \oplus_B g(a) = h(a) \tag{3}$$

Dockins also defines a "lift" operator, which is a kind of coercion between multi-unit separation algebras and single-unit separation algebras. Hereafter we will call this the "smash" operator because of its relationship to the "smash product" of category theory. Starting from a DSA $(A, \oplus)$ one first constructs the related set A^+ as follows:

$$A^+ \quad \equiv \quad \{a \in A \mid \neg \mathsf{identity}\ a\} \tag{4}$$

That is, A^+ is obtained from A by removing all the identity elements over $\oplus$. One then adds some distinguished element $\bot \notin A^+$ to reach the set $A^+_\bot$ and defines the join relation $\oplus_\bot$ as the least relation satisfying the following rules:

$$\begin{array}{lll} 1. & \bot \oplus_\bot a = a & \\ 2. & a \oplus_\bot \bot = a & \\ 3. & a_1 \oplus a_2 = a_3 & \to \quad a_1 \oplus_\bot a_2 = a_3 \end{array} \tag{5}$$

One common use for the smash operator is to construct a separation algebra for resource sharing in a concurrent language as follows. We start with a *share model*: a separation algebra $(\mathcal{S}, \oplus_\mathcal{S})$ that models what kinds of sharing we would like to support. One simple share model defines $\mathcal{S} = \{q \mid 0 \leq q \leq 1\}$ and $\oplus_\mathcal{S}$ as partial addition (*i.e.*,

undefined when the sum is greater than 1)[1]. In a concurrent setting, a share of 1 (of some resource) will denote full ownership; 0 will denote no ownership; and $0 < x < 1$ will denote partial ownership. Let $\mathcal{L}$ be a set of heap locations (addresses) and $\mathcal{V}$ be a set of values in our operational semantics. Now we can define heaps $\mathcal{H}$ as follows:

$$\mathcal{H} \quad \equiv \quad \mathcal{L} \to (\mathcal{S} \times \mathcal{V}_{=})_{\perp} \tag{6}$$

This simple-looking equation describes **both** the type of heaps (functions from locations to a pointed set containing pairs of shares and values) **and** how the join function should be constructed: start with the function constructor (3) from $\mathcal{L}$ to the smashed (5) product (2) of shares $\mathcal{S}$ and values $\mathcal{V}$ under the discrete construction (1). That is, each location is associated with either the distinguished element $\perp$, indicating that the thread has now ownership, or by a nonempty ($\neg\mathsf{identity}$) pair of share and value. Using the constructors guarantees that axioms (a)–(f) hold on heaps $\mathcal{H}$ without having to prove them directly.

4 The Problem with Smash

As elegant as equation (6) is, its form conceals some problems, particularly with regard to the smash operator. The first problem is the location of the side condition from equation (4): that is, the side condition that $(\mathcal{S} \times \mathcal{V}_{=})^{+}$ contains no identity elements. A quick reflection reveals that under the product DSA constructor a pair is not an identity if and only if at least one of its components is not be an identity, *i.e.*:

$$\neg\mathsf{identity}(a, b) \quad \Leftrightarrow \quad (\neg\mathsf{identity}\ a)\ \vee\ (\neg\mathsf{identity}\ b) \tag{7}$$

Since **every** element in $\mathcal{V}_{=}$ is an identity, however, for $(\mathcal{S} \times \mathcal{V}_{=})^{+}$ we have the following:

$$\neg\mathsf{identity}(s, v) \quad \Leftrightarrow \quad \neg\mathsf{identity}\ s \tag{8}$$

In other words, in set theory $(\mathcal{S} \times \mathcal{V}_{=})^{+}$ **equals** $\mathcal{S}^{+} \times \mathcal{V}_{=}$. However, in type theory the two are not quite equal since the non-identity side condition is carried around via a dependent (Σ-) type, and the definitions differ due to where this is placed. That is,

$$(\mathcal{S} \times \mathcal{V}_{=})^{+} \quad \equiv \quad \Sigma p : (\mathcal{S} \times \mathcal{V}).\ \neg\mathsf{identity}\ p \tag{9}$$

is what one gets by applying the smash operator, instead of the more preferable

$$\mathcal{S}^{+} \times \mathcal{V}_{=} \quad \equiv \quad (\Sigma s : \mathcal{S}.\ \neg\mathsf{identity}\ s) \times \mathcal{V} \tag{10}$$

One reason to prefer equation (10) over equation (9) is that the dependent type is "closer" to the object being restricted. Consider the operation of updating a heap cell: one takes apart a share/value pair (s, v) and reconstitutes the new pair (s, v'). If we are using equation (9), we discover a new proof obligation during reconstitution:

$$\mathsf{identity}\ (s, v) \ \Rightarrow\ \mathsf{identity}\ (s, v') \tag{11}$$

[1] Dockins develops more powerful kinds of share models, but this is sufficient here [2].

Of course, this obligation is not very hard to satisfy using equation (8), but it is inconvenient that it shows up at all. In fact, in a fully-formal development (mechanized or paper), almost every use of heaps runs into similar irritating problems. Although it may seem minor, these kinds of "stupid obligations" can take up a surprising amount of effort: for example, in a recent proof development approximately 5% of the Coq code (more than 500 lines, distributed over more than a hundred places in the development!) was dealing with this kind (and related) obligations [3]. Needless to say, these were not the most interesting parts of the mechanization effort.

Using equation (10), breaking apart an (s, v) pair is simpler since one does not end up with an associated $\neg\mathsf{identity}\,(s, v)$ proof, and updating the value component of a pair is simpler since the dependency is directly attached to the share, which is reconstituted into the new pair unchanged. Only updating the share itself requires an update of the associated proof—and even then, the value does not get involved.

Beyond the engineering concerns outlined above, there are good theoretical reasons to prefer a style closer to equation (10). In the particular example given in equation (6), the values $\mathcal{V}$ were only given the trivial discrete separation structure, but in general one wants to be able to use richer separation structures. However, if one does give $\mathcal{V}$ a richer structure, one runs into the problem that equation (8) no longer holds. Instead, even if one's intention is to restrict the shares to the nonidentity elements, one must return to the more general disjunctive equivalence given in equation (7). This inconvenient disjunction may require the imposition of additional side conditions in awkward places.

Why do we not apply the smash operator directly to shares $\mathcal{S}$, as equation (10) would seem to suggest? Unfortunately, if we do so then we produce not the positive shares, but the positive shares **plus** a fresh bottom element. That is, we end up with $\mathcal{S}_\bot \times \mathcal{V}_=$ instead of $(\mathcal{S}^+ \times \mathcal{V}_=)_\bot$. These are not the same: the left-hand has infinitely many identity elements ($(\bot, v)$ for all v), whereas the right-hand has only one ($\bot$).

Another question is why we do not simply define some constructor from a DSA $(A, \oplus_A)$ to some positive subset DSA $(A^+, \oplus_{A^+})$. The answer is that the positive subset lacks identities and thus does not satisfy axiom (e)—that is, $(A^+, \oplus_{A^+})$ **is not a DSA**.

5 Positive Disjoint Separation Algebras

The good news is that $(\mathcal{S}^+ \times \mathcal{V}_=)_\bot$ **does** satisfy the DSA axioms; the bad news is that it is not decomposable componentwise into DSA subparts: we have lost modularity.

However, $\mathcal{S}^+ \times \mathcal{V}_=$ is **almost** a DSA; indeed, it is a member of a well-behaved mathematical structure hereby christened a *positive disjoint separation algebra* (PDSA). A PDSA, like a DSA, is a set S^+ and an associated 3-place positive join relation $\oplus^+$. A PDSA's $\oplus$ satisfies axioms (a)–(d) from a DSA. Since it does not have any units, it drops axiom (e), and to enforce positivity it uses a modified version of axiom (f):

(f′) Positive Disjointness: $x \oplus x = y \;\Rightarrow\; \mathsf{False}$

That is, f′ says that no elements join with themselves. PDSAs are well-behaved because they also enjoy many of the constructions from category theory (*e.g.*, products, sums).

What is particularly interesting is how PDSAs and DSAs can be constructed from each other. For example, let us suppose that we have a PDSA such as $\mathcal{S}^+ \times \mathcal{V}_=$; we can define the *lowering constructor*, which takes a PDSA $(S^+, \oplus^+)$ and turns it into a DSA

by adding a fresh unit $\bot$ and by defining $\oplus_\downarrow$ as the least relation satisfying:

$$\begin{array}{lllllll} 1. & \bot & \oplus_\downarrow & a & = & a & \\ 2. & a & \oplus_\downarrow & \bot & = & a & \\ 3. & a_1 & \oplus^+ & a_2 & = & a_3 & \rightarrow \quad a_1 \oplus_\downarrow a_2 = a_3 \end{array} \tag{12}$$

By design, this construction is remarkably similar to the construction for the smash operator; the difference is that while the smash operator takes a DSA and produces another DSA, the lowering operator takes a PDSA and produces a DSA.

In contrast, the *lift constructor* takes a DSA and produces a PDSA by removing all of the units. If $(S, \oplus)$ is a DSA, then the lifted join operation on S^+, written $\oplus_\uparrow$, is:

$$s_1^+ \oplus_\uparrow s_2^+ = s_3^+ \qquad \equiv \qquad s_1^+ \oplus s_2^+ = s_3^+ \tag{13}$$

In other words, the lift operator gets its join structure directly from the underlying operation. The purpose is to move from a DSA to a PDSA by exchanging S with S^+.

We observe that the new lower and lift operators are at least as powerful as the old smash operator because for any DSA $(S, \oplus)$ we have $S_\bot \cong (S_\uparrow)_\downarrow$. The question then is if we can get any additional leverage out of the intermediate structure. The answer, happily, is that we can by defining other operators that connect DSAs and PDSAs.

One important additional operator we can define is the *semiproduct constructor*, which takes a PDSA $(A^+, \oplus^+_{A^+})$ and DSA $(B, \oplus_B)$ and forms a PDSA over the set $A^+ \times B$ by defining the positive join operation $\oplus^+_{A^+ \times B}$ as follows

$$(a_1^+, b_1) \oplus^+_{A^+ \times B} (a_2^+, b_2) = (a_3^+, b_3) \quad \equiv \quad (a_1^+ \oplus^+_{A^+} a_2^+ = a_3^+) \wedge (b_1 \oplus_B b_2 = b_3) \tag{14}$$

That is, we define the semiproduct constructor componentwise in exactly the same fashion as the regular product constructor on DSAs (and also the regular product constructor on PDSAs). However, since the operation takes different structures on the left- and right-hand sides, we prefer to write $A^+ \propto B$ to indicate the semiproduct[2].

There are other kinds of constructions that illustrate the connections between DSAs and PDSAs, but we already have enough to reformulate heaps $\mathcal{H}$ the way we want:

$$\mathcal{H} \qquad \equiv \qquad \mathcal{L} \rightarrow (\mathcal{S}_\uparrow \propto \mathcal{V}_=)_\downarrow \tag{15}$$

That is, heaps $\mathcal{H}$ are functions (3) from locations $\mathcal{L}$ to a lowered (12) semiproduct (14) of a lifted (13) share $\mathcal{S}$ and discrete (1) value $\mathcal{V}$. Elements of $\mathcal{H}$ are functions to a pointed set whose nonidentity elements are exactly of the desirable form given in equation (10): that is, the dependent type is attached to the share instead of the share-value pair.

We can illustrate the additional expressivity our PDSAs with the following observation. Suppose we have an additional DSA $(\mathcal{O}, \oplus_\mathcal{O})$ that we want to attach to the heap cells. We can easily modify our heaps to accommodate this as follows:

$$\mathcal{H}' \qquad \equiv \qquad \mathcal{L} \rightarrow (\mathcal{S}_\uparrow \propto (\mathcal{V}_= \times \mathcal{O}))_\downarrow \tag{16}$$

[2] We could also define a semiproduct that takes a DSA on the left and a PDSA on the right.

This is better than what we would obtain with the smash operator because it **specifies exactly whence positivity is obtained**. If we tried to do the same with smash:

$$\mathcal{H}'' \qquad \equiv \qquad \mathcal{L} \to (\mathcal{S} \times \mathcal{V}_{=} \times \mathcal{O})_{\perp} \tag{17}$$

Now if we have a nonidentity (s, v, o) triple from $\mathcal{H}''$, then we do not know whether the share s is positive or if the other data o is positive. In contrast, the same triple from $\mathcal{H}'$ (equation 16) **must** have s positive. This is a feature, not a bug, but if we did want o to be positive as well then we can use $(\mathcal{S}_{\uparrow} \propto \mathcal{V}_{=}) \times \mathcal{O}_{\uparrow}$. We could allow the same ambiguity that the smash operator does by using $((\mathcal{S} \times \mathcal{O} \times \mathcal{V}_{=})_{\uparrow})_{\downarrow}$, and so forth. The key point is that we have greater expressibility with lift/lower than we have with smash.

6 Conclusion

References

1. Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Symposium on Logic in Computer Science*, 2007.
2. Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *The 7th Asian Symposium on Programming Languages and Systems*, pages 161–177. Springer ENTCS, 2009.
3. Aquinas Hobor and Cristian Gherghina. Barriers in concurrent separation logic. In *20th European Symposium of Programming (ESOP 2011)*, page to appear, 2011.