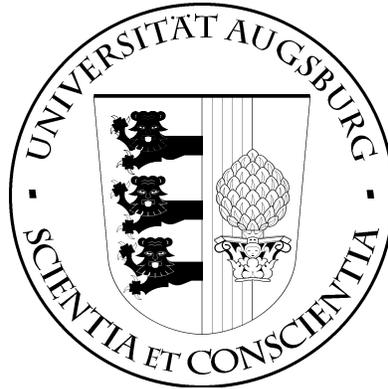


UNIVERSITÄT AUGSBURG

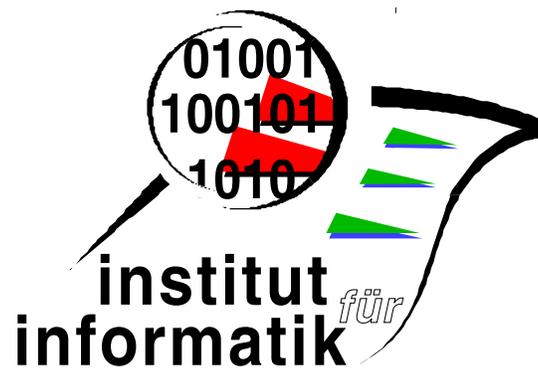


Towards Algebraic Separation Logic

H.-H. Dang P. Höfner B. Möller

Report 2009-12

June 2009



INSTITUT FÜR INFORMATIK

D-86135 AUGSBURG

Copyright © H.-H. Dang P. Höfner B. Möller
Institut für Informatik
Universität Augsburg
D-86135 Augsburg, Germany
<http://www.Informatik.Uni-Augsburg.DE>
— all rights reserved —

Towards Algebraic Separation Logic

Han-Hing Dang, Peter Höfner, and Bernhard Möller

Institut für Informatik, Universität Augsburg, D-86135 Augsburg, Germany
{h.dang,hoefner,moeller}@informatik.uni-augsburg.de

Abstract. We present an algebraic approach to separation logic. In particular, we give algebraic characterisations for all constructs of separation logic like assertions and commands. The algebraic view does not only yield new insights on separation logic but also shortens proofs and enables the use of automated theorem provers for verifying properties at a more abstract level.

1 Introduction

In many applications correctness of programs is essential. Over the last decades formal methods have found their way into specification and verification. The most prominent ones are probably Hoare logic [7] and the wp-calculus of Dijkstra [5]. However, they lack expressiveness for shared mutable data structures, i.e., structures where updatable fields can be referenced from more than one point (e.g. [18]). To overcome this deficiency Reynolds, O’Hearn and others have developed *separation logic* that allows reasoning about such data structures. Their approach extends Hoare logic by assertions to express separation within memory, both in store and heap. Furthermore the command language is enriched by some constructs that allows altering these separate ranges. The introduced mechanism has been extended to allow reasoning about concurrent programs that work on shared mutable data structures [16].

This paper presents an algebraic approach to separation logic. As a result many proofs become simpler while still being fully precise. Moreover, this places the topic into a more general context and therefore allows re-use of a large body of existing theory.

In Section 2 we recapitulate syntax and semantics of expressions in separation logic and give a formal definition of an update-operator for relations. Section 3 gives the semantics of assertions. After providing the algebraic background in Section 4, we shift from the validity semantics of separation logic to one based on the set of states satisfying an assertion. Abstracting from the set view yields an algebraic interpretation of assertions in the setting of semirings and quantales. In Section 6 we discuss special classes of assertions: pure assertions do not depend on the heap at all; intuitionistic assertions do not specify the heap exactly. After that we extend our algebra to cover the command language of separation logic in Section 7. We conclude with a short outlook.

2 Basic Definitions

Separation logic, as an extension of Hoare logic, does not only allow reasoning about explicitly named program variables, but also about anonymous variables in dynamically allocated storage. Therefore a program state in separation logic consists of a *store* and a *heap*. In the remainder we consistently write s for stores and h for heaps.

To simplify the formal treatment, one defines values and addresses as integers, stores and heaps as partial functions from variables or addresses to values and states as pairs of stores and heaps:

$$\begin{aligned} \text{Values} &= \mathbb{Z} , \\ \{\text{nil}\} \cup \text{Addresses} &\subseteq \text{Values} , \\ \text{Stores} &= V \rightsquigarrow \text{Values} , \\ \text{Heaps} &= \text{Addresses} \rightsquigarrow \text{Values} , \\ \text{States} &= \text{Stores} \times \text{Heaps} , \end{aligned}$$

where V is the set of all variables, \cup denotes the disjoint union on sets and $M \rightsquigarrow N$ denotes the set of partial functions between M and N . With this definition, we slightly deviate from [18] where stores are defined as functions from variables to values of \mathbb{Z} and heaps as functions from addresses into values of \mathbb{Z} , while addresses are also values of \mathbb{Z} .

The constant `nil` is a value for pointers that denotes an improper reference like `null` in programming languages like `JAVA`; by the above definitions, `nil` is not an address and hence heaps do not assign values to `nil`.

As usual we denote the domain of a relation (partial function) R by $\text{dom}(R)$:

$$\text{dom}(R) =_{df} \{x : \exists y. (x, y) \in R\} .$$

In particular, the domain of a store denotes all currently used program variables and $\text{dom}(h)$ is the set of all currently allocated addresses on a heap h .

As in [13] and for later definitions we also need an operator called *update* operator. This operator is used to model changes in stores and heaps. We will first give a definition and then explain its meaning.

Let R and S be partial functions. Then we define

$$R | S =_{df} R \cup \{(x, y) \mid (x, y) \in S \wedge x \notin \text{dom}(R)\} . \quad (1)$$

Hence the relation R updates the relation S with all possible pairs of R in such a way that $R | S$ is again a partial function. The domain of the right hand side of \cup above is disjoint from that of R . In particular, $R | S$ can be seen as an extension of R to $\text{dom}(R) \cup \text{dom}(S)$. In later definitions we abbreviate an update $\{(x, v)\} | S$ on a single variable or address by omitting the set-braces and simply writing $(x, v) | S$ instead.

Expressions are used to denote values or Boolean conditions on stores and are independent of the heap, i.e., they only need the store component of a given state

for their evaluation. Informally *exp*-expressions are simple arithmetical expressions over variables and values, while *bexp*-expressions are Boolean expressions over simple comparisons and true, false. Their syntax is given by

$$\begin{aligned} var &::= x \mid y \mid z \mid \dots \\ exp &::= 0 \mid 1 \mid 2 \mid \dots \mid var \mid exp \pm exp \mid \dots \\ bexp &::= \text{true} \mid \text{false} \mid exp = exp \mid exp < exp \mid \dots \end{aligned}$$

The semantics e^s of an expression e w.r.t. a store s is straightforward (assuming that all variables occurring in e are contained in $\text{dom}(s)$). For example,

$$c^s = c \quad \forall c \in \mathbb{Z}, \quad \text{true}^s = \text{true} \quad \text{and} \quad \text{false}^s = \text{false}.$$

3 Assertions

Assertions play an important rôle in separation logic. They are used as predicates to describe the contents of heaps and stores and as pre- or postconditions in programs, like in Hoare logic:

$$\begin{aligned} assert &::= bexp \mid \neg assert \mid assert \vee assert \mid \forall var. assert \mid \\ &\text{emp} \mid exp \mapsto exp \mid assert * assert \mid assert -* assert. \end{aligned}$$

In the remainder we consistently write p , q and r for assertions. Assertions are split into two parts: the “classical” ones from predicate logic and four new ones that express properties of the heap. The former are supplemented by the logical connectives \wedge , \rightarrow and \exists that are defined, as usual, by $p \wedge q =_{df} \neg(\neg p \vee \neg q)$, $p \rightarrow q =_{df} \neg p \vee q$ and $\exists v. p =_{df} \neg \forall v. \neg p$.

The semantics of assertions is given by the relation $s, h \models p$ of *satisfaction*. It is defined inductively as follows (see e.g. [18]).

$$\begin{aligned} s, h \models b &\iff_{df} b^s = \text{true} \\ s, h \models \neg p &\iff_{df} s, h \not\models p \\ s, h \models p \vee q &\iff_{df} s, h \models p \quad \text{or} \quad s, h \models q \\ s, h \models \forall v. p &\iff_{df} \forall x \in \mathbb{Z} : (v, x) \mid s, h \models p \\ s, h \models \text{emp} &\iff_{df} h = \emptyset \\ s, h \models e_1 \mapsto e_2 &\iff_{df} h = \{(e_1^s, e_2^s)\} \\ s, h \models p * q &\iff_{df} \exists h_1, h_2 \in \text{Heaps} : \text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset \text{ and} \\ &\quad h = h_1 \cup h_2 \text{ and } s, h_1 \models p \text{ and } s, h_2 \models q \\ s, h \models p -* q &\iff_{df} \forall h' \in \text{Heaps} : (\text{dom}(h') \cap \text{dom}(h) = \emptyset \text{ and } s, h' \models p) \\ &\quad \text{implies } s, h' \cup h \models q. \end{aligned}$$

Here e is an *exp*-expression, b a *bexp*-expression and p , q are assertions. Informally, $s, h \models p$ holds if the state (s, h) satisfies the assertion p ; an assertion p is called *valid* iff p holds in every state and finally p is *satisfiable* if there exists a state (s, h) which satisfies p . The first four clauses do not make any assumptions about the heap and only carry it along without making any changes to it; they are well known from predicate logic or Hoare logic [7].

The remaining lines describe the new parts in separation logic: For an arbitrary state (s, h) , `emp` ensures that the heap h is empty and contains no addressable cells. An assertion $e_1 \mapsto e_2$ with expressions e_1 and e_2 characterises states with the singleton heap that has exactly one cell at the address e_1^s with the value e_2^s . To reason about more complex heaps, the *separating conjunction* $*$ is used. It allows expressing properties of heaps that result from merging smaller disjoint heaps, i.e., heaps with disjoint domains.

The *separating implication* $p \multimap q$ guarantees that if the current heap h is extended with a heap h' satisfying p , the merged heap $h \cup h'$ satisfies q (see [18] and Figure 1). If the heaps are not disjoint, the situation is interpreted as an error case and the assertion is not satisfied.

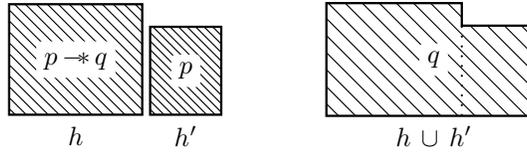


Fig. 1. Separating implication ¹

4 Quantales and Residuals

To present our algebraic semantics of separation logic in the next section we now prepare the algebraic background.

A *quantale* [19] is a structure $(S, \leq, 0, \cdot, 1)$ where (S, \leq) is a complete lattice and \cdot is completely disjunctive. The infimum and supremum of two elements $a, b \in S$ are denoted by $a \sqcap b$ and $a + b$, resp. The greatest element of S is denoted by \top . The definition implies that \cdot is strict, i.e., that $0 \cdot a = 0 = a \cdot 0$ for all $a \in S$. The notion of a quantale is equivalent to that of a *standard Kleene algebra* [3] and a special case of the notion of an idempotent semiring.

A quantale is called *Boolean* if its underlying lattice is distributive and complemented, whence a Boolean algebra. Equivalently, a quantale S is Boolean if it satisfies the Huntington axiom $a = \overline{\overline{a} + \overline{b}} + \overline{\overline{a} + \overline{b}}$ for all $a, b \in S$ [11, 10]. The infimum is then defined by the de Morgan duality $a \sqcap b =_{df} \overline{\overline{a} + \overline{b}}$. An important Boolean quantale is REL, the algebra of binary relations over a set under union and composition.

A quantale is called *commutative* if \cdot commutes, i.e., $a \cdot b = b \cdot a$ for all a, b .

In any quantale, the *right residual* $a \setminus b$ exists and is characterised by the Galois connection

$$x \leq a \setminus b \Leftrightarrow_{df} a \cdot x \leq b.$$

¹ The right picture might suggest that the heaps are adjacent after the join. But the intention is only to bring out abstractly that the united heap satisfies q .

Symmetrically, the *left residual* b/a can be defined. However, if the underlying quantale is commutative then both residuals coincide, i.e., $a \setminus b = b/a$. In REL, $R \setminus S$ is characterised by $R \setminus S =_{df} \overline{R^\smile ; S}$, where \smile denotes relational converse and $;$ is relational composition.

5 An Algebraic Model of Assertions

We now give an algebraic interpretation for the semantics of separation logic. The main idea is to switch from the satisfaction-based semantics for single states to an equivalent set-based one where every assertion is associated with the set of all states satisfying it. This considerably eases the proofs of most of the laws given in [18].

For an arbitrary assertion p we therefore define its set-based semantics as

$$\llbracket p \rrbracket =_{df} \{(s, h) : s, h \models p\}.$$

The sets $\llbracket p \rrbracket$ of states will be the elements of our algebra. By this we then have immediately the connection $s, h \models p \Leftrightarrow (s, h) \in \llbracket p \rrbracket$. This validity assertion can be lifted to set of states by setting, for $A \subseteq States$, $A \models p \Leftrightarrow A \subseteq \llbracket p \rrbracket$. The embedding of the standard Boolean connectives is given by

$$\begin{aligned} \llbracket \neg p \rrbracket &= \{(s, h) : s, h \not\models p\} = \overline{\llbracket p \rrbracket}, \\ \llbracket p \vee q \rrbracket &= \llbracket p \rrbracket \cup \llbracket q \rrbracket, \\ \llbracket \forall v. p \rrbracket &= \{(s, h) : \forall x \in \mathbb{Z}. (v, x) \mid s, h \models p\}. \end{aligned}$$

Using these definitions, it is straightforward to show that

$$\begin{aligned} \llbracket p \wedge q \rrbracket &= \llbracket p \rrbracket \cap \llbracket q \rrbracket, & \llbracket p \rightarrow q \rrbracket &= \overline{\llbracket p \rrbracket} \cup \llbracket q \rrbracket, & \text{and} \\ \llbracket \exists v. p \rrbracket &= \overline{\llbracket \forall v. \neg p \rrbracket} = \{(s, h) : \exists x \in \mathbb{Z}. (v, x) \mid s, h \models p\}, \end{aligned}$$

where \mid is the update operation defined in (1).

The emptiness assertion \mathbf{emp} and the assertion operator \mapsto are given by

$$\begin{aligned} \llbracket \mathbf{emp} \rrbracket &=_{df} \{(s, h) : h = \emptyset\} \\ \llbracket e_1 \mapsto e_2 \rrbracket &=_{df} \{(s, h) : h = \{(e_1^s, e_2^s)\}\}. \end{aligned}$$

Next, we model the separating conjunction $*$ algebraically by

$$\begin{aligned} \llbracket p * q \rrbracket &=_{df} \llbracket p \rrbracket \cup \llbracket q \rrbracket, \\ P \cup Q &=_{df} \{(s, h \cup h') : (s, h) \in P \wedge (s, h') \in Q \wedge \text{dom}(h) \cap \text{dom}(h') = \emptyset\}. \end{aligned}$$

In this way inconsistent states as well as “erroneous” merges of non-disjoint heaps are excluded.

These definitions yield an algebraic embedding of separation logic.

Theorem 5.1 *The structure $\mathbf{AS} =_{df} (\mathcal{P}(States), \subseteq, \emptyset, \cup, \llbracket \mathbf{emp} \rrbracket)$ is a commutative and Boolean quantale with $P + Q = P \cup Q$.*

The proof is by straightforward calculations; it can be found in [4]. It is easy to show that $\llbracket \text{true} \rrbracket$ is the greatest element in the above quantale, i.e., $\llbracket \text{true} \rrbracket = \top$, since every state satisfies the assertion true . This implies immediately that $\llbracket \text{true} \rrbracket$ is the neutral element for \sqcap . However, in contrast to addition \cup , multiplication \cup is in general not idempotent.

Example 5.2 In AS

$$\llbracket (x \mapsto 1) * (x \mapsto 1) \rrbracket = \llbracket x \mapsto 1 \rrbracket \cup \llbracket x \mapsto 1 \rrbracket = \emptyset .$$

This can be shown by straightforward calculations using the above definitions.

$$\begin{aligned} & \llbracket (x \mapsto 1) * (x \mapsto 1) \rrbracket \\ &= \{(s, h) \cup (s', h') : (s, h), (s', h') \in \llbracket x \mapsto 1 \rrbracket\} \\ &= \{(s, h \cup h') : (s, h), (s, h') \in \llbracket x \mapsto 1 \rrbracket \wedge \text{dom}(h) \cap \text{dom}(h') = \emptyset\} \\ &= \emptyset . \end{aligned}$$

$\llbracket x \mapsto 1 \rrbracket$ is the set of all states that have the single-cell heap $\{(s(x), 1)\}$. The states (s, h) and (s, h') have to share this particular heap. Hence the domains of the merged heaps would not be disjoint. Therefore the last step yields the empty result. \square

As a check of the adequacy of our definitions we list a couple of properties.

Lemma 5.3 *In separation logic, for assertions p, q, r , we have*

$$\overline{(p \wedge q) * r \Rightarrow (p * r) \wedge (q * r)} \quad \text{and} \quad \frac{p \Rightarrow r \quad q \Rightarrow s}{p * q \Rightarrow r * s} .$$

The second property denotes isotony of separating conjunction.

Proof. As an example we prove the first statement. Algebraically, it corresponds to $(p \sqcap q) \cdot r \leq (p \cdot r) \sqcap (q \cdot r)$ which is shown by idempotence and isotony of \sqcap :

$$(p \sqcap q) \cdot r = ((p \sqcap q) \cdot r) \sqcap ((p \sqcap q) \cdot r) \leq (p \cdot r) \sqcap (q \cdot r) . \quad \square$$

More laws and examples can be found in [4].

For the separating implication the set-based semantics extracted from the definition in Section 3 is

$$\begin{aligned} \llbracket p \multimap q \rrbracket &=_{df} \{(s, h) : \forall h' \in \text{Heaps} : (\text{dom}(h) \cap \text{dom}(h') = \emptyset \wedge (s, h') \in \llbracket p \rrbracket) \\ &\quad \Rightarrow (s, h \cup h') \in \llbracket q \rrbracket\} . \end{aligned}$$

This implies that separating implication corresponds to a residual.

Lemma 5.4 *In AS, $\llbracket p \multimap q \rrbracket = \llbracket p \rrbracket \setminus \llbracket q \rrbracket = \llbracket q \rrbracket / \llbracket p \rrbracket$.*

Proof. We first show the claim for a single state. By definition above, set theory and definition of \cup , we have

$$\begin{aligned}
& (s, h) \in \llbracket p \multimap q \rrbracket \\
\Leftrightarrow & \forall h' : ((s, h') \in \llbracket p \rrbracket \wedge \text{dom}(h) \cap \text{dom}(h') = \emptyset \Rightarrow (s, h \cup h') \in \llbracket q \rrbracket) \\
\Leftrightarrow & \{(s, h \cup h') : (s, h') \in \llbracket p \rrbracket \wedge \text{dom}(h) \cap \text{dom}(h') = \emptyset\} \subseteq \llbracket q \rrbracket \\
\Leftrightarrow & \{(s, h)\} \cup \llbracket p \rrbracket \subseteq \llbracket q \rrbracket .
\end{aligned}$$

and therefore, for arbitrary set R of states,

$$\begin{aligned}
& R \subseteq \llbracket p \multimap q \rrbracket \\
\Leftrightarrow & \forall (s, h) \in R : (s, h) \in \llbracket p \multimap q \rrbracket \\
\Leftrightarrow & \forall (s, h) \in R : \{(s, h)\} \cup \llbracket p \rrbracket \subseteq \llbracket q \rrbracket \\
\Leftrightarrow & R \cup \llbracket p \rrbracket \subseteq \llbracket q \rrbracket .
\end{aligned}$$

Hence, by definition of the residual, $\llbracket p \multimap q \rrbracket = \llbracket p \rrbracket \setminus \llbracket q \rrbracket$. The second equation follows immediately from Section 2, since multiplication \cup in AS commutes. \square

Now all laws of [18] about \multimap follow from the standard theory of residuals (e.g. [2]). Many of these laws are proved algebraically in [4]. For example, the two main properties of separating implication, namely the currying and decurrying rules, are nothing but the transcriptions of the defining Galois connection for right residuals.

Corollary 5.5 *In separation logic the following inference rules hold:*

$$\frac{p * q \Rightarrow r}{p \Rightarrow (q \multimap r)} , \quad (\text{currying}) \qquad \frac{p \Rightarrow (q \multimap r)}{p * q \Rightarrow r} . \quad (\text{decurrying})$$

This means that $q \multimap r$ is the weakest assertion guaranteeing that a state in $\llbracket q \multimap r \rrbracket$ merged with a state in $\llbracket q \rrbracket$ yields a state in $\llbracket r \rrbracket$.

In his work Reynolds only derives these laws in separation logic (e.g. [18]). We are not aware of any proof of the equivalence of Lemma 5.4, although many authors state this claim and refer to Reynolds.

As a further example we prove the algebraic counterpart of the inference rule

$$\overline{q * (q \multimap p) \Rightarrow p} .$$

Lemma 5.6 *Let S be a quantale. For $a, b \in S$ the inequality $q \cdot (q \setminus p) \leq p$ holds.*

Proof. By definition of residuals we immediately get

$$q \cdot (q \setminus p) \leq p \Leftrightarrow q \setminus p \leq q \setminus p \Leftrightarrow \text{true} . \quad \square$$

6 Special Classes of Assertions

Reynolds distinguishes different classes of assertions [18]. We will give algebraic characterisations for two main classes, namely *pure assertions* and *intuitionistic assertions*. The former are independent of the heap of a state. Therefore these assertions only express conditions on store variables. The latter do not describe the domain of a heap exactly. Hence, when using these assertions one does not know whether the heap contains additional anonymous cells.

6.1 Pure Assertions

An assertion p is called *pure* iff it is independent of the heaps of the states involved, i.e.,

$$p \text{ is pure} \Leftrightarrow_{df} (\forall h, h' \in \text{Heaps} : s, h \models p \Leftrightarrow s, h' \models p) .$$

Theorem 6.1 *In AS an element $\llbracket p \rrbracket$ is pure iff it satisfies, for all $\llbracket q \rrbracket$ and $\llbracket r \rrbracket$,*

$$\llbracket p \rrbracket \cup \llbracket \text{true} \rrbracket \subseteq \llbracket p \rrbracket \quad \text{and} \quad \llbracket p \rrbracket \cap (\llbracket q \rrbracket \cup \llbracket r \rrbracket) \subseteq (\llbracket p \rrbracket \cap \llbracket q \rrbracket) \cup (\llbracket p \rrbracket \cap \llbracket r \rrbracket) .$$

Before we give the proof we derive an abstract law to shorten it. The above theorem motivates the following definition.

Definition 6.2 In an arbitrary Boolean quantale S an element p is called *pure* iff it satisfies, for all $a, b \in S$,

$$p \cdot \top \leq p , \tag{2}$$

$$p \sqcap (a \cdot b) \leq (p \sqcap a) \cdot (p \sqcap b) . \tag{3}$$

The first equation models upwards closure of pure elements. It can be strengthened to an equation since its converse holds for arbitrary Boolean quantales. The second equation enables pure elements to distribute over meet and implies downward closure. It can be strengthened to equations if $p \cdot p \leq p$. This is for example the case if (2) holds. Moreover we get a compact characterisation if the underlying quantale commutes.

Lemma 6.3 *In a commutative and Boolean quantale, an element p is pure iff $p = (p \sqcap 1) \cdot \top$ holds.*

Proof. We first show that $p = (p \sqcap 1) \cdot \top$ follows from Equations (2) and (3). By neutrality of \top for \sqcap , neutrality of 1 for \cdot , meet-distributivity (3) and isotony, we get

$$p = p \sqcap \top = p \sqcap (1 \cdot \top) \leq (p \sqcap 1) \cdot (p \sqcap \top) \leq (p \sqcap 1) \cdot \top .$$

The converse inequation follows by isotony and Equation (2):

$$(p \sqcap 1) \cdot \top \leq p \cdot \top \leq p .$$

Next we show that $p = (p \sqcap 1) \cdot \top$ implies the two Equations (2) and (3). The first equation is shown by the assumption, the general law $\top \cdot \top = \top$ and the assumption again:

$$p \cdot \top = (p \sqcap 1) \cdot \top \cdot \top = (p \sqcap 1) \cdot \top = p .$$

For the second equation, we note that in a Boolean quantale the law $(s \cdot a) \sqcap b = a \sqcap (s \cdot b)$ holds for all subidentities s ($s \leq 1$) (e.g., [15, 8]). From this we get for arbitrary c by the assumption

$$p \sqcap c = ((p \sqcap 1) \cdot \top) \sqcap c = \top \sqcap ((p \sqcap 1) \cdot c) = (p \sqcap 1) \cdot c . \tag{4}$$

Moreover in a Boolean quantale subidentities are idempotent, i.e., $s \cdot s = s$. Using Equation (4), idempotence, commutativity and Eq. (4) (twice) we get the claim:

$$\begin{aligned} p \sqcap (a \cdot b) &= (p \sqcap 1) \cdot a \cdot b = (p \sqcap 1) \cdot (p \sqcap 1) \cdot a \cdot b \\ &= (p \sqcap 1) \cdot a \cdot (p \sqcap 1) \cdot b = (p \sqcap a) \cdot (p \sqcap b) . \end{aligned} \quad \square$$

Corollary 6.4 *The set of pure elements forms a complete lattice.*

Proof. Lemma 6.3 characterises the pure elements as the fixed points of the isotone function $f(x) = (x \sqcap 1) \cdot \top$ on the quantale. By Tarski's fixed point theorem these form a complete lattice. \square

Proof of Theorem 6.1. By Lemma 6.3 and definition of the elements of AS it is sufficient to show that the following formulas are equivalent in separation logic

$$\forall s \in Stores, \forall h, h' \in Heaps : (s, h \models p \Leftrightarrow s, h' \models p) , \quad (5)$$

$$\forall s \in Stores, \forall h \in Heaps : (s, h \models p \Leftrightarrow s, h \models (p \wedge \text{emp}) * \text{true}) . \quad (6)$$

Since both assertions are universally quantified over states we omit that quantification in the remainder and only keep the quantifiers on heaps. Before proving this equivalence we simplify $s, h \models (p \wedge \text{emp}) * \text{true}$. Using the definitions of Section 3, we get for all $h \in Heaps$

$$\begin{aligned} & s, h \models (p \wedge \text{emp}) * \text{true} \\ \Leftrightarrow & \exists h_1, h_2 \in Heaps : \text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset \text{ and } h = h_1 \cup h_2 \\ & \text{and } s, h_1 \models p \text{ and } s, h_1 \models \text{emp} \text{ and } s, h_2 \models \text{true} \\ \Leftrightarrow & \exists h_1, h_2 \in Heaps : \text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset \text{ and } h = h_1 \cup h_2 \\ & \text{and } s, h_1 \models p \text{ and } h_1 = \emptyset \\ \Leftrightarrow & \exists h_2 \in Heaps : h = h_2 \text{ and } s, \emptyset \models p \\ \Leftrightarrow & s, \emptyset \models p . \end{aligned}$$

The last line shows that a pure assertion is independent of the heap and hence, in particular, has to be satisfied for the empty heap. Next we show the implication (5) \Rightarrow (6). Instantiating Equation (5) and using the above result immediately imply the claim:

$$\begin{aligned} & \forall h, h' \in Heaps : (s, h \models p \Leftrightarrow s, h' \models p) \\ \Rightarrow & \forall h \in Heaps : (s, h \models p \Leftrightarrow s, \emptyset \models p) \\ \Leftrightarrow & \forall h \in Heaps : (s, h \models p \Leftrightarrow s, h \models (p \wedge \text{emp}) * \text{true}) . \end{aligned}$$

For the converse direction, we take two instances of (6). Then, using again the above result, we get

$$\begin{aligned} & \forall h \in Heaps : (s, h \models p \Leftrightarrow s, h \models (p \wedge \text{emp}) * \text{true}) \\ & \text{and } \forall h' \in Heaps : (s, h' \models p \Leftrightarrow s, h' \models (p \wedge \text{emp}) * \text{true}) \\ \Rightarrow & \forall h, h' \in Heaps : (s, h \models p \Leftrightarrow s, h \models (p \wedge \text{emp}) * \text{true}) \\ & \text{and } s, h' \models p \Leftrightarrow s, h' \models (p \wedge \text{emp}) * \text{true}) \\ \Leftrightarrow & \forall h, h' \in Heaps : (s, h \models p \Leftrightarrow s, \emptyset \models p \text{ and } s, h' \models p \Leftrightarrow s, \emptyset \models p) \\ \Rightarrow & \forall h, h' \in Heaps : (s, h \models p \Leftrightarrow s, h' \models p) . \end{aligned}$$

□

The complexity of this proof in predicate-logic illustrates the advantage that is gained by passing to an algebraic treatment. Logic-based formulas (in particular in separation logic) can become long and complicated. Calculating at the abstract level of quantales often shorten the proofs. Moreover the abstraction paves the way to using first-order off-the-shelf theorem provers for verifying properties; whereas a first-order theorem prover for separation logic has yet to be developed and implemented (cf. Section 8).

To conclude the paragraph concerning pure elements we list a couple of properties which can be proved very easily by our algebraic approach.

Lemma 6.5 *Consider a Boolean quantale S , pure elements $p, q \in S$ and arbitrary elements $a, b \in S$. Then*

- (a) $p \cdot a = p \sqcap a \cdot \top$;
- (b) $(p \sqcap a) \cdot b = p \sqcap a \cdot b$;
- (c) $p \cdot q = p \sqcap q$; in particular $p \cdot p = p$.

Their corresponding counterparts in separation logic and the proofs can again be found in [4].

The following lemma shows a.o. that in the complete lattice of pure elements meet and join coincide with composition and sum, respectively.

Lemma 6.6 *Pure elements form a Boolean lattice, i.e., they are closed under $+$, \sqcap and $\bar{}$. In particular, $p \cdot \bar{p} = 0$.*

Proof. We only show the closure for $\bar{}$; the others are straightforward. Since p is pure we can instantiate the meet-distributivity axiom (3) for pure assertions using $q = \bar{p}$ and $r = \top$, i.e., $(\bar{p} \cdot \top) \sqcap p \leq (\bar{p} \sqcap p) \cdot (\top \sqcap p) = 0$. From this we obtain by Boolean algebra (shunting) Equation (2). In Boolean quantales, we have for arbitrary elements a, b, c

$$\begin{aligned} a \cdot b &= (c \sqcap a) \cdot b + (\bar{c} \sqcap a) \cdot b = (c \sqcap a) \cdot b + (\bar{c} \sqcap a) \cdot (c \sqcap b) + (\bar{c} \sqcap a) \cdot (\bar{c} \sqcap b) \\ &\leq c \cdot \top + \top \cdot c + (\bar{c} \sqcap a) \cdot (\bar{c} \sqcap b) . \end{aligned}$$

The first two equations simply use Boolean algebra and distributivity laws, the approximation in the second line is by isotony. Setting c as a pure element p we get immediately by Equation (2) and commutativity

$$a \cdot b \leq p \cdot \top + \top \cdot p + (\bar{p} \sqcap a) \cdot (\bar{p} \sqcap b) \leq p + (\bar{p} \sqcap a) \cdot (\bar{p} \sqcap b) .$$

The remaining claim then follows again by shunting. □

As far as we know these closure properties are new and were not shown in separation logic so far.

6.2 Intuitionistic Assertions

Let us now turn to intuitionistic assertions. Following [18], an assertion p is *intuitionistic* iff

$$\forall s \in \text{Stores}, \forall h, h' \in \text{Heaps} : (h \subseteq h' \text{ and } s, h \models p) \text{ implies } s, h' \models p . \quad (7)$$

This means for a heap that satisfies an intuitionistic assertion p that it can be extended by arbitrary cells and still satisfies p .

Similar calculations as in the proof of Theorem 6.1 yield the equivalence of Equation (7) and

$$\forall s \in \text{Stores}, \forall h, h' \in \text{Heaps} : (s, h \models p * \text{true} \Rightarrow s, h \models p) . \quad (8)$$

Lifting this to an abstract level motivates the following definition.

Definition 6.7 In an arbitrary Boolean quantale S an element i is called intuitionistic iff it satisfies

$$i \cdot \top \leq i . \quad (9)$$

Elements of the form $i \cdot \top$ are also called vectors or ideals.

Corollary 6.8 *Every pure element of a Boolean quantale is intuitionistic.*

As before we just give a couple of properties. The proofs are again straightforward at the algebraic level.

Lemma 6.9 *Consider a Boolean quantale S , intuitionistic elements $i, j \in S$ and arbitrary elements $a, b \in S$ Then*

- (a) $(i \sqcap 1) \cdot \top \leq i$;
- (b) $i \cdot a \leq i \sqcap (a \cdot \top)$;
- (c) $(i \sqcap a) \cdot b \leq i \sqcap (a \cdot b)$;
- (d) $i \cdot j \leq i \sqcap j$.

Using the quantale AS, it is easy to see that none of these inequations can be strengthened to an equation. In particular, unlike as for pure assertions, multiplication and meet need not coincide.

Example 6.10 Consider $i =_{df} j =_{df} \llbracket x \mapsto 1 * \text{true} \rrbracket = \llbracket x \mapsto 1 \rrbracket \cup \llbracket \text{true} \rrbracket$. By this definition it is obvious that i and j are intuitionistic. The definitions of Section 3 then immediately imply

$$\begin{aligned} i \cap j &= \llbracket x \mapsto 1 \rrbracket \cup \llbracket \text{true} \rrbracket \\ i \cup j &= \llbracket x \mapsto 1 \rrbracket \cup \llbracket \text{true} \rrbracket \cup \llbracket x \mapsto 1 \rrbracket \cup \llbracket \text{true} \rrbracket = \emptyset . \end{aligned}$$

The last step follows from Example 5.2. □

Other classes of assertions for separation logic are given in [18] and most of their algebraic counterparts in [4].

7 Commands

7.1 Syntax and Semantics

In this section we introduce the command constructs of separation logic. Syntactically, they are given by

$$\begin{aligned}
\text{comm} ::= & \text{var} := \text{exp} \mid \text{skip} \mid \text{comm}; \text{comm} \\
& \mid \text{if } \text{bexp} \text{ then } \text{comm} \text{ else } \text{comm} \mid \text{while } \text{bexp} \text{ do } \text{comm} \\
& \mid \text{newvar } \text{var} \text{ in } \text{comm} \mid \text{newvar } \text{var} := \text{exp} \text{ in } \text{comm} \\
& \mid \text{var} := \text{cons}(\text{exp}, \dots, \text{exp}) \\
& \mid \text{var} := [\text{exp}] \mid [\text{exp}] := \text{exp} \\
& \mid \text{dispose } \text{exp}
\end{aligned}$$

We skip explanations for the well-known first three lines.

In a given state s the command $v := \text{cons}$ allocates n cells with e_i^s as the contents of the i -th cell. The cells have to form an unused contiguous region somewhere on the heap; the concrete allocation process is chosen non-deterministically. The address of the first cell is stored in x while the rest of the cells can be addressed indirectly via start address.

A dereferencing assignment $v := [e]$ assumes that e is an *exp*-expression and the value e^s (corresponding to $*e$ in \mathbb{C}) is an allocated address on the heap, i.e., $e^s \in \text{dom}(h)$ for the current heap h . In particular, after execution, the value of x is the contents of a dereferenced heap cell.

Conversely, an execution of $[e_1] := e_2$ assigns the value of the expression on the right hand side to the cell whose address is the value of the left hand side.

Finally, the command $\text{dispose } e$ is used for deallocating the heap cell with address e^s . After execution the disposed cell is not valid anymore, i.e., dereferencing that cell would cause a fault in the program execution.

Formally we model commands as relations between states and set $\text{Cmds} =_{df} \mathcal{P}(\text{States} \times \text{States})$. With every command d we associate a relation $\llbracket d \rrbracket_c \in \text{Cmds}$; in particular, $\llbracket \text{skip} \rrbracket_c =_{df} I$ where I is the identity relation. Using standard relational composition we get $(\text{Cmds}, \subseteq, \emptyset, ;, I)$ as basic quantale structure. For all commands we assume that the free variables of all expressions are within the domains of the stores s involved.

To abbreviate the next definition for the new commands, we use a convention similar to that of the refinement calculus (e.g. [1]). We characterise relations by predicates linking the input states (s, h) and output states (s', h') . If P is such a predicate then $R \hat{=} P$ abbreviates the clause $(s, h) R (s', h') \Leftrightarrow_{df} P$.

$$\begin{aligned}
\llbracket [e_1] := e_2 \rrbracket_c & \hat{=} s' = s \wedge h' = (e_1^s, e_2^s) \mid h \wedge e_1^s \in \text{dom}(h) , \\
\llbracket v := [e] \rrbracket_c & \hat{=} s' = (v, h(e^s)) \mid s \wedge h' = h , \\
\llbracket \text{dispose } e \rrbracket_c & \hat{=} s' = s \wedge e^s \in \text{dom}(h) \wedge h' = h - \{(e^s, h(e^s))\} , \\
\llbracket v := \text{cons}(e_1, \dots, e_n) \rrbracket_c & \hat{=} \exists a \in \text{Addresses} : s' = (v, a) \mid s \wedge \\
& a, \dots, a + n - 1 \notin \text{dom}(h) \wedge h' = \{(a, e_1^s), \dots, (a + n - 1, e_n^s)\} \mid h .
\end{aligned}$$

The condition for `dispose` guarantees that the dereferenced heap cell is already allocated, since otherwise we would immediately get a fault in the program execution. The statement $a, \dots, a + n - 1 \notin \text{dom}(h)$ expresses that all these addresses have to be unallocated. More details can be found in [4].

7.2 Tests and Hoare Triples

Hoare triples $\{p\} c \{q\}$ with p and q as predicates about states and c as command are used to reason about partial correctness. To model the pre- and postconditions of such triples in a semiring and quantale setting one uses *tests*. These are elements p below the multiplicative identity 1 that have complements $\neg p$ relative to 1. In the command quantale they are given by the partial identity relations of the form $\hat{P} = \{(t, t) : t \in P\}$ for some set P of states. It is clear that these subidentities, sets of states and predicates characterising states are in one-to-one correspondence. Join and meet of two tests coincide with their union \cup and composition $;$, resp. The set P can be retrieved from \hat{P} as $P = \text{dom}(\hat{P})$. Employing tests, we have the following equivalences (e.g., [12]):

$$\{p\} c \{q\} \Leftrightarrow p \cdot c \cdot \neg q \leq 0 \Leftrightarrow p \cdot c \leq c \cdot q \Leftrightarrow p \cdot c = p \cdot c \cdot q .$$

To use this general approach to Hoare logic we simply need to embed the quantale of assertions from the previous section into the set of tests of the relational quantale by the above correspondence between sets of states and partial identity relations. The operation \cup on assertions is lifted to tests by

$$\hat{P} \uplus \hat{Q} =_{df} \widehat{P \cup Q} .$$

The *frame rule* [17] allows local reasoning and describes the interaction between the separating conjunction in pre- and postconditions for a command c . For assertions p, q and r and an arbitrary command c it reads

$$\frac{\{p\} c \{q\}}{\{p * r\} c \{q * r\}}$$

assuming that no free variable of r is modified by c . The premise ensures that starting the execution of c in a state satisfying p ends in a state satisfying q . Furthermore the conclusion says that extending the initial and final heaps consistently with disjoint heaps will not invalidate the triple in the premise. The additional heap cells remain unchanged, as long as no free variable of r is changed by c . The algebraic counterpart is

$$P ; C \subseteq C ; Q \Rightarrow (P \uplus R) ; C \subseteq C ; (Q \uplus R) ,$$

assuming that P, Q and R are partial identities and C is an arbitrary element of our relational command quantale. Furthermore we have to ensure that C does not modify any free variables of the assertion R .

Finally we discuss inference rules for commands. By the above abstractions these rules become simple consequences of the well established general relational view of commands. Due to lack of space, we only sketch the rule for *mutation*.

Theorem 7.1 *Let p be an arbitrary assertion and e, e' be exp-expressions. Then valid triples for mutation commands are*

$$\begin{aligned} \{(e \mapsto -)\} [e] := e' \{(e \mapsto e')\} , & \quad (\text{local}) \\ \{(e \mapsto -) * p\} [e] := e' \{(e \mapsto e') * p\} , & \quad (\text{global}) \\ \{(e \mapsto -) * ((e \mapsto e') -*p)\} [e] := e' \{p\} . & \quad (\text{backward reasoning}) \end{aligned}$$

The local mutation law ensures that the cell at address e has the value e' after $[e] := e'$ is executed; provided there is an allocated cell at e . In the relational semantics the precondition is satisfied by $e^s \in \text{dom}(h)$. The global mutation law follows from the local one as an instantiation of the frame rule. It implies that reasoning can be modularised. Conversely, the local rule can be derived from the global one by setting $p = \text{emp}$. The backward reasoning law is used to determine a precondition that ensures the postcondition p . The global and the backward reasoning rules are again interderivable (see [18, 4]).

Further inference rules, like for disposal, overwriting and non-overwriting allocation as well as lookups are presented and explained in [18]; the algebraic treatment is similar to the one of mutation and straightforward.

8 Conclusion and Outlook

We have presented an algebraic treatment of separation logic. For assertions we have introduced a model based on sets of states. By this, separating implication coincides with a residual and most of the inference rules of [18] are simple consequences of standard residual laws. For pure and intuitionistic assertions we have given algebraic axiomatisations. Next we have embedded the command language of separation logic into a relational algebraic structure. In particular, we have defined a relational semantics for the heap-dependent commands and lifted the set-based semantics of assertions to relations.

To underpin our approach we have algebraically verified one of the standard examples — an in-place list reversal algorithm. The details can be found in [4]. The term *in-place* means that there is no copying of whole structures, i.e., the reversal is done by simple pointer modifications.

So far we have not analysed situations where data structures share parts of their cells (cf. Figure 2). First steps towards an algebraic handling of such situations are given in [14, 6]. In future work, we will adapt these approaches for our algebra of separation logic.

Our algebraic approach to separation logic also paves the way to verifying-properties with off-the-shelf theorem provers. Boolean semirings and quantales have proved to be reasonably well suitable for automated theorem provers [9]. Hence one of the next plans for future work is to analyse the power of such systems for reasoning with separation logic. A long-term perspective is to incorporate reasoning about concurrent programs with shared linked data structures along the lines of [16].

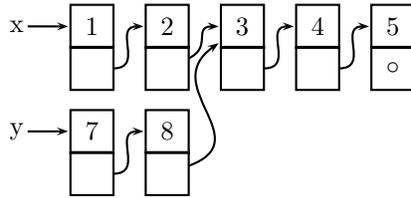


Fig. 2. Two lists with shared cells.

References

1. R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.
2. T. Blyth and M. Janowitz. *Residuation Theory*. Pergamon Press, 1972.
3. J. H. Conway. *Regular Algebra and Finite Machines*. Chapman & Hall, 1971.
4. H.-H. Dang. Algebraic aspects of separation logic. Technical Report 2009-01, Institut für Informatik., 2009.
5. E. Dijkstra. *A discipline of programming*. Prentice Hall, 1976.
6. T. Ehm. Pointer Kleene algebra. In R. Berghammer, B. Möller, and G. Struth, editors, *Relational and Kleene-Algebraic Methods in Computer Science*, volume 3051 of *LNCS*, pages 99–111. Springer, 2004.
7. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
8. P. Höfner and B. Möller. An algebra of hybrid systems. *J. Logic and Algebraic Programming*, 78:74–97, 2009.
9. P. Höfner and G. Struth. Automated reasoning in Kleene algebra. In F. Pfennig, editor, *Automated Deduction*, volume 4603 of *LNAI*, pages 279–294. Springer, 2007.
10. E. V. Huntington. Boolean algebra. A correction. *Trans. AMS*, 35:557–558, 1933.
11. E. V. Huntington. New sets of independent postulates for the algebra of logic. *Trans. AMS*, 35:274–304, 1933.
12. D. Kozen. On Hoare logic, Kleene algebra, and types. In P. Gärdenfors, J. Woleński, and K. Kijania-Placek, editors, *In the Scope of Logic, Methodology, and Philosophy of Science: Volume One of the 11th Int. Congress Logic, Methodology and Philosophy of Science, Cracow, August 1999*, volume 315 of *Studies in Epistemology, Logic, Methodology, and Philosophy of Science*, pages 119–133. Kluwer, 2002.
13. B. Möller. Towards pointer algebra. *Science of Computer Prog.*, 21(1):57–90, 1993.
14. B. Möller. Calculating with acyclic and cyclic lists. *Information Sciences*, 119(3-4):135–154, 1999.
15. B. Möller. Kleene getting lazy. *Science of Computer Prog.*, 65:195–214, 2007.
16. P. O’Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375:271–307, 2007.
17. P. W. O’Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL ’01: 15th International Workshop on Computer Science Logic*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.
18. J. C. Reynolds. An introduction to separation logic. Proceedings Marktobendorf Summer School 2008 (forthcoming).
19. K. Rosenthal. Quantales and their applications. *Pitman Research Notes in Mathematics Series*, 234, 1990.