

## Transitive separation logic

Han-Hing Dang, Bernhard Möller

### Angaben zur Veröffentlichung / Publication details:

Dang, Han-Hing, and Bernhard Möller. 2012. "Transitive separation logic."  
Lecture Notes in Computer Science 7560: 1-16.  
[https://doi.org/10.1007/978-3-642-33314-9\\_1](https://doi.org/10.1007/978-3-642-33314-9_1).

### Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under the following conditions:

**Deutsches Urheberrecht**

Weitere Informationen finden Sie unter: / For more information see:

<https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publizieren>



# Transitive Separation Logic

Han-Hing Dang and Bernhard Möller

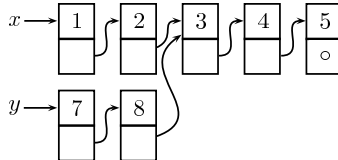
Institut für Informatik, Universität Augsburg, D-86159 Augsburg, Germany  
{h.dang,moeller}@informatik.uni-augsburg.de

**Abstract.** Separation logic (SL) is an extension of Hoare logic by operations and formulas that not only talk about program variables, but also about heap portions. Its general purpose is to enable more flexible reasoning about linked object/record structures. In the present paper we give an algebraic extension of SL at the data structure level. We define operations that additionally to heap separation make assumptions about the linking structure. Phenomena to be treated comprise reachability analysis, (absence of) sharing, cycle detection, preservation of substructures under destructive assignments. We demonstrate the practicality of this approach with the examples of in-place list-reversal and tree rotation.

**Keywords:** Separation logic, reachability, sharing, strong separation

## 1 Introduction

Separation logic (SL) as an extension of Hoare logic includes spatial operations and formulas that do not only talk about single program variables, but also about heap portions (*heaplets*). The original purpose of SL was to enable more flexible reasoning about linked object/record structures in a sequential version. This has also been extended to concurrent contexts [18]. The central connective of this logic is the *separating conjunction*  $P_1 * P_2$  of formulas  $P_1, P_2$ . It guarantees that the set of resources characterised by the  $P_i$  are disjoint. This allows under some circumstances e.g. that an assignment to a resource of  $P_1$  does not change any value of resources in  $P_2$ . Consider the following concrete example:



From the variables  $x$  and  $y$  two singly linked lists can be accessed. If we would run, e.g., an in-place list reversal algorithm on the list accessible from  $x$ , this would at the same time change the contents of the list accessible from  $y$ . This is because the lists show the phenomenon of *sharing*. Note that separating conjunction  $*$  in the above situation alone would only guarantee that the cells with contents  $1, \dots, 5$  have different addresses than the ones with contents  $7, 8$ .

The purpose of the present paper is to define in an abstract fashion connectives stronger than  $*$  that ensure the absence of sharing as depicted above. With this, we also hope to facilitate reachability analysis within SL as, e.g., needed in garbage collection algorithms, or the detection and exclusion of cycles to guarantee termination in such algorithms. Moreover, we provide a collection of predicates that characterise structural properties of linked structures and prove inference rules for them that express preservation of substructures under destructive assignments. Finally, we include abstraction functions into the program logic which allows very concise and readable reasoning. The approach is illustrated with two examples, namely in-situ list reversal and tree rotation.

## 2 Basics and Definitions

Following [4, 10] we define abstract separation algebras that are used to represent the abstract structure of the underlying considered sets of resources.

**Definition 2.1** A *separation algebra* is a partial commutative monoid  $(H, \bullet, u)$ . We call the elements of  $H$  *states*. The operation  $\bullet$  denotes state combination and the *empty state*  $u$  is its unit. A partial commutative monoid is given by a partial binary operation satisfying the unity, commutativity and associativity laws w.r.t. the equality that holds for two terms iff both are defined and equal or both are undefined. The induced *combinability* or *disjointness* relation  $\#$  is defined, for  $h_1, h_2 \in H$  by  $h_0 \# h_1 \Leftrightarrow_{df} h_0 \bullet h_1$  is defined.

As a concrete example we could instantiate states with heaplets, i.e., parts of the global heap in memory. Heaplets are modelled as partial functions  $H =_{df} \mathbb{N} \rightsquigarrow \mathbb{N}$ , i.e., from naturals to naturals for simplicity. The  $\bullet$  operation then corresponds to union of partial functions and  $u$  denotes the empty set or the everywhere undefined function. The original disjointness relation for heaplets reads  $h_0 \# h_1 \Leftrightarrow_{df} \text{dom } h_0 \cap \text{dom } h_1 = \emptyset$  for heaps  $h_0, h_1$ , where  $\text{dom } h$  is the domain of  $h$ . For more concrete examples we refer to [2].

**Definition 2.2** Let  $(H, \bullet, u)$  be a separation algebra. Predicates  $P, Q, \dots$  over  $H$  are elements of the powerset  $\mathcal{P}(H)$ . On predicates the separating conjunction is defined by pointwise lifting:

$$P * Q =_{df} \{h_1 \bullet h_2 : h_1 \in P, h_2 \in Q, h_1 \# h_2\}, \quad \text{emp} =_{df} \{u\}.$$

We now abstract from the above definition of heaplets by replacing them by elements of a *modal Kleene algebra* [6], our main algebraic structure. We will introduce its constituents in several steps. The basic layer is an *idempotent semiring*  $(S, +, \cdot, 0, 1)$ , where  $(S, +, 0)$  forms an idempotent commutative monoid and  $(S, \cdot, 1)$  a plain monoid.

A concrete example of an idempotent semiring is the set of relations. The natural order coincides with inclusion  $\subseteq$ , while  $+$  abstracts  $\cup$  and  $\cdot$  abstracts relational composition  $;$ . The element  $0$  represents the empty relation while  $1$  denotes the identity relation.

To express reachability in this algebra we need to represent sets of nodes. Relationally, this can be done using subsets of the identity relation. In general semirings this can be mimicked by sub-identity elements  $p \leq 1$ , called *tests* [14, 13]. These elements are requested to have a complement relative to 1, i.e., an element  $\neg p$  that satisfies  $p + \neg p = 1$  and  $p \cdot \neg p = 0 = \neg p \cdot p$ . Thus tests have to form a Boolean subalgebra. By this,  $+$  coincides with the binary supremum  $\sqcup$  and  $\cdot$  with binary infimum  $\sqcap$  on tests. Every semiring contains at least the greatest test 1 and the least test 0.

The product  $p \cdot a$  means restriction of the domain of a relation  $a$  to starting nodes in  $p$  while multiplication  $a \cdot p$  from the right restricts the range of  $a$ . By this, we can now axiomatise domain  $\lceil$  and codomain  $\rceil$  tests, following [6]. Note that, according to the general idea of tests, in the relation semiring these operations will yield sub-identity relations in one-to-one correspondence with the usual domain and range. For arbitrary element  $a$  and test  $p$  we have the axioms

$$\begin{aligned} a &\leq \lceil a \cdot a, & \lceil (p \cdot a) &\leq p, & \lceil (a \cdot b) &= \lceil (a \cdot \lceil b), \\ a &\leq a \cdot \rceil, & (a \cdot p) \rceil &\leq p, & (a \cdot b) \rceil &= (a \rceil \cdot b) \rceil. \end{aligned}$$

These imply additivity and isotony, among others, see [6]. Based on domain, we now define the *diamond* operation that plays a central role in our reachability analyses:  $\langle a | p =_{df} (p \cdot a) \rceil$ . Since this is an abstract version of the diamond operator from modal logic, an idempotent semiring with it is called *modal*.

The diamond  $\langle a | p$  calculates all immediate successor nodes under  $a$  starting from the set of nodes  $p$ , i.e., all nodes that are reachable within one  $a$ -step, aka the *image* of  $p$  under  $a$ . This operation distributes through union and is strict and isotone in both arguments.

Finally, to be able to calculate reachability within arbitrarily many steps, we extend the algebraic structure to a modal *Kleene algebra* [12] by an iteration operator  $*$ . It can be axiomatised by the following unfold and induction laws:

$$\begin{aligned} 1 + x \cdot x^* &\leq x^*, & x \cdot y + z &\leq y \Rightarrow x^* \cdot z &\leq y, \\ 1 + x^* \cdot x &\leq x^*, & y \cdot x + z &\leq y \Rightarrow z \cdot x^* &\leq y. \end{aligned}$$

This implies that  $a^*$  is the least fixed-point  $\mu_f$  of  $f(x) = 1 + a \cdot x$ .

Next, we define the reachability function *reach* by

$$reach(p, a) =_{df} \langle a^* | p.$$

Among other properties, *reach* distributes through  $+$  in its first argument and is isotone in both arguments. Moreover we have the *induction rule*

$$reach(p, a) \leq q \iff p \leq q \wedge \langle a | q \leq q.$$

### 3 A Stronger Notion of Separation

As we have described in Section 1 the standard separating conjunction  $*$  alone often does not describe sufficient disjointness needed for program verification. Simple sharing patterns in data structures can not be excluded from the only use of  $*$  as can be seen in the following examples: For addresses  $x_1, x_2, x_3$  with



$h_1$  and  $h_2$  satisfy the disjointness property since  $\overline{h_1} \cap \overline{h_2} = \emptyset$ . But still  $h = h_1 \cup h_2$  does not appear very separated from the viewpoint of reachable cells since in the left example both subheaps refer to the same address and in the right they form a simple cycle. This can be an undesired behaviour since acyclicity of the data structure is a main correctness property needed for many algorithms working, e.g., on linked lists or tree structures. In many cases the separation expressed by  $\overline{h_1} \cap \overline{h_2} = \emptyset$  is too weak. We want to find a stronger disjointness condition that takes such phenomena into account.

First, to simplify the description, for our new disjointness condition, we abstract from non-pointer attributes of objects, since they do not play a role for reachability questions. One can always view the non-pointer attributes of an object as combined with its address into a “super-address”. Therefore we give all definitions in the following only on the relevant part of a state that affects the reachability observations.

Moreover, in reachability analysis we are only interested in the existence of paths and not in path labels etc. Hence, multiple, differently labelled links from one object to another are projected into a single non-labelled link. Such a projection function on labels can, e.g., be found in [7], which gives an algebraic approach for representing labelled graphs, based on fuzzy relations.

With this abstraction, a linked object structure can be represented by an *access relation* between object addresses. Again, we pass to the more abstract algebraic view by using elements from a modal Kleene algebra to stand for concrete access relations; hence we call them *access elements*. In the following we will denote access elements by  $a, b, \dots$

Extending [8, 17] we give a stronger separation relation  $\oplus$  on access elements.

**Definition 3.1** For access elements  $a_1, a_2$ , we define the *strong disjointness relation*  $\oplus$  by setting  $a = a_1 + a_2$  in

$$a_1 \oplus a_2 \Leftrightarrow_{df} reach(\overline{a_1}, a) \cdot reach(\overline{a_2}, a) = 0.$$

Intuitively,  $a$  is strongly separated into  $a_1$  and  $a_2$  if each address reachable from  $a_1$  is unreachable from  $a_2$  w.r.t.  $a$ , and vice versa. Note that since all results of the *reach* operation are tests,  $\cdot$  coincides with their meet, i.e., intersection in the concrete algebra of relations.

This stronger condition rules out the above examples.

Clearly,  $\oplus$  is commutative. Moreover, since we have for all  $p, b$  that  $p \leq reach(p, b)$ , the new separation condition indeed implies the analogue of the old one, i.e., both parts are disjoint:  $a_1 \oplus a_2 \Rightarrow \overline{a_1} \cdot \overline{a_2} = 0$ . Finally,  $\oplus$  is downward closed by isotony of *reach*:  $a_1 \oplus a_2 \wedge b_1 \leq a_1 \wedge b_2 \leq a_2 \Rightarrow b_1 \oplus b_2$ .

It turns out that  $\oplus$  can be characterised in a much simpler way. To formulate it, we define an auxiliary notion.

**Definition 3.2** The *nodes*  $\overline{a}$  of an access element  $a$  are given by  $\overline{a} =_{df} \overline{a + a}$ .

From the definitions it is clear that  $\overline{a + b} = \overline{a} + \overline{b}$  and  $\overline{0} = 0$ .

We show two further properties that link the nodes operator with reachability.

**Lemma 3.3** For an access element  $a$  we have

1.  $\overline{a} \leq \text{reach}(\overline{a}, a)$ ,
2.  $\overline{a} \cdot \overline{b} = 0 \Rightarrow \text{reach}(\overline{a}, a + b) = \overline{a}$ ,
3.  $\overline{a} = \text{reach}(\overline{a}, a)$ .

*Proof.*

1. First,  $\overline{a} \leq \text{reach}(\overline{a}, a)$  by the reach induction rule from Section 2. Second, by a domain property,  $\overline{a} = (\overline{a} \cdot a) = \langle a | \overline{a} \leq \text{reach}(\overline{a}, a) \rangle$ .
2. For  $(\leq)$  we know by diamond star induction:

$$\text{reach}(\overline{a}, a + b) \leq \overline{a} \Leftrightarrow \overline{a} \leq \overline{a} \wedge \langle (a + b) | \overline{a} \leq \overline{a} \rangle.$$

$\overline{a} \leq \overline{a}$  holds by definition of  $\overline{\phantom{a}}$ , while  $\langle (a + b) | \overline{a} \leq \overline{a} \rangle$  resolves by diamond distributivity to  $\langle a | \overline{a} \leq \overline{a} \rangle \wedge \langle b | \overline{a} \leq \overline{a} \rangle$ . This is equivalent to  $(\overline{a} \cdot a) \leq \overline{a} \wedge (\overline{a} \cdot b \cdot b) \leq \overline{a}$  by definition and a property of domain. Finally, the claim holds by  $(\overline{a} \cdot a) \leq \overline{a}$  and  $\overline{a} \cdot \overline{b} = 0$  by the assumption  $\overline{a} \cdot \overline{b} = 0$ . The direction  $(\geq)$  follows from Part 1,  $a \leq a + b$  and isotony of  $\text{reach}$ .

3. This follows by setting  $b = 0$  in Part 2.  $\square$

Trivially, the first and last law state that all nodes in the domain and range of an access element  $a$  are reachable from  $\overline{a}$ , while the second law denotes a locality condition. If the domain as well as the range of a second access element  $b$  are disjoint from both components of  $a$  then  $b$  does not affect reachability via  $a$ . Using these theorems we can give a simpler equivalent characterisation of  $\oplus$ .

**Lemma 3.4**  $a \oplus b \Leftrightarrow \overline{a} \cdot \overline{b} = 0$ .

*Proof.*  $(\Rightarrow)$  From Lemma 3.3.1 and isotony of  $\text{reach}$  we infer  $\overline{a} \leq \text{reach}(\overline{a}, a) \leq \text{reach}(\overline{a}, a + b)$ . Likewise,  $\overline{b} \leq \text{reach}(\overline{b}, a + b)$ . Now the claim is immediate.

$(\Leftarrow)$  Lemma 3.3.2 tells us  $\text{reach}(\overline{a}, a + b) \cdot \text{reach}(\overline{b}, a + b) = \overline{a} \cdot \overline{b}$ , from which the claim is again immediate.  $\square$

**Corollary 3.5** For an access element  $a$  we always have  $0 \oplus a \Leftrightarrow a \oplus 0 \Leftrightarrow \text{true}$ .

By the use of Lemma 3.4, it is not difficult to derive the following result that will give us the possibility to characterise the interplay of the new separation operation with standard separating conjunction.

**Lemma 3.6** The relation  $\oplus$  is bilinear, i.e., it satisfies

$$(a + b) \oplus c \Leftrightarrow a \oplus c \wedge b \oplus c \quad \text{and} \quad a \oplus (b + c) \Leftrightarrow a \oplus b \wedge a \oplus c.$$

As in standard SL, strong separation can be lifted to predicates.

**Definition 3.7** For predicates  $P_1$  and  $P_2$ , we define the *strongly separating conjunction* by  $P_1 \circledast P_2 =_{df} \{a + b : a \in P_1, b \in P_2, a \oplus b\}$ .

Recall that  $\text{emp} = \{0\}$ . We have the following result.

**Lemma 3.8** Let  $S$  denote the set of predicates. Then  $(S, \circledast, 0)$  is a separation algebra, i.e.,  $\circledast$  is commutative and associative and  $P \circledast \text{emp} = \text{emp} \circledast P = P$ .

*Proof.* Commutativity is immediate from the definition. Neutrality of **emp** follows from Corollary 3.5 and by neutrality of 0 w.r.t. +.

For associativity, assume  $a \in (P_1 \circledast P_2) \circledast P_3$ , say  $a = a_{12} + a_3$  with  $a_{12} \circledast a_3$  and  $a_{12} \in P_1 \circledast P_2$  and  $a_3 \in P_3$ . Then there are  $a_1, a_2$  with  $a_1 \circledast a_2$  and  $a_{12} = a_1 + a_2$  and  $a_i \in P_i$ . From Lemma 3.4 we obtain therefore  $\overline{a_{12}} \cdot \overline{a_3} = 0 \wedge \overline{a_1} \cdot \overline{a_2} = 0$ . Moreover, by Lemma 3.6 the first conjunct is equivalent to  $\overline{a_1} \cdot \overline{a_3} = 0 \wedge \overline{a_2} \cdot \overline{a_3} = 0$ . Therefore we also have  $a \in P_1 \circledast (P_2 \circledast P_3)$ .

Hence we have shown that  $(P_1 \circledast P_2) \circledast P_3 \subseteq P_1 \circledast (P_2 \circledast P_3)$ . The reverse inequation follows analogously.  $\square$

## 4 Relating Strong Separation With Standard SL

A central question that may arise while reading this paper is: why does classical SL get along with the weaker notion of separation rather than the stronger one?

We will see that some aspects of our stronger notion of separation are in SL implicitly welded into recursive data type predicates. To explain this, we first concentrate on singly linked lists. In [19] the predicate  $list(x)$  states that the heaplet under consideration consists of the cells of a singly linked list with starting address  $x$ . Its validity in a heaplet  $h$  is defined by the following clauses:

$$\begin{aligned} h \models list(\text{nil}) &\Leftrightarrow_{df} h = \emptyset, \\ x \neq \text{nil} \Rightarrow (h \models list(x)) &\Leftrightarrow_{df} \exists y : h \models [x \mapsto y] * list(y). \end{aligned}$$

Hence  $h$  has to be an empty list when  $x = \text{nil}$ , and a list with least one cell at its beginning when  $x \neq \text{nil}$ , namely  $[x \mapsto y]$ .

First, note that using  $\circledast$  instead of  $*$  would not work, because the heaplets used are obviously not strongly separate: their heaplets are connected by forward pointers to their successor heaplets.

To make our model more realistic, we now define the concept of *closed* relations and a special element that represents the improper reference **nil**.

**Definition 4.1** A test  $p$  is called *atomic* iff  $p \neq 0$  and  $q \leq p \Rightarrow q = 0 \vee q = p$  for any other test  $q$ . We assume a special atomic test  $\square$  that characterises the **nil** object. Then an access element  $a$  is called *proper* iff  $\square \cdot a = 0$  and *closed* iff  $\overline{a} \leq \overline{a} + \square$ .

Proper access elements do not link from the pseudo-reference  $\square$  to another one. By closedness, there exist no dangling references in the access element  $a$ .

We summarise a few consequences of this.

**Corollary 4.2** *If  $a_1$  and  $a_2$  are proper/closed then  $a_1 + a_2$  is also proper/closed.*

**Lemma 4.3** *For an access element  $a$  the following properties are equivalent:*

1.  $a$  is proper,
2.  $\square \cdot a = 0$ ,
3.  $a = \neg \square \cdot a$ .

*Proof.* 1. implies 2. immediately by the definition of domain. To see that 2. implies 3. we calculate  $a = \square \cdot a + \neg \square \cdot a = \neg \square \cdot a$ . Finally, 3. implies 1. by  $\square \cdot \overline{a} = \square \cdot \overline{(\neg \square \cdot a)} \leq \square \cdot \neg \square = 0$  since  $\square$  is a test.  $\square$

**Lemma 4.4** *An access element  $a$  is closed iff  $a^\top - \lceil a \leq \square$ .*

*Proof.* As tests form a Boolean subalgebra we immediately conclude  $a^\top - \lceil a \leq \square$   
 $\Leftrightarrow a^\top \cdot \neg \lceil a \leq \square \Leftrightarrow a^\top \leq \lceil a + \square$ .  $\square$

To include the special test  $\square$  for later treatments we redefine the strong disjointness relation into a weaker version. Since  $\text{nil}$  is frequently used as a terminator reference in data structures, it should still be reachable.

**Definition 4.5** For access elements  $a, a_1, a_2$  and  $a = a_1 + a_2$ , we define the *stronger disjointness relation*  $\#$  w.r.t  $\square$  by

$$a_1 \# a_2 \Leftrightarrow_{df} \lceil a_1 \cdot \lceil a_2 \leq \square.$$

Lemma 3.8 is not affected by this redefinition, i.e.,  $\circledast$  is still commutative and associative when based on this new version of  $\#$ .

**Lemma 4.6** *For proper and closed  $a_1, a_2$  with  $\lceil a_1 \cdot \lceil a_2 = 0$  we have  $a_1 \# a_2$ .*

*Proof.* By distributivity and order theory we know

$$\lceil a_1 \cdot \lceil a_2 \leq \square \Leftrightarrow \lceil a_1 \cdot \lceil a_2 \leq \square \wedge \lceil a_1 \cdot a_2^\top \leq \square \wedge a_1^\top \cdot \lceil a_2 \leq \square \wedge a_1^\top \cdot a_2^\top \leq \square.$$

The first conjunct holds by the assumption and isotony. For the second and analogously for the third we calculate  $\lceil a_1 \cdot a_2^\top \leq \lceil a_1 \cdot (\lceil a_2 + \square) = \lceil a_1 \cdot \lceil a_2 + \lceil a_1 \cdot \square = 0 \leq \square$ . The last conjunct again reduces by distributivity and the assumptions to  $\square \cdot \square \leq \square$  which is trivial since  $\square$  is a test.  $\square$

Domain-disjointness of access elements is also ensured by the standard separating conjunction. Moreover, it can be shown by induction on the structure of the *list* predicate that all access elements characterised by its analogue are closed, so that the lemma applies. This is why for a large part of SL the standard disjointness property suffices.

## 5 An Algebra of Linked Structures

According to [20], generally recursive predicate definitions, such as the list predicate, are semantically not well defined in classical SL. Formally, their definitions require the inclusion of fixpoint operators and additional syntactic sugar. This often makes the used assertions more complicated; e.g., by expressing reachability via existentially quantified variables, formulas often become very complex. The direction we will take in the following is rather trying to hide such additional information by defining operations and predicates that implicitly include necessary correctness properties like the exclusion of sharing and reachability.

First, following precursor work in [16, 17, 7, 8], we give some definitions to describe the shape of linked object structures, in particular of tree-like ones. We start by a characterisation of acyclicity.

**Definition 5.1** Call an access element  $a$  *acyclic* iff for all atomic tests  $p \neq \square$  we have  $p \cdot \langle a^+ \mid p = 0$ , where  $a^+ = a \cdot a^*$ .

Concretely, for an access relation  $a$ , each entry  $(x, y)$  in  $a^+$  denotes the existence of a path from  $x$  to  $y$  within  $a$ . Atomicity is needed to represent a single node; the definition would not work for arbitrary sets of nodes.

A simpler characterisation can be given as follows.

**Lemma 5.2**  $a$  is acyclic iff for all atomic tests  $p \neq \square$  we have  $p \cdot a^+ \cdot p = 0$ .

*Proof.*  $p \cdot \langle a^+ | p = 0 \Leftrightarrow (p \cdot a^+)^{\neg} \cdot p = 0 \Leftrightarrow (p \cdot a^+ \cdot p)^{\neg} = 0 \Leftrightarrow p \cdot a^+ \cdot p = 0$ .  $\square$

Next, since certain access operations are deterministic, we need an algebraic characterisation of determinacy. We borrow it from [5]:

**Definition 5.3** An access element  $a$  is *deterministic* iff  $\forall p : \langle a | a \rangle p \leq p$ , where the dual diamond is defined by  $|a\rangle p = \lceil (a \cdot p)$ .

A relational characterisation of determinacy of  $a$  is  $a^{\smile} \cdot a \leq 1$ , where  $\smile$  is the converse operator. Since our basic structures are semirings, in which no general converse operation is available, we have to express the respective properties in another way. We have chosen to use the well established notion of modal operators. This way our algebra works also for other structures than relations. The roles of the expressions  $a^{\smile}$  and  $a$  are now played by  $\langle a |$  and  $|a\rangle$ , respectively.

Now we define our model of linked object structures.

**Definition 5.4** We assume a finite set  $L$  of *selector names*, left or right in binary trees, and a modal Kleene algebra  $S$ .

1. A *linked structure* is a family  $a = (a_l)_{l \in L}$  of proper and deterministic access elements  $a_l \in S$ . This reflects that access along each particular selector is deterministic. The overall access element associated with  $a$  is then  $\Sigma_{l \in L} a_l$ , by slight abuse of notation again denoted by  $a$ ; the context will disambiguate.
2.  $a$  is a *forest* if  $a$  is acyclic and has maximal in-degree 1. Algebraically this is expressed by the dual of the formula for determinacy, namely  $\forall p : |a\rangle \langle a | p \leq p$ . A forest is called a *tree* if  $\lceil \bar{a} = \langle a^* | r$  for some atomic test  $r$ ; in this case  $r$  is called the *root* of the tree and denoted by  $root(a)$ .

Note that  $\square$  is a tree, while  $0$  is not a tree, since it has no root. But at least,  $0$  is a forest. It can be shown that the root of a tree is uniquely defined, namely

$$root(a) = \begin{cases} \square & \text{if } a = \square \\ \lceil \bar{a} - \bar{a} & \text{otherwise} \end{cases}.$$

Singly linked lists arise as the special case where we have only one selector next. In this case we call a tree a *chain*.

We now want to define programming constructs and assertions that deal with linked structures. We start with expressions.

**Definition 5.5** A *store* is a mapping from program identifiers to atomic tests. A *state* is a pair  $\sigma = (s, a)$  consisting of a store  $s$  and an access element  $a$ . For an identifier  $i$  and a selector name  $l$ , the semantics of the expression  $i.l$  w.r.t. a state  $(s, a)$  with  $a$  being a linked structure is defined as

$$\llbracket i.l \rrbracket_{(s,a)} =_{df} \langle a_l | (s(i)).$$

Program *commands* are modelled as relations between states. The semantics of plain assignment is as usual: For identifier  $i$  and expression  $e$  we set

$$i := e \quad =_{df} \quad \{((s, a), (s[i \leftarrow p], a)) : p = \llbracket e \rrbracket_{(s, a)} \text{ is an atomic test} \} .$$

We will show below how to model assignments of the form  $i.l := e$ .

As already mentioned in Section 2, one can encode subsets or predicates as sub-identity relations. This way we can view state predicates  $S$  as commands of the form  $\{(\sigma, \sigma) : \sigma \in S\}$ . We will not distinguish predicates and their corresponding relations notationally. Following [13, 4] we encode Hoare triples with state predicates  $S, T$  and command  $C$  as

$$\{S\} C \{T\} \Leftrightarrow_{df} S ; C \subseteq C ; T \Leftrightarrow S ; C \subseteq U ; T,$$

where  $U$  is the universal relation on states.

To treat assignments  $i.l := e$ , we add another ingredient to our algebra. Its purpose is to describe updates of access elements by adding or changing links.

**Definition 5.6** Assuming atomic tests with  $p \cdot q = 0 \wedge p \cdot \square = 0$ , we define a *twig* by  $p \mapsto q =_{df} p \cdot \top \cdot q$  where  $\top$  denotes the greatest element of the algebra. The corresponding *update* is  $(p \mapsto q) | a =_{df} p \mapsto q + \neg p \cdot a$ .

Note, that by  $p, q \neq 0$  also  $p \mapsto q \neq 0$ . Intuitively, in  $(p \mapsto q) | a$ , the single node of  $p$  is connected to the single node in  $q$ , while  $a$  is restricted to links that start from  $\neg p$  only.

Now we can define the semantics of selective assignments.

**Definition 5.7** For identifiers  $i, j$  and selector name  $l$  we set

$$i.l := j \quad =_{df} \quad \{((s, a), ((s, s(i) \mapsto s(j)) | a)) : s(i) \neq \square, s(i) \leq \lceil a \rceil \} .$$

In general such an assignment does not preserve treeness. We provide sufficient conditions for that in the form of Hoare triples in the next section.

## 6 Expressing Structural Properties of Linked Structures

**Definition 6.1** The set *terms*( $a$ ) of *terminal nodes* of a tree  $a$  is  $terms(a) =_{df} \bar{a}^\top - \lceil a \rceil$ , while the set of *linkable nodes* of the tree is  $links(a) =_{df} terms(a) - \square$ . A binary tree  $b$  is *linkable* iff  $links(a) \neq 0$ .

Assuming the *Tarski rule*, i.e.,  $\forall a : a \neq 0 \Rightarrow \top \cdot a \cdot \top = \top$ , we can easily infer for a twig  $(p \mapsto q)^\top = q$  and  $\lceil (p \mapsto q) \rceil = p$ .

**Lemma 6.2**  $\overline{(p \mapsto q)} = p + q$  and  $terms(p \mapsto q) = q$  and  $root(p \mapsto q) = p$ .

*Proof.* The first result is trivial.  $terms(p \mapsto q) = (p \mapsto q)^\top \cdot \neg \lceil (p \mapsto q) \rceil = q \cdot \neg p = q$  since  $p \cdot q = 0 \Leftrightarrow q \leq \neg p$ . The proof for *root* is analogous.  $\square$

**Definition 6.3** For an atomic test  $p$  and a predicate  $P$  we define the subpredicate  $P(p)$  and its validity in a state  $(s, a)$  with a tree  $a$  by

$$P(p) =_{df} \{a : a \in P, root(a) = p\} , \quad (s, a) \models P(i) \Leftrightarrow a \in P(s(i)) .$$

By this we can refer to the root of an access element  $a$  in predicates. A main tool for expressing separateness and decomposability is the following.

**Definition 6.4** For linked structures  $a_1, a_2$  we define *directed combinability* by

$$a_1 \triangleright a_2 \Leftrightarrow_{df} \lceil a_1 \cdot \overline{a_2} \rceil = 0 \wedge a_1^\neg \cdot a_2^\neg \leq \square \wedge \text{root}(a_2) \leq \text{links}(a_1) + \square .$$

This relation guarantees domain disjointness and excludes occurrences of cycles, since  $\lceil a_1 \cdot \overline{a_2} \rceil = 0 \Leftrightarrow \lceil a_1 \cdot a_2 \rceil = 0 \wedge \lceil a_1 \cdot \text{terms}(a_2) \rceil = 0$ . Moreover, it excludes links from non-terminal nodes of  $a_1$  to non-root nodes of  $a_2$ . If  $a_1, a_2$  are trees, it ensures that  $a_1$  and  $a_2$  can be combined by identifying some non-nil terminal node of  $a_1$  with the root of  $a_2$ . Since  $a_1$  is a tree, that node is unique, i.e., cannot occur more than once in  $a_1$ .

Note that by Lemma 4.6 the second conjunct above can be dropped when both arguments are singly-linked lists. We summarise some useful consequences of Definition 6.4.

**Lemma 6.5** *If  $a$  is a tree then  $a \triangleright 0$  and  $a \triangleright \square$ . Moreover,  $\square \triangleright a \Rightarrow a = \square$ .*

**Lemma 6.6** *For trees  $a_1$  and  $a_2$ , assume  $a_1 \triangleright a_2$ . Then  $\text{terms}(a_1 + a_2) = (\text{terms}(a_1) - \text{root}(a_2)) + \text{terms}(a_2)$  and hence  $\text{links}(a_1 + a_2) = (\text{links}(a_1) - \text{root}(a_2)) + \text{links}(a_2)$ . Symmetrically, if  $a_1 \neq 0$  then  $\text{root}(a_1 + a_2) = \text{root}(a_1)$ .*

*Proof.* By domain distributivity and De Morgan's laws we get

$$\text{terms}(a_1 + a_2) = a_1^\neg \cdot \neg \lceil a_1 \cdot \neg a_2 + a_2^\neg \cdot \neg \lceil a_1 \cdot \neg a_2 \rceil = \text{terms}(a_1) \cdot \neg \lceil a_2 \rceil + \text{terms}(a_2) \cdot \neg \lceil a_1 \cdot \neg a_2 \rceil .$$

Since  $\text{terms}(a_2) \leq \overline{a_2}$  by definition and  $\overline{a_2} \leq \neg \lceil a_1 \rceil$  by the assumption  $a_1 \triangleright a_2$ , the right summand reduces to  $\text{terms}(a_2)$ . To bring the left one into the claimed form, we first assume  $a_2 \neq \square$  and calculate

$$\begin{aligned} \text{terms}(a_1) - \text{root}(a_2) &= \text{terms}(a_1) \cdot (\neg \lceil a_2 + a_2^\neg \rceil) \\ &= \text{terms}(a_1) \cdot \neg \lceil a_2 + \text{terms}(a_1) \cdot a_2^\neg \rceil = \text{terms}(a_1) \cdot \neg \lceil a_2 \rceil , \end{aligned}$$

since  $\text{terms}(a_1) \cdot a_2^\neg = a_1^\neg \cdot \neg \lceil a_1 \cdot a_2^\neg \rceil = 0$  by  $a_1 \triangleright a_2$ . The case  $a_2 = \square$  follows immediately.

For  $\text{root}$  we first assume  $a_1 \notin \{\square, 0\}$ , thus  $a_1 + a_2 \neq \square$ . Next, we calculate, symmetrically,  $\text{root}(a_1 + a_2) = \lceil a_1 \cdot \neg a_1^\neg \cdot \neg a_2^\neg \rceil + \lceil a_2 \cdot \neg a_1^\neg \cdot \neg a_2^\neg \rceil$ .

The first summand reduces to  $\lceil a_1 \cdot \neg a_1^\neg \rceil = \text{root}(a_1)$ , since  $a_1 \not\oplus a_2$  implies  $\lceil a_1 \cdot a_2^\neg \rceil = 0$ , i.e.,  $\lceil a_1 \rceil \leq \neg a_2^\neg$ . The second summand is, by definition, equal to  $\text{root}(a_2) \cdot \neg a_1^\neg$ . Since  $a_1 \triangleright a_2$  implies  $\text{root}(a_2) \leq \text{terms}(a_1)$  and hence  $\text{root}(a_2) \leq a_1^\neg$ , this summand reduces to 0. If  $a_1 = \square$  then also  $a_2 = \square$  by Lemma 6.5 and the claim follows.  $\square$

**Definition 6.7** We define the predicate  $\text{tree} =_{df} \{a : a \text{ is a tree}\}$ . For  $P_1, P_2 \subseteq \text{tree}$  we define *directed combinability*  $\otimes$  by

$$P_1 \otimes P_2 =_{df} \{a_1 + a_2 : a_i \in P_i, a_1 \triangleright a_2\} .$$

This allows, conversely, talking about decomposability: If  $a \in P_1 \otimes P_2$  then  $a$  can be split into two disjoint parts  $a_1, a_2$  such that  $a_1 \triangleright a_2$  holds.

## 7 Examples

Using the just defined operations and predicates we give two concrete examples namely in-situ list reversal and rotation of binary trees.

## 7.1 List Reversal

This example is mainly intended to show the basic ideas of our approach. The algorithm is well known, for variables  $i, j, k$ :

$$j := \square ; \text{ while } (i \neq \square) \text{ do } (k := i.\text{next} ; i.\text{next} := j ; j := i ; i := k) .$$

**Definition 7.1** We call a chain  $a$  a *cell* if  $\lceil a$  is an atomic test.

Note that by  $a = 0 \Leftrightarrow \lceil a = 0$ , cells are always non-empty. This will be important for some of the predicates defined below.

**Lemma 7.2** For a cell  $a$  we have  $\text{root}(a) = \lceil a$ , hence  $\neg \text{root}(a) \cdot a = 0$ .

*Proof.* By definition  $\text{root}(a) \leq \lceil a$  and  $\text{root}(a) \neq 0$ . Thus  $\text{root}(a) = \lceil a$ .  $\square$

**Lemma 7.3** Twigs  $p \mapsto q$  are cells.

*Proof.* By assumption,  $\lceil (p \mapsto q) = p$  is atomic. Moreover, by properness,  $\text{reach}(p, p \mapsto q) = \overline{p \mapsto q} = p + q$ , acyclicity holds by  $p \cdot q = 0$ . It remains to show determinacy: for arbitrary tests  $s$  we have  $q \cdot s \leq q \Rightarrow q \cdot s = 0 \vee q \cdot s = q \Leftrightarrow q \cdot s = 0 \vee q \leq s$ . Hence,  $\langle p \mapsto q \mid p \mapsto q \rangle s \leq \langle p \mapsto q \mid p \leq q \leq s \rangle$ .  $\square$

Now, we define predicates  $\text{LIST} =_{df} \{\text{list}, \text{l\_cell}\}$  for singly linked lists by

$$\begin{aligned} \text{list} &=_{df} \{a : a \text{ is an unlinkable chain } \}, \\ \text{l\_cell} &=_{df} \{a : a \text{ is a linkable cell } \}. \end{aligned}$$

**Lemma 7.4** For predicates in  $\text{LIST}$ , the operator  $\otimes$  is associative.

*Proof.* Assume  $a \in (P_1 \otimes P_2) \otimes P_3$  with  $a = a_{12} + a_3 \wedge a_{12} \triangleright a_3$  and  $a_{12} \in P_1 \otimes P_2 \wedge a_3 \in P_3$ . There exist  $a_1, a_2$  with  $a_1 \triangleright a_2$  and  $a_{12} = a_1 + a_2 \wedge a_i \in P_i$ .

From the definitions we first know  $\lceil a_1 \cdot \lceil a_2 = 0$  and  $\lceil (a_1 + a_2) \cdot \lceil a_3 = 0 \Leftrightarrow \lceil a_1 \cdot \lceil a_3 = 0 \wedge \lceil a_2 \cdot \lceil a_3 = 0$ . Hence, we can conclude  $\lceil a_1 \cdot \lceil (a_2 + a_3) = 0$ . Analogously,  $\lceil a_1 \cdot \lceil (a_2 + a_3) = 0$  and  $a_1 \cdot \lceil (a_2 + a_3) = 0$ .

Finally, we show  $\text{root}(a_3) \leq \text{links}(a_2)$  and hence  $a_2 + a_3 \in P_2 \otimes P_3$ : By the assumption  $a_1 \triangleright a_2$  and Lemma 6.6 we have  $\text{root}(a_3) \leq \text{links}(a_1 + a_2) = (\text{links}(a_1) - \text{root}(a_2)) + \text{links}(a_2)$ . Since  $a_1$  is a chain,  $\text{links}(a_1)$  is at most an atom, hence  $\text{links}(a_1) - \text{root}(a_2) = 0$  by  $a_1 \triangleright a_2$ , and we are done.

Moreover,  $\text{root}(a_2 + a_3) \leq \text{links}(a_1)$  also follows from Lemma 6.6 and therefore  $a \in P_1 \otimes (P_2 \otimes P_3)$ . The reverse inclusion is proved analogously.  $\square$

We give some further results used in the list reversal example.

**Lemma 7.5** If  $a$  is an unlinkable chain then  $\langle a^* \mid \text{root}(a) = \lceil a + \square$ .

**Lemma 7.6**  $\text{l\_cell} \otimes \text{list} \subseteq \text{list}$ .

*Proof.* Let  $a_1 \in \text{l\_cell}$  and  $a_2 \in \text{list}$  and assume  $a_1 \triangleright a_2$ . We need to show  $a_1 + a_2 \in \text{list}$ . Properness of  $a_1 + a_2$  follows from Corollary 4.2. Using distributivity of domain, we get  $\langle a_1 + a_2 \mid a_1 + a_2 \rangle p \leq p$  since  $\langle a_i \mid a_i \rangle p \leq p$  as  $a_1, a_2$  are deterministic and  $\langle a_2 \mid a_1 \rangle p \leq 0 \wedge \langle a_1 \mid a_2 \rangle p \leq 0$  by  $\lceil a_1 \cdot \lceil a_2 = 0$  which follows from the definition.

It remains to show  $\langle (a_1 + a_2)^* \mid \text{root}(a_1 + a_2) = \lceil a_1 + \lceil a_2 + \square \rceil \rceil$ . By definition we know  $\lceil a_1 \cdot a_2 \rceil = 0$ , hence  $a_2 \cdot a_1^* = a_2$ . Finally, using  $(a_1 + a_2)^* = a_1^* \cdot (a_2 \cdot a_1^*)^* = a_1^* \cdot a_2^*$  and Lemma 6.6 we have  $\langle (a_1 + a_2)^* \mid \text{root}(a_1 + a_2) = \langle a_1^* \cdot a_2^* \mid \text{root}(a_1) = \langle a_1^* \mid \text{root}(a_1) + \langle a_2 \cdot a_2^* \mid \text{root}(a_1) = \lceil a_1 + \text{links}(a_1) \rceil + \langle a_2 \cdot a_2^* \mid \lceil a_1 + \text{links}(a_1) \rceil \rangle$ . By atomicity, definitions and Lemma 7.5,  $\lceil a_1 + \text{root}(a_2) \rceil + \langle a_2 \cdot a_2^* \mid \text{root}(a_2) = \lceil a_1 + \langle a_2^* \mid \text{root}(a_2) = \lceil a_1 + \lceil a_2 + \square \rceil \rceil$ .  $\square$

**Lemma 7.7**  $p \neq \square \Rightarrow \text{list}(p) \subseteq \text{l\_cell}(p) \otimes \text{list}$  and  $\text{list}(\square) = \{\square\}$ .

**Definition 7.8** Assume a set  $L$  of selector names. For  $l \in L$ , an  $l$ -context is a linked structure  $a$  over  $L$  with  $a_l \in \text{l\_cell}$ . The corresponding predicate is  $\text{l\_context} =_{df} \{a : a \text{ is an } l\text{-context}\}$ .

**Lemma 7.9** For predicates  $Q, R$  and  $l \in L$  we have

$$\begin{aligned} \{ \text{l\_context}(i) \otimes Q \} \otimes R(j) \} \quad i.l := j \quad \{ \text{l\_context}(i) \otimes R(j) \} \otimes Q \}, \\ \{ Q \otimes \text{l\_context}(i) \} \otimes R(j) \} \quad i.l := j \quad \{ Q \otimes (\text{l\_context}(i) \otimes R(j)) \}, \\ \{ \text{l\_context}(i) \otimes Q \} \quad k := i.l \quad \{ \text{l\_context}(i) \otimes Q(j) \}. \end{aligned}$$

*Proof.* Consider a store  $s$  and  $a \in (\text{l\_context}(p) \otimes Q) \otimes R(q)$ . Thus,  $a = a_1 + a_2 + a_3$  with  $a_1 \in \text{l\_context} \wedge a_2 \in Q \wedge a_3 \in R$  and  $a_1 \# a_3 \wedge a_2 \# a_3 \wedge a_1 \triangleright a_2 \wedge s(i) = \text{root}(a_1) \wedge s(j) = \text{root}(a_3)$ . With  $a_1 \triangleright a_2$  we have  $\lceil a_1 \cdot \lceil a_2 \rceil = 0 \wedge \lceil a_1 \rceil \cdot \lceil a_2 \rceil \leq \square \wedge \text{root}(a_2) = \text{links}(a_1)$ .

We show  $(\text{root}(a_1) \mapsto \text{root}(a_3)) + \neg \text{root}(a_1) \cdot a \in (\text{l\_context}(p) \otimes R(q)) \otimes Q$ . First,  $a_1 \# a_3 \wedge a_1 \triangleright a_2$  implies  $\neg \text{root}(a_1) \cdot a_3 = a_3 \wedge \neg \text{root}(a_1) \cdot a_2 = a_2$  and  $\neg \text{root}(a_1) \cdot a_1 = 0$  by Lemma 7.2. Hence,  $\neg \text{root}(a_1) \cdot a = a_2 + a_3$ .

Further, we know  $\text{root}(a_1), \text{root}(a_3) \neq 0$  by definition. Assume  $a_3 \neq \square$ , thus  $\text{root}(a_3) \leq \lceil a$  and  $\text{root}(a_3) \neq \square$ . Then  $a_1 \# a_3 \Rightarrow \text{root}(a_1) \cdot \text{root}(a_3) = 0$ , Lemma 6.2 and assumptions imply  $\text{links}(\text{root}(a_1) \mapsto \text{root}(a_3)) = \text{root}(a_3)$ . Moreover,  $\lceil (\text{root}(a_1) \mapsto \text{root}(a_3)) \cdot \lceil a_3 \rceil = \text{root}(a_1) \cdot \lceil a_3 \rceil \leq \lceil a_1 \cdot \lceil a_3 \rceil = 0$  and  $(\text{root}(a_1) \mapsto \text{root}(a_3)) \cdot \lceil a_3 \rceil = \text{root}(a_3) \cdot \lceil a_3 \rceil \leq 0$ . Finally by Lemma 7.3 we have  $(\text{root}(a_1) \mapsto \text{root}(a_3)) + a_3 \in \text{l\_context}(p) \otimes R(q)$ .

It remains to show  $(\text{root}(a_1) \mapsto \text{root}(a_3)) \# a_2$  and  $a_3 \# a_2$ . The latter follows from commutativity of  $\#$  while the former resolves to  $\text{root}(a_1) \cdot \lceil a_2 \rceil = 0$  and  $\text{root}(a_3) \cdot \lceil a_2 \rceil = 0$  by Lemma 6.2. We calculate  $\lceil a_2 \rceil = \langle a_2^* \mid \text{root}(a_2) = \lceil a_2 + \square \rceil$  by Lemma 7.5. Hence,  $\text{root}(a_1) \cdot \lceil a_2 \rceil = \text{root}(a_1) \cdot (\lceil a_2 + \square \rceil) = 0$  by  $a_1 \triangleright a_2$  and  $a_1$  linkable. Similarly,  $\text{root}(a_3) \cdot \lceil a_2 \rceil \leq 0$  by assumptions.

If  $a_3 = \square$  then  $\text{root}(a_3) = \square$  and  $R = \{\square\}$ ; the proof for this case is analogous to the above one, except that  $\text{root}(a_3) \cdot \lceil a_2 \rceil \leq \square$  and  $\text{root}(a_3) \cdot a_3 \leq \square$ .

Proofs for the remaining triples can be given similarly.  $\square$

For proving functional correctness we introduce the concept of *abstraction functions* [9]. They are used, e.g., to state invariant properties.

**Definition 7.10** Assume  $a \in \text{list}$  and an atom  $p \in \lceil a \rceil$ . We define the abstraction function  $li_a$  w.r.t.  $a$  as well as the semantics of the expression  $i \mapsto$  for a program identifier  $i$  as follows:

$$li_a(p) =_{df} \begin{cases} \langle \rangle & \text{if } p \cdot \ulcorner a = 0, \\ \langle p \rangle \bullet li_a(\langle a \mid p \rangle) & \text{otherwise,} \end{cases} \quad \llbracket i^\rightarrow \rrbracket_{(s,a)} =_{df} li_a(s(i)).$$

Here  $\bullet$  stands for concatenation and  $\langle \rangle$  denotes the empty word.

Now using Lemma 7.9 and Hoare logic proof rules for variable assignment and while-loops, we can provide a full correctness proof of the in-situ list reversal algorithm. The invariant of the algorithm is defined by  $I \Leftrightarrow_{df} (j^\rightarrow)^\dagger \bullet i^\rightarrow = \alpha$  where  $\_^\dagger$  denotes word reversal. Note that  $(s, a) \models I \Leftrightarrow \llbracket (j^\rightarrow)^\dagger \bullet i^\rightarrow \rrbracket_{(s,a)} = s(\alpha)$  where  $\alpha$  represents a sequence.

```

{list (i)  $\wedge$   $i^\rightarrow = \alpha$ }
j :=  $\square$ ;
{(list (i)  $\otimes$  list (j))  $\wedge$  I}
while (i  $\neq$   $\square$ ) do (
  {(l_cell (i)  $\oplus$  list (i.next))  $\otimes$  list (j))  $\wedge$  I}
  k := i.next;
  {(l_cell (i)  $\oplus$  list (k))  $\otimes$  list (j)  $\wedge$  (j $^\rightarrow$ ) $^\dagger$   $\bullet$  i  $\bullet$  k $^\rightarrow = \alpha$ }
  {(l_cell (i)  $\oplus$  list (k))  $\otimes$  list (j)  $\wedge$  (i  $\bullet$  j $^\rightarrow$ ) $^\dagger$   $\bullet$  k $^\rightarrow = \alpha$ }
  i.next := j;
  {(list (i)  $\otimes$  list (k))  $\wedge$  (i $^\rightarrow$ ) $^\dagger$   $\bullet$  k $^\rightarrow = \alpha$ }
  j := i; i := k
  {(list (j)  $\otimes$  list (i))  $\wedge$  I}
)
{list (j)  $\wedge$  (j $^\rightarrow$ ) $^\dagger = \alpha$ }
{list (j)  $\wedge$  j $^\rightarrow = \alpha^\dagger$ }

```

Each assertion consists of a structural part and a part connecting the concrete and abstract levels of reasoning. The same pattern will occur in the tree rotation algorithm in the next subsection.

Compared to [19] we hide the existential quantifiers that were necessary there to describe the sharing relationships. Moreover, we include all correctness properties of the occurring data structures and their interrelationship in the definitions of the new connectives. Quantifiers to state functional correctness are not needed due to abstraction functions. Hence the formulas become easier to read and more concise.

To further underpin practicability of our approach, one could e.g. change the first two commands in the while loop of the list reversal algorithm (inspired by [3]), so that the algorithm could possibly leave a memory leak. Then after the assignment  $i.next := j$  we would get the postcondition  $(l\_cell (i) \oplus list (j)) \otimes list$ . This shows that the memory part characterised by  $list$  cannot be reached from  $i$  nor from  $j$  due to strong separation. Moreover, there is no program variable to the root reference of  $list$ .

## 7.2 Tree Rotation

To model binary trees we use the selector names `left` and `right`. A *binary tree* is then a tree  $(b_{left}, b_{right})$  over  $\{left, right\}$ . Setting  $b =_{df} b_{left} + b_{right}$ , we define an

abstraction function  $tr$  similar to the  $\rightarrow$  function above:

$$tr_b(p) =_{df} \begin{cases} \langle \rangle & \text{if } p \cdot \bar{b} = 0, \\ \langle tr_b(\langle b_{\text{left}} | p \rangle), p, tr_b(\langle b_{\text{right}} | p \rangle) \rangle & \text{otherwise,} \end{cases}$$

$$\llbracket i^{\leftrightarrow} \rrbracket_{(s,b)} =_{df} tr_b(s(i)).$$

As an example, we now present the correctness proof of an algorithm for tree rotation as known from the data structure of AVL trees. We first give a “clean” version, in which all occurring subtrees are separated. After that we will show an optimised version, where, however, sharing occurs in an intermediate state. The verification of that algorithm would take a few more steps, and hence we will not include it because of space restrictions.

For abbreviation we define two more predicates, for  $l \in \{\text{left}, \text{right}\}$ , by

$$l\_tree\_context =_{df} tree \cap l\_context.$$

Let  $T_l, T_k, T_r$  stand for trees and  $p, q$  denote atomic tests:

$$\begin{aligned} & \{ tree(i) \wedge i^{\leftrightarrow} = \langle T_l, p, \langle T_k, q, T_r \rangle \rangle \} \\ & j := i.\text{right}; \\ & \{ (right\_tree\_context(i)) \textcircled{\text{D}} tree(j) \wedge \\ & \quad i^{\leftrightarrow} = \langle T_l, p, \langle T_k, q, T_r \rangle \rangle \wedge j^{\leftrightarrow} = \langle T_k, q, T_r \rangle \} \\ & i.\text{right} := \square, \\ & \{ tree(i) \textcircled{\text{*}} tree(j) \wedge \\ & \quad i^{\leftrightarrow} = \langle T_l, p, \langle \rangle \rangle \wedge j^{\leftrightarrow} = \langle T_k, q, T_r \rangle \} \\ & k := j.\text{left}; \\ & \{ tree(i) \textcircled{\text{*}} ((left\_tree\_context(j)) \textcircled{\text{D}} tree(k)) \wedge \\ & \quad i^{\leftrightarrow} = \langle T_l, p, \langle \rangle \rangle \wedge j^{\leftrightarrow} = \langle T_k, q, T_r \rangle \wedge k^{\leftrightarrow} = T_k \} \\ & j.\text{left} := \square; \\ & \{ tree(i) \textcircled{\text{*}} tree(j) \textcircled{\text{*}} tree(k) \wedge \\ & \quad i^{\leftrightarrow} = \langle T_l, p, \langle \rangle \rangle \wedge j^{\leftrightarrow} = \langle \langle \rangle, q, T_r \rangle \wedge k^{\leftrightarrow} = T_k \} \\ & j.\text{left} := i; \\ & \{ (left\_tree\_context(j)) \textcircled{\text{D}} tree(i) \textcircled{\text{*}} tree(k) \wedge \\ & \quad i^{\leftrightarrow} = \langle T_l, p, \langle \rangle \rangle \wedge j^{\leftrightarrow} = \langle \langle T_l, p, \langle \rangle \rangle, q, T_r \rangle \wedge k^{\leftrightarrow} = T_k \} \\ & i.\text{right} := k; \\ & \{ left\_tree\_context(j) \textcircled{\text{D}} (right\_tree\_context(i) \textcircled{\text{D}} tree(k)) \wedge \\ & \quad j^{\leftrightarrow} = \langle \langle T_l, p, T_k \rangle, q, T_r \rangle \wedge i^{\leftrightarrow} = \langle T_l, p, T_k \rangle \wedge k^{\leftrightarrow} = T_k \} \end{aligned}$$

In particular,  $j$  now points to the rotated tree. The optimised version reads, without intermediate assertions, as follows:

$$\begin{aligned} & \{ i^{\leftrightarrow} = \langle T_l, p, \langle T_m, q, T_r \rangle \rangle \} \\ & j := i.\text{right}; \\ & i.\text{right} := j.\text{left}; \\ & j.\text{left} := i; \\ & \{ j^{\leftrightarrow} = \langle \langle T_l, p, \langle \rangle \rangle, q, T_r \rangle \} \end{aligned}$$

## 8 Related Work

There exist several approaches to extend SL by additional constructs to exclude sharing or restrict outgoing pointers of disjoint heaps to a single direction. Wang

et al. [22] defined an extension called *Confined Separation Logic* and provided a relational model for it. They defined various central operators to assert, e.g., that all outgoing references of a heap  $h_1$  point to another disjoint one  $h_2$  or all outgoing references of  $h_1$  either point to themselves or to  $h_2$ .

Our approach is more general due to its algebraicity and hence also able to express the mentioned operations. It is intended as a general foundation for defining further operations and predicates for reasoning about linked object structures.

Another calculus that follows a similar intention as our approach is given in [3]. Generally, there heaps are viewed as labelled object graphs. Starting from an abstract foundation the authors define a logic e.g. for lists with domain-specific predicates and operations which is feasible for automated reasoning.

By contrast, our approach enables abstract derivations in a fully first-order algebraic approach, called pointer Kleene algebra [7]. The given simple (in-)equational laws allow a direct usage of automated theorem proving systems as PROVER9 [15] or any other systems through the TPTP LIBRARY [21] at the level of the underlying separation algebra [11]. This supports and helpfully guides the development of domain specific predicates and operations. The assertions we have presented are suitable due to their simplicity for expressing shapes of linked list structures without the need of any arithmetic as in [3]. Part of these assertions can be automatically verified using SMALLFOOT [1].

## 9 Conclusion and Outlook

A general intention of the present work was relating the approach of pointer Kleene algebra with SL. The algebra has proved to be applicable for stating abstract reachability conditions and the derivation of such. Therefore, it can be used as an underlying separation algebra in SL. We defined extended operations similar to separating conjunction that additionally assert certain conditions on the references of linked object structures. As a concrete example we defined predicates and operations on linked lists and trees that enabled correctness proofs of an in-situ list-reversal algorithm and tree rotation.

As future work, it will be interesting to explore more complex or other linked object structures such as doubly-linked lists or threaded trees. In particular, more complex algorithms like the *Schorr-Waite Graph Marking* or concurrent garbage collection algorithms should be treated.

**Acknowledgements:** We thank all reviewers for their fruitful comments that helped to significantly improve the paper. This research was partially funded by the DFG project *MO 690/9-1 AlgSep — Algebraic Calculi for Separation Logic*.

## References

1. Berdine, J., Calcagno, C., O’Hearn, P.: A decidable fragment of separation logic. FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science pp. 110–117 (2005)

2. Calcagno, C., O'Hearn, P.W., Yang, H.: Local Action and Abstract Separation Logic. In: Proc. of the 22nd Symposium on Logic in Computer Science. pp. 366–378. IEEE Press (2007)
3. Chen, Y., Sanders, J.W.: Abstraction of Object Graphs in Program Verification. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) Proc. of 10th Intl. Conference on Mathematics of Program Construction. LNCS, vol. 6120, pp. 80–99. Springer (2010)
4. Dang, H.H., Höfner, P., Möller, B.: Algebraic separation logic. *Journal of Logic and Algebraic Programming* 80(6), 221–247 (2011)
5. Desharnais, J., Möller, B.: Characterizing determinacy in Kleene algebra. *Information Sciences* 139, 253–273 (2001)
6. Desharnais, J., Möller, B., Struth, G.: Kleene algebra with domain. *ACM Transactions on Computational Logic* 7(4), 798–833 (2006)
7. Ehm, T.: The Kleene algebra of nested pointer structures: Theory and applications, PhD Thesis (2003).  
[http://www.opus-bayern.de/uni-augsburg/frontdoor.php?source\\_opus=89](http://www.opus-bayern.de/uni-augsburg/frontdoor.php?source_opus=89)
8. Ehm, T.: Pointer Kleene algebra. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS/AKA 2003. LNCS, vol. 3051, pp. 99–111. Springer (2004)
9. Hoare, C.A.R.: Proofs of correctness of data representations. *Acta Informatica* 1, 271–281 (1972)
10. Hoare, C.A.R., Hussain, A., Möller, B., O'Hearn, P.W., Petersen, R.L., Struth, G.: On Locality and the Exchange Law for Concurrent Processes. In: Katoen, J.P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 250–264. Springer (2011)
11. Höfner, P., Struth, G.: Can refinement be automated? In: Boiten, E., Derrick, J., Smith, G. (eds.) Refine 2007. ENTCS, vol. 201, pp. 197–222. Elsevier (2008)
12. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation* 110(2), 366–390 (1994)
13. Kozen, D.: Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems* 19(3), 427–443 (1997)
14. Manes, E., Benson, D.: The inverse semigroup of a sum-ordered semiring. *Semigroup Forum* 31, 129–152 (1985)
15. McCune, W.W.: Prover9 and Mace4. <http://www.cs.unm.edu/~mccune/prover9>
16. Möller, B.: Some applications of pointer algebra. In: Broy, M. (ed.) *Programming and Mathematical Method*. pp. 123–155. No. 88 in NATO ASI Series, Series F: Computer and Systems Sciences, Springer (1992)
17. Möller, B.: Calculating with acyclic and cyclic lists. *Information Sciences* 119(3-4), 135–154 (1999)
18. O'Hearn, P.W.: Resources, Concurrency, and Local Reasoning. *Theoretical Computer Science* 375(1-3), 271–307 (2007)
19. Reynolds, J.C.: An introduction to separation logic. In: Broy, M. (ed.) *Engineering Methods and Tools for Software Safety and Security*. pp. 285–310. IOS Press (2009)
20. Sims, E.J.: Extending Separation Logic with Fixpoints and Postponed Substitution. *Theoretical Computer Science* 351(2), 258–275 (2006)
21. Sutcliffe, G., Suttner, C.: The TPTP problem library: CNF release v1.2.1. *Journal of Automated Reasoning* 21(2), 177–203 (1998)
22. Wang, S., Barbosa, L.S., Oliveira, J.N.: A Relational Model for Confined Separation Logic. In: Proc. of the 2nd IFIP/IEEE Intl. Symposium on Theoretical Aspects of Software Engineering. pp. 263–270. TASE '08, IEEE Press (2008)