Extended transitive separation logic

Han Hing Dang, Bernhard Möller

Angaben zur Veröffentlichung / Publication details:

Dang, Han Hing, and Bernhard Möller. 2015. "Extended transitive separation logic." *Journal of Logical and Algebraic Methods in Programming* 84 (3): 303–25. https://doi.org/10.1016/j.jlamp.2014.12.002.

Extended Transitive Separation Logic

H.-H. Dang[a], B. Möller[a]

[a]*Institut für Informatik, Universität Augsburg, D-86135 Augsburg, Germany*

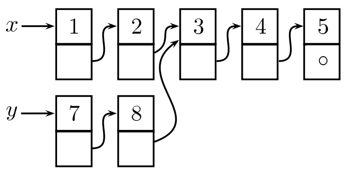
Abstract

Separation logic (SL) is an extension of Hoare logic by operations and formulas to reason more flexibly about heap portions or, more concretely, 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 guaranteeing heap separation, make assumptions about the linking structure. Phenomena to be treated comprise reachability analysis, (absence of) sharing, cycle detection and preservation of substructures under destructive assignments. We demonstrate the practicality of this approach with examples of in-place list-reversal, tree rotation and threaded trees.

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

1. Introduction

Separation logic (SL) is an extension of Hoare logic includes operations and formulas to reason more flexibly about heap portions (*heaplets*) or, more concretely, about linked object/record structures. The central connective of this logic is the *separating conjunction* $P_1 * P_2$ of formulas P_1, P_2. It guarantees that the the addresses of the resources mentioned by the P_i are disjoint. Hence, a simple assignment like `x = y` to a resource `x` of P_1 does not change any value of resources in P_2. By this, one gets a compositional approach to reasoning about programs. However, the situation becomes more complex, e.g., when considering a dereferencing of `x` like in `*x = y`. For a concrete example consider



Clearly, from the variables x and y two singly linked lists can be accessed. Now, let P_1 mention the starting addresses of the list records with contents $1, \dots 5$ and P_2 those of the records with contents $7, 8$. Note that $P_1 * P_2$ holds, since separating conjunction only guarantees that these address sets are disjoint, but not the *contents* of the memory cells of the records. Running, e.g., an in-place list reversal algorithm on the list accessible from x would at the same time unintendedly change the contents of the list accessible from y, because the lists show the phenomenon of *sharing*.

The purpose of the present paper is to define in an abstract fashion connectives stronger than $*$ that ensure the absence of sharing for situations as depicted above or that restrict sharing in a way that the absence of unintended changes can be ensured. With this, we hope to facilitate reachability analysis within SL as, e.g., needed in garbage collection algorithms, or the detection and exclusion of cycles to guarantee

Email addresses: `h.dang@informatik.uni-augsburg.de` (H.-H. Dang), `moeller@informatik.uni-augsburg.de` (B. Möller)
This paper is a significantly revised and extended version of [1].

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 selective assignments. Finally, we include abstraction functions into the program logic which allows very concise and readable reasoning. The approach is illustrated with examples as in-situ list reversal, tree rotation and a treatment of overlaid data structures as threaded trees.

2. Basics and Definitions

The basic algebraic structure we start from is that of a *modal Kleene algebra* [2], since it allows simple proofs in a calculational style and has proved to enable a suitable abstraction for pointer structures [3]. It further allows the application of first-order theorem provers [4] and captures a lot of models as relations, regular languages or finite traces. 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 monoid. An intuitive example of an idempotent semiring is provided by taking S to be the set of binary relations over some set X, with relational union as $+$, relational composition as $\cdot$, the empty relation as 0 and the identity relation $\{(x, x) : x \in X\}$ as 1.

The operation $+$ induces the *natural order* given by $x \leq y \Leftrightarrow_{df} x+y = y$. In the relational interpretation, $\leq$ coincides with inclusion $\subseteq$.

When the elements of the set X are interpreted as nodes in a linked data structure, such as records in a linked list, subsets of the identity relation can be used as an adequate representation for sets of nodes in X. In general semirings, this approach is mimicked by using sub-identity elements $p \leq 1$, called *tests* [5, 6]. Each of these elements is 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. This implies that $+$ coincides with the binary supremum $\sqcup$ and $\cdot$ with the binary infimum $\sqcap$ on tests. Every semiring contains at least the greatest test 1 and the least test 0.

When using tests, the abstract product $p \cdot a$ can be used to restrict an element a to links that start in nodes from p while, symmetrically, $a \cdot p$ restricts a to links ending in nodes from p. Following [2], this behaviour is used to axiomatise the operators $\ulcorner\_$ and $\_\urcorner$ that represent the domain and codomain of a semiring element as tests, 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. Abstractly, for arbitrary element a and test p we have the axioms

$$a \leq \ulcorner a \cdot a \ , \quad \ulcorner(p \cdot a) \leq p \ , \quad \ulcorner(a \cdot b) \ = \ \ulcorner(a \cdot \ulcorner b) \ , \qquad a \leq a \cdot a\urcorner \ , \quad (a \cdot p)\urcorner \leq p \ , \quad (a \cdot b)\urcorner \ = \ (a\urcorner \cdot b)\urcorner \ .$$

These imply fundamental properties such as additivity and isotony, among others, see [2].

Using these notions we can now define the *diamond* operation that plays a central role in our reachability analyses:

$$\langle a|\, p \ =_{df} \ (p \cdot a)\urcorner \ .$$

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 calculate reachability via arbitrarily many links in a data structure, we extend the algebraic structure to a modal *Kleene algebra* [7] by an iteration operator *. It can be axiomatised by the following unfold and induction laws:

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

This implies that a^* is the least fixed-point μ_f of $f(x) = 1 + a \cdot x$. Next, we define the reachability function:

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

Among other properties, *reach* distributes through $+$ in its first argument and is isotone in both arguments. Moreover we have the *induction rule* $p \leq q \wedge \langle a|\, q \leq q \;\Rightarrow\; reach(p,a) \leq q$.

The last ingredient needed to treat pointer structures is a special element within the algebra that represents the improper reference nil. Relationally, we can express it as the singleton relation $\square =_{df} \{(\intercal, \intercal)\}$, where $\intercal$ is a distinguished element of the set of nodes that represents nil or `null`.

Singleton sub-identity relations can abstractly be defined as *atomic* tests p. We call a test p *atomic* iff $p \neq 0$ and $q \leq p \Rightarrow q = 0 \vee q = p$ for arbitrary test q. In particular, we assume $\square$ to be an atomic test.

Using $\square$ we also characterise the subset of elements that have no links emanating from the pseudo-reference $\square$ to any other address $\neq \square$. This is a natural requirement, since the general purpose of $\square$ is to denote a terminator reference. We refer to this property as *properness*. Formally, an element a is called *proper* iff $\square \cdot a \leq \square$. We summarise a few consequences.

Corollary 2.1. *If a_1, a_2 are proper then also $a_1 + a_2$ is proper.*

Lemma 2.2. *For an access element a with $\square \cdot \ulcorner a = 0$ 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 $\square \cdot \ulcorner a = \square \cdot \ulcorner(\neg\square \cdot a) \leq \square \cdot \neg\square = 0$, since $\square$ is a test. $\square$

3. A Stronger Notion of Separation

Following the example given in Section 1, we now continue to define an adequate operation that excludes arbitrary sharing. We start by another simple sharing pattern in data structures that cannot be excluded from the only use of $*$ as can be seen in the following example.

Figure 1: Sharing examples for addresses x_1, x_2, x_3

h_1 and h_2 satisfy the disjointness property , since $\ulcorner h_1 \cap \ulcorner h_2 = \emptyset$. 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.

Hence, in many cases the separation expressed by $\ulcorner h_1 \cap \ulcorner h_2 = \emptyset$ 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.

With this abstraction, a linked object structure can be represented by an *access relation* between object addresses which we call *nodes* in the sequel. 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 $a, b, \ldots$. In this view, nodes are represented by atomic tests.

Extending [3, 8] we give a stronger separation relation $\#\!\!\!\bigcirc$ on access elements.

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

$$a_1 \#\!\!\!\bigcirc a_2 \Leftrightarrow_{df} reach(\ulcorner a_1, a) \cdot reach(\ulcorner a_2, a) \leq \square \, .$$

Intuitively, a is strongly separated into a_1 and a_2 if each address except $\square$ reachable from a_1 is unreachable from a_2 w.r.t. a, and vice versa. However, since $\square$ or, more concrete nil, is frequently used as a terminator reference in data structures, it should still be allowed to be reachable. 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.

The condition of strong disjointness rules out the data structures in Figure 1.

Clearly, $\circledast$ is commutative, $0 \circledast a$ and $\square \circledast a$. Moreover, since by definition we have for all p, b that $p \leq \mathit{reach}(p, b)$, the new separation condition indeed implies the analogue of the old one, i.e., both parts are disjoint: $a_1 \circledast a_2 \;\Rightarrow\; \ulcorner a_1 \cdot \ulcorner a_2 = 0$.

Finally, $\circledast$ is downward closed by isotony of *reach*: $a_1 \circledast a_2 \,\wedge\, b_1 \leq a_1 \,\wedge\, b_2 \leq a_2 \;\Rightarrow\; b_1 \circledast b_2$.

It turns out that $\circledast$ 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 $\overline{a} =_{df} \ulcorner a + a\urcorner$. A node in $\overline{a} - \ulcorner a$ is called *terminal* in a, since it has no link to other nodes.

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. $\overline{a} \leq \mathit{reach}(\ulcorner a, a)$,
2. $\langle b|\, \overline{a} \leq \overline{a} \;\Rightarrow\; \mathit{reach}(\ulcorner a, a+b) = \overline{a}$ *and hence* $\overline{a} = \mathit{reach}(\ulcorner a, a)$.

Trivially, the first law states that all nodes in the domain and range of an access element a are reachable from $\ulcorner a$, while the second law denotes a locality condition. If the b successors of all nodes of a are again at most a node of a then b does not affect reachability via a. Using these theorems we can give a simpler equivalent characterisation of $\circledast$.

Lemma 3.4. *If a, b are proper then* $a \circledast b \;\Leftrightarrow\; \overline{a} \cdot \overline{b} \leq \square$.

Proof. $(\Rightarrow)$ From Lemma 3.3.1 and isotony of *reach* we infer $\overline{a} \leq \mathit{reach}(\ulcorner a, a) \leq \mathit{reach}(\ulcorner a, a+b)$. Likewise, $\overline{b} \leq \mathit{reach}(\ulcorner b, a+b)$. Now the claim is immediate.
$(\Leftarrow)$ $\overline{a} \cdot \overline{b} \leq \square$ implies $\overline{a} \cdot \ulcorner b \leq \square$. Hence $\langle b|\, \overline{a} = (\overline{a} \cdot \overline{a} \cdot \ulcorner b \cdot b)\urcorner \leq (\overline{a} \cdot \square \cdot b)\urcorner \leq (\overline{a} \cdot \square)\urcorner \leq \overline{a}$, since b is proper and $\overline{a}, \square$ are tests. Symmetrically $\langle a|\, \overline{b} \leq \overline{b}$ holds. Now, Lemma 3.3.2 tells us $\mathit{reach}(\ulcorner a, a+b) \cdot \mathit{reach}(\ulcorner b, a+b) = \overline{a} \cdot \overline{b}$, from which the claim is again immediate. $\square$

The use of the condition in Lemma 3.4 instead of that in Definition 3.1 will considerably simplify the proofs to follow, since the Kleene * induction and unfold laws are no longer needed. Moreover, we can stay within the setting of a modal idempotent semiring using $\overline{\_}$. The assumption of proper access elements is not severe, since properness is a fundamental property of pointer structures.

Lemma 3.5. *On proper access elements the relation $\circledast$ is bilinear, i.e., satisfies*

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

Proof. We use the characerisation of $\circledast$ from Lemma 3.4. First, we calculate $(a+b) \circledast c \;\Leftrightarrow\; \overline{a+b} \cdot \overline{c} \leq \square \;\Leftrightarrow\; (\overline{a} + \overline{b}) \cdot \overline{c} \leq \square \;\Leftrightarrow\; \overline{a} \cdot \overline{c} \leq \square \,\wedge\, \overline{b} \cdot \overline{c} \leq \square \;\Leftrightarrow\; a \circledast c \,\wedge\, b \circledast c$. The other equivalence follows from commutativity of $\circledast$. $\square$

This result implies several standard laws that are crucial for calculations at the level of predicates. In particular, it enables a characterisation of the interplay between the new strong separation operation and the standard separating conjunction.

Similar as in standard SL, the strong separation relation can be lifted to predicates.

Definition 3.6. For predicates P_1 and P_2, we define the *separating conjunction* $*$ and the *strongly separating conjunction* $\circledast$ by

$$\begin{array}{rcl} P_1 * P_2 & =_{df} & \{a+b : a \in P_1\,, b \in P_2\,, \ulcorner a \cdot \ulcorner b \leq 0\}\ , \\ P_1 \circledast P_2 & =_{df} & \{a+b : a \in P_1\,, b \in P_2\,, a \,\#\, b\}\ . \end{array}$$

Moreover, we call a predicate *proper* if all its elements are proper.

Lemma 3.7. *$\circledast$ is commutative and associative. Moreover, $P \circledast \mathsf{emp} = P$ where $\mathsf{emp} =_{df} \{0\}$.*

Proof. Commutativity is immediate from the definition. Neutrality of emp follows from $0 \,\#\, a$ 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 $a_{12} \,\#\, a_3$ and $a_{12} \in P_1 \circledast P_2$ and $a_3 \in P_3$. Then there are a_1, a_2 with $a_1 \,\#\, a_2$ and $a_{12} = a_1 + a_2$ and $a_i \in P_i$. By Lemma 3.5 $a_{12} \,\#\, a_3$ is equivalent to $a_1 \,\#\, a_3 \wedge a_2 \,\#\, a_3$. Using Lemma 3.5 again $a_1 \,\#\, a_2 \wedge a_1 \,\#\, a_3 \Leftrightarrow a_1 \,\#\, a_{23}$ where $a_{23} = a_2 + a_3$. Therefore $a \in P_1 \circledast (P_2 \circledast P_3)$. Hence $(P_1 \circledast P_2) \circledast P_3 = P_1 \circledast (P_2 \circledast P_3)$. □

The defined connectives are structurally similar to operations given in [9]. Although that paper presented them with another application, they still can be interpreted for our applications due to abstractness. We present some of their properties and use them to characterise the interplay between separating conjunction and our stronger connective.

Lemma 3.8 (Exchange [9]). *Assume a semigroup $(A, +)$. Then for bilinear relations R and S with $R \subseteq S$ we have*

$$\begin{array}{rcl} P_1 Ⓡ P_2 & \subseteq & P_1 Ⓢ P_2\,, \\ (P_1 Ⓢ P_2) Ⓡ P_3 & \subseteq & P_1 Ⓢ (P_2 Ⓡ P_3)\,, \\ (P_1 Ⓢ P_2) Ⓡ (P_3 Ⓢ P_4) & \subseteq & (P_1 Ⓡ P_3) Ⓢ (P_2 Ⓡ P_4)\,, \end{array}$$

with $P_i \subseteq A$ and $P Ⓡ Q =_{df} \{a+b : a \in P\,, b \in Q\,, a\,R\,b\}$.

Since $\#$ and the standard domain disjointness condition are bilinear and $a_1 \,\#\, a_2 \Rightarrow \ulcorner a_1 \cdot \ulcorner a_2 = 0$ as mentioned above, results from [9] immediately yield:

Corollary 3.9. *For proper predicates P_i the following inequations hold:*

$$\begin{array}{rcl} P_1 \circledast P_2 & \subseteq & P_1 * P_2\,, \\ (P_1 * P_2) \circledast P_3 & \subseteq & P_1 * (P_2 \circledast P_3)\,, \\ P_1 \circledast (P_2 * P_3) & \subseteq & (P_1 \circledast P_2) * P_3\,, \\ (P_1 * P_2) \circledast (P_3 * P_4) & \subseteq & (P_1 \circledast P_3) * (P_2 \circledast P_4)\,. \end{array}$$

4. A Brief Excursion: 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 concentrate on singly linked lists. In [10] 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:

$$\begin{array}{l} h \models \mathit{list}(\mathsf{nil}) \;\Leftrightarrow_{df}\; h = \emptyset\,, \\ x \neq \mathsf{nil} \Rightarrow (h \models \mathit{list}(x) \Leftrightarrow_{df} \exists\, y : h \models [x \mapsto y] * \mathit{list}(y))\,. \end{array}$$

For simplicity, we omit the store component of the original definition that records the values of the program variables. Hence h has to be an empty heap when $x = \mathsf{nil}$, and a heap with at least one cell at its beginning when $x \neq \mathsf{nil}$, namely $[x \mapsto y]$.

First, note that using $\circledast$ instead of $*$ would not work, because the heaplets used are obviously not strongly separate: their cells are connected by forward pointers to their successor cells. In the next section we introduce an approach to represent such a connection within our algebra.

To understand the relationship of strong separation and the standard separation condition we now define the concept of *closedness*.

Definition 4.1. An access element a is called *closed* iff $a^{\urcorner} \leq {}^{\ulcorner}a + \square$.

In a closed element a there exist no dangling references. As an example, the above defined lists are closed as they are terminated by the value nil which abstractly corresponds to the element $\square$.

We summarise a few consequences of Definition 4.1.

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

Lemma 4.3. *An access element a is closed iff $a^{\urcorner} - {}^{\ulcorner}a \leq \square$.*

Proof. As tests form a Boolean subalgebra we conclude $a^{\urcorner} - {}^{\ulcorner}a \leq \square \;\Leftrightarrow\; a^{\urcorner} \cdot \neg{}^{\ulcorner}a \leq \square \;\Leftrightarrow\; a^{\urcorner} \leq {}^{\ulcorner}a + \square$. ☐

Lemma 4.4. *For proper and closed a_1, a_2 with ${}^{\ulcorner}a_1 \cdot {}^{\ulcorner}a_2 = 0$ we have $a_1 \circledast a_2$.*

Proof. By distributivity and order theory we know

$$\ulcorner\!\overline{a_1}\!\urcorner \cdot \ulcorner\!\overline{a_2}\!\urcorner \leq \square \;\Leftrightarrow\; {}^{\ulcorner}a_1 \cdot {}^{\ulcorner}a_2 \leq \square \wedge {}^{\ulcorner}a_1 \cdot a_2^{\urcorner} \leq \square \wedge a_1^{\urcorner} \cdot {}^{\ulcorner}a_2 \leq \square \wedge a_1^{\urcorner} \cdot a_2^{\urcorner} \leq \square.$$

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

Domain-disjointness of access elements is ensured by the standard separating conjunction. 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 [11], 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. To overcome this deficiency we provide operators and predicates that implicitly include such additional information, i.e., necessary correctness properties like the exclusion of sharing and reachability.

In what follows we extend our algebra following precursor work in [3, 12, 8, 13] and 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 $p \cdot \langle a^{+} |\, p = 0$, where $a^{+} = a \cdot a^{*}$.

For a concrete example one can think of 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.The element $\square$ is excluded, since it is used as a terminator reference and no structural properties are needed for it.

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. $p \cdot \langle a^+ | \, p = 0 \Leftrightarrow (p \cdot a^+)^{\urcorner} \cdot p = 0 \Leftrightarrow (p \cdot a^+ \cdot p)^{\urcorner} = 0 \Leftrightarrow p \cdot a^+ \cdot p = 0\,.$ $\square$

Next, since certain access operations are deterministic, we need an algebraic characterisation of determinacy. We borrow it from [14]:

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 $|a\rangle \, p = {}^{\ulcorner}(a \cdot p)\,.$

A relational characterisation of determinacy of a is $a^{\smile} \cdot a \leq 1$, where ${}^{\smile}$ is the converse operator. Since in our basic structure, the semiring, 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.

Lemma 5.4. *If a is deterministic and ${}^{\ulcorner}a$ is an atom then also $a^{\urcorner}$ is an atom.*

A proof can be found in the appendix. Interestingly, that proof does not presuppose that the set of all tests is an atomic lattice.

Now we define our model of linked object structures.

Definition 5.5. We assume a finite set L of *selector names* and a modal Kleene algebra S.

- 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. The set of all linked structures over L in denoted by S_L. Since $\square$ is proper and deterministic we will also view it as an element of S_L although it does not have selectors.

- A linked structure a is a *forest* iff a is acyclic and *injective*, i.e., has maximal in-degree 1 except possibly for $\square$. Algebraically this is expressed by the dual of the formula for determinacy, namely

  $$\forall\, p : |a'\rangle \, \langle a'| \, p \leq p\,, \qquad \text{where } a' =_{df} a \cdot \neg\square\,.$$

  Moreover, we define for forests a

  $$roots(a) =_{df} ({}^{\ulcorner}a - a^{\urcorner}) + \square \cdot {}^{\ulcorner}a$$

  By properness and since $\square$ is atomic, the term $\square \cdot {}^{\ulcorner}a$ equals $\square$ when $\square \leq a$ and is 0 otherwise.

- A forest a is called a *tree* iff $r =_{df} roots(a)$ is atomic and $\overline{\ulcorner a \urcorner} = \langle a^* | \, r$; in this case r is called the *root* of the tree and denoted by $root(a)$. If additionally $L = \{\mathsf{left}, \mathsf{right}\}$ then a is a binary tree while singly linked lists arise as the special case where we have only one selector, for instance next. In this case we call a tree a *chain*. Finally, a tree a is called a *cell* if ${}^{\ulcorner}a$ is an atomic test.

Note that $\square$ is a tree, while 0 is not, since it has no root. But at least, 0 is a forest. For a tree a we obtain from the above definition

$$root(a) = \begin{cases} \square & \text{if } a = \square \\ {}^{\ulcorner}a - a^{\urcorner} & \text{otherwise}\,. \end{cases}$$

6. Expressing Structural Properties of Linked Structures

As a further step we now define another separation relation that permits restricted sharing within linked structures. More precisely, we start with tree-like structures, e.g. a_1, a_2 and define them to be connected iff the root of a_2 equals one of the leafs of a_1. A main tool for expressing separateness and decomposability in such a fashion is the following.

Definition 6.1. Consider a selector set L. For trees $a_1, a_2 \in S_L$ we define *directed combinability* by

$$a_1 \rhd a_2 \;\Leftrightarrow_{df}\; {}^{\ulcorner}a_1 \cdot \overline{\ulcorner a_2 \urcorner} = 0 \,\land\, a_1^{\urcorner} \cdot a_2^{\urcorner} \leq \square \,\land\, a_1^{\urcorner} \cdot {}^{\ulcorner}a_2 = root(a_2)\,.$$

This relation guarantees domain disjointness and excludes occurrences of cycles, since $\ulcorner a_1 \cdot \overline{\ulcorner a_2 \urcorner} = 0 \Leftrightarrow \ulcorner a_1 \cdot \ulcorner a_2 = 0 \wedge \ulcorner a_1 \cdot a_2 \urcorner = 0$. Moreover, it excludes links from non-terminal nodes of a_1 to non-root nodes of a_2. Since 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 (cf. Figure 2, where the arrows with strokes indicate in which directions links are ruled out by the definition). Note that that root cannot occur more than once in a_1.

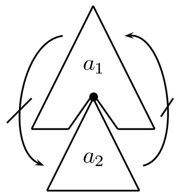


Figure 2: $\rhd$-combination of two trees

Note that by Lemma 4.4 the second conjunct above can be dropped when both arguments are singly-linked lists. We summarise some useful consequences of Definition 6.1.

Lemma 6.2. *If a is a tree then $\square \rhd a \Leftrightarrow \mathsf{FALSE}$ and $a \rhd \square \Leftrightarrow \square \leq a\urcorner$.*

Proof. First, we have $\square \rhd a \Leftrightarrow \square \cdot \overline{\ulcorner a \urcorner} = 0 \wedge \square \cdot a\urcorner \leq \square \wedge \square \cdot \ulcorner a = root(a)$. Now, $\square \cdot \ulcorner a = root(a)$ implies $root(a) \leq \square$ and , since $root(a)$ is atomic and hence $\neq 0$, it must equal $\square$. By definition also $a = \square$ which immediately contradicts $\square \cdot \overline{\ulcorner a \urcorner} = 0$.

Second, $a \rhd \square \Leftrightarrow \ulcorner a \cdot \square = 0 \wedge a\urcorner \cdot \square \leq \square \wedge a\urcorner \cdot \square = \square$. By the first result and since a is a tree the first conjunct follows from properness, the second is obvious and the third is equivalent to $\square \leq a\urcorner$. $\square$

Lemma 6.3. *For trees a_1 and a_2 with $a_1 \rhd a_2$ we have $root(a_1 + a_2) = root(a_1)$.*

Proof. First observe that $a_1 \neq \square$ by Lemma 6.2 and $a_1 \neq 0$ by definition. This implies $a_1 + a_2 \neq \square$, and we calculate $root(a_1 + a_2) = \ulcorner a_1 \cdot \neg a_1 \urcorner \cdot \neg a_2 \urcorner + \ulcorner a_2 \cdot \neg a_1 \urcorner \cdot \neg a_2 \urcorner$.

The first summand reduces to $\ulcorner a_1 \cdot \neg a_1 \urcorner = root(a_1)$, since $a_1 \rhd a_2$ implies $\ulcorner a_1 \cdot a_2 \urcorner = 0$, i.e., $\ulcorner a_1 \leq \neg a_2 \urcorner$. The second summand is, by definition, equal to $root(a_2) \cdot \neg a_1 \urcorner$. Since $a_1 \rhd a_2$ implies $root(a_2) \leq a_1 \urcorner$, this summand reduces to 0. $\square$

Since the directed disjointness relation $\rhd$ is defined only on tree-like structures, we extend it now to arbitrary forests.

Definition 6.4. Consider a selector set L and let $a, b \in S_L$ be forests with $a = \sum a_i$ and $b = \sum b_j$, where the a_i and b_j are the constituent trees with $a_{i_1} \text{\textcircled{\#}} a_{i_2}$ $(i_1 \neq i_2)$ and $b_{j_1} \text{\textcircled{\#}} b_{j_2}$ $(j_1 \neq j_2)$. Then we define *directed combinability* by

$$a \rhd b \ \Leftrightarrow_{df} \ \forall j : a \text{\textcircled{\#}} b_j \vee (\exists i : \ a_i \rhd b_j \wedge \bigwedge_{k \neq i} a_k \text{\textcircled{\#}} b_j) \ .$$

This requires at least two constituent trees of forests a and b to be connected wrt. $\rhd$ while all unconnected trees must be strongly disjoint.

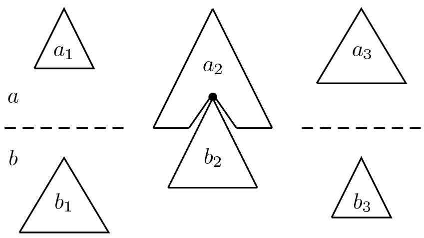


Figure 3: $\rhd$-combination of two forests a, b

We now show that $\rhd$ guarantees preservation of linked structures under $+$.

Lemma 6.5. *Let a_1, a_2 be arbitrary elements of a modal semiring.*

1. *If the a_i are deterministic and $\ulcorner a_1 \cdot \ulcorner a_2 = 0$ then also $a_1 + a_2$ is deterministic.*
2. *If the a_i are injective and $a_1^{\urcorner} \cdot a_2^{\urcorner} \leq \square$ then also $a_1 + a_2$ is injective.*
3. *If the a_i are acyclic and $a_2^{\urcorner} \cdot \ulcorner a_1 = 0$ then also $a_1 + a_2$ is acyclic.*

Proof.

1. By distributivity, $\langle a_1 + a_2|\, |a_1 + a_2\rangle\, p \leq p$, since $\langle a_i|\, |a_i\rangle\, p \leq p$ and $\langle a_2|\, |a_1\rangle\, p \leq 0 \wedge \langle a_1|\, |a_2\rangle\, p \leq 0$ by $\ulcorner a_1 \cdot \ulcorner a_2 = 0$.
2. By definition and distributivity we have $(a_1 + a_2)' = (a_1 + a_2) \cdot \neg\square = a_1' + a_2'$. Now we can reason symmetrically to Part 1.
3. Assume an arbitrary atomic test $p \neq \square$. We show $p \cdot (a_1 + a_2)^+ \cdot p = 0$. First note that if $a_2^{\urcorner} \cdot \ulcorner a_1 = 0$ then $(a_1 + a_2)^+ = {a_1}^+ + {a_1}^+ \cdot {a_2}^+ + {a_2}^+$. This follows using $(x + y)^* = x^* \cdot (y \cdot x^*)^*$, domain properties and the definition of $\_^+$.

   Hence, it remains to show $p \cdot {a_1}^+ \cdot p = 0 \wedge p \cdot {a_1}^+ \cdot {a_2}^+ \cdot p = 0 \wedge p \cdot {a_2}^+ \cdot p = 0$. The first and last conjuncts follow from the assumption.

   If the second conjunct were false, then necessarily $0 \neq p \cdot {a_1}^+ = p \cdot a_1 \cdot a_1^*$ and hence $p \cdot \ulcorner a_1 \neq 0$. Likewise, $p \cdot a_2^{\urcorner} \neq 0$. Since p is an atom, these two conditions are equivalent to $p \leq \ulcorner a_1$ and $p \leq a_2^{\urcorner}$, resp., and hence imply $p \leq a_2^{\urcorner} \cdot \ulcorner a_1$. This is a contradiction to $a_2^{\urcorner} \cdot \ulcorner a_1 = 0$.

□

Corollary 6.6. *Consider a selector set L. If $a_1, a_2 \in S_L$ are linked structures with $\ulcorner a_1 \cdot \overline{a_2} = 0$ and $a_1^{\urcorner} \cdot \ulcorner a_2 \leq \square$ then also $a_1 + a_2$ is a linked structure in S_L.*

Proof. Properness of $a_1 + a_2$ follows from Corollary 2.1. The remaining properties required of $a_1 + a_2$ are implied by Lemma 6.5. □

Lemma 6.7. *If a_1, a_2 are trees with $a_1 \triangleright a_2$ and then $a_1 + a_2$ is again a tree whose root is that of a_1.*

Proof. Since $a_1 \triangleright a_2$ implies the assumptions of Cor. 6.6, $a_1 + a_2$ is a linked structure. Moreover, we know by Lemma 6.3 that $root(a_1 + a_2) = root(a_1)$ and thus is atomic. It remains to show $\overline{a_1 + a_2} = \langle (a_1 + a_2)^*|\, root(a_1)$. We know that $\overline{a_1 + a_2} = \overline{a_1} + \overline{a_2}$.
($\leq$) By the assumptions and isotony, $\overline{a_1} = \langle {a_1}^*|\, root(a_1) \leq \langle (a_1 + a_2)^*|\, root(a_1)$.
Second, again by the assumptions, $\langle b|\, \langle a|\, p = \langle a \cdot b|\, p$ and isotony, we obtain

$$\overline{a_2} = \langle {a_2}^*|\, root(a_2) \leq \langle {a_2}^*|\, \overline{a_1} = \langle {a_2}^*|\, \langle {a_1}^*|\, root(a_1) = \langle {a_1}^* \cdot {a_2}^*|\, root(a_1) \leq \langle (a_1 + a_2)^*|\, root(a_1)\ .$$

($\geq$) For abbreviation, set $q =_{df} \overline{a_1 + a_2} = \langle {a_1}^*|\, root(a_1) + \langle {a_2}^*|\, root(a_2)$. Using diamond induction, $\langle (a_1 + a_2)^*|\, root(a_1) \leq q$ is implied by $root(a_1) \leq q$ and $\langle a_1 + a_2|\, q \leq q$. The first assertion is clear. The second one is, by distributivity and again $\langle b|\, \langle a|\, p = \langle a \cdot b|\, p$, equivalent to

$$\langle {a_1}^* \cdot a_1|\, root(a_1) + \langle {a_1}^* \cdot a_2|\, root(a_1) + \langle {a_2}^* \cdot a_1|\, root(a_2) + \langle {a_2}^* \cdot a_2|\, root(a_2) \leq q\ .$$

For the first and last summands this is clear. The remaining ones are treated by

$$\langle {a_1}^* \cdot a_2|\, root(a_1) = \langle a_2|\, root(a_1) + \langle {a_1}^* \cdot a_1 \cdot a_2|\, root(a_1) = \langle {a_1}^* \cdot a_1 \cdot root(a_2) \cdot a_2|\, root(a_1) =$$
$$\langle a_2|\, ((root(a_1) \cdot {a_1}^* \cdot a_1)^{\urcorner} \cdot root(a_2)) \leq \langle a_2|\, root(a_2)$$

and

$$\langle {a_2}^* \cdot a_1|\, root(a_2) = \langle a_1|\, root(a_2) + \langle {a_2}^* \cdot a_2 \cdot a_1|\, root(a_2) = 0\ .$$

□

Corollary 6.8. *Since lists are a special case of trees, the same holds for lists.*

Corollary 6.9. *If a_1, a_2 are forests and $a_1 \rhd a_2$ or $a_1 \circledast a_2$ holds then also $a_1 + a_2$ is a forest.*

Proof. Immediate from Lemma 6.7 and the definition of $\rhd$ on forests. $\square$

Again we can lift the relation $\rhd$ to predicates. First, we define the following special predicates

$$\begin{aligned}
\mathsf{cell} &=_{df} \{a : a \text{ is a cell}\}\,, \\
\mathsf{list} &=_{df} \{a : a \text{ is a chain}\}\,, \\
\mathsf{tree} &=_{df} \{a : a \text{ is a tree}\}\,, \\
\mathsf{forest} &=_{df} \{a : a \text{ is a forest}\}\,.
\end{aligned}$$

Clearly, $\mathsf{cell} \cap S_{\mathsf{next}} \subseteq \mathsf{list} \subseteq \mathsf{tree} \subseteq \mathsf{forest}$ and $\mathsf{cell} \subseteq \mathsf{tree}$.

Definition 6.10. For a selector set L and $P, Q \subseteq \mathsf{forest} \cap S_L$ we define *directed combinability* $\circledcirc$ by

$$P \circledcirc Q =_{df} \{a_1 + a_2 : a_1 \in P,\, a_2 \in Q,\, a_1 \rhd a_2\}\,.$$

To avoid excessive notation, in the sequel we tacitly assume that all predicates involved in our formulas are restricted to the same set of selectors as in this definition.

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

Lemma 6.11. $\mathsf{forest} \circledcirc \mathsf{forest} \subseteq \mathsf{forest}$, $\mathsf{tree} \circledcirc \mathsf{tree} \subseteq \mathsf{tree}$ *and* $\mathsf{list} \circledcirc \mathsf{list} \subseteq \mathsf{list}$. *As particular cases* $\mathsf{cell} \circledcirc \mathsf{list} \subseteq \mathsf{list}$, $\mathsf{tree} \circledcirc \mathsf{cell} \subseteq \mathsf{tree}$ *and* $\mathsf{cell} \circledcirc \mathsf{tree} \subseteq \mathsf{tree}$.

Lemma 6.12. *Let $P, Q, R \subseteq \mathsf{tree}$ then*

$$P \circledcirc (Q \circledcirc R) \subseteq (P \circledcirc Q) \circledcirc R\,, \quad (1)$$

$$P \circledcirc (Q \circledcirc R) \subseteq (P \circledast R) * Q\,, \quad (2)$$

$$(P \circledcirc Q) \circledast R \subseteq P \circledcirc (Q \circledast R)\,, \quad (3)$$

$$P \circledast (Q \circledcirc R) \subseteq (P \circledast Q) \circledcirc R\,. \quad (4)$$

Proof. We start with the first two laws. Assume $a_1 \in P$, $a_2 \in Q$, $a_3 \in R$ and $a_1 \rhd (a_2 + a_3)$ and $a_2 \rhd a_3$. By Lemma 6.2 we know $a_1, a_2 \neq \square$. Moreover, by Lemma 6.7 $a_2 + a_3$ is a tree with $root(a_2 + a_3) = root(a_2)$. Now, $a_1 \rhd (a_2 + a_3)$ implies $a_1^{\urcorner} \cdot {}^{\ulcorner}a_2 + a_1^{\urcorner} \cdot {}^{\ulcorner}a_3 = {}^{\ulcorner}a_2 - a_2^{\urcorner}$. Multiplying this equation by ${}^{\ulcorner}a_2$ and using that $a_2 \rhd a_3$ implies ${}^{\ulcorner}a_3 \cdot {}^{\ulcorner}a_2 = 0$ we obtain $a_1^{\urcorner} \cdot {}^{\ulcorner}a_2 = {}^{\ulcorner}a_2 - a_2^{\urcorner} = root(a_2)$. Hence, $a_1^{\urcorner} \cdot {}^{\ulcorner}a_3 = 0$, since $root(a_2)$ is atomic.

By this we can immediately derive from distributivity and the definitions that $a_1 \rhd a_2 \wedge (a_1 + a_2) \rhd a_3$ and $a_1 \circledast a_3 \wedge {}^{\ulcorner}(a_1 + a_3) \cdot {}^{\ulcorner}a_2 \leq 0$, which shows the first two laws.

For the third law, assume $a_1 \rhd a_2$ and $(a_1 + a_2) \circledast a_3$ which is equivalent to $a_1 \circledast a_3 \wedge a_2 \circledast a_3$. Note, that $a_2 + a_3$ is a forest. Hence by Definition 6.4 the claim is immediate.

Finally, the last law follows directly from bilinearity of $\circledast$ and the definition of $\rhd$ on forests. $\square$

7. Assertions and Program Commands

We now define programming constructs to treat concrete verification examples.

As a first step we extend our predicates by a possibility of directly addressing the roots of the characterised structures. For this we start by defining, similar to standard separation logic, so-called *stores*.

Definition 7.1. A *store* is a partial mapping from program identifiers to nodes, i.e., atomic tests. The domain of a store s is denoted by $dom(s)$. A *state* is a pair (s, a) with a store s and a linked structure a. For an identifier i and a sequence $l = l_1 \dots l_n \in L^+$ of selector names, the semantics of the expression $\mathsf{i}.l$ w.r.t. a state (s, a) is defined as

$$[\![\mathsf{i}.l]\!]_{(s,a)} =_{df} \begin{cases} \langle a_{l_1} \cdot \dots \cdot a_{l_n} | (s(\mathsf{i})) & \text{if } \mathsf{i} \in dom(s)\,, \\ 0 & \text{otherwise}\,. \end{cases}$$

Note that $\langle a_{l_1} \dots a_{l_n} | (s(\mathsf{i}))$ is either an atomic test or 0 by determinacy of each access element a_{l_i}.

Definition 7.2. For an identifier i and a predicate $P \subseteq \mathsf{tree}$ we define its extension $P(\mathsf{i})$ to states by

$$P(\mathsf{i}) =_{df} \{(s, a) : a \in P,\, \mathsf{i} \in dom(s),\, root(a) = s(\mathsf{i})\}\ .$$

By this we can refer to the root of an access element a in predicates about tree-like structures. If we are not interested in the root nodes we will, by slight abuse of notation, simply write P also to mean the extension of P to states, i.e., $P =_{df} \{(s, a) : a \in P\}$. In particular, for operator $\circ \in \{=, \neq\}$ and $l, m \in L^+$, we define special predicates by

$$\begin{array}{rcl} (\ \mathsf{i} \circ \square\ ) & =_{df} & \{(s, a) : \mathsf{i} \in dom(s),\, s(\mathsf{i}) \circ \square\}\ , \\ (\ \mathsf{i}.l \circ \square\ ) & =_{df} & \{(s, a) : 0 \neq [\![\mathsf{i}.l]\!]_{(s,a)} \circ \square\}\ , \\ (\ \mathsf{i}.l = \mathsf{j}.m\ ) & =_{df} & \{(s, a) : 0 \neq [\![\mathsf{i}.l]\!]_{(s,a)} = [\![\mathsf{j}.m]\!]_{(s,a)} \neq 0\}\ . \end{array}$$

The mechanism of predicate extension cannot be used with expressions e involving selector chains. Simply setting $P(e) =_{df} \{(s, a) : a \in P, root(a) = [\![e]\!]_{(s,a)}\}$ would, for instance, not work in a formula like $P(\mathsf{i}) \circledR Q(\mathsf{i}.l)$, since by the definition of $\circledR$ we cannot have $s(\mathsf{i}) \leq \ulcorner a$ with $a \in Q$. Instead, we use a syntactic solution: we view $P(\mathsf{i}) \circledR Q(\mathsf{i}.l)$ as an abbreviation for $(P(\mathsf{i}) \circledR Q(\mathsf{j})) \cap (\mathsf{j} = \mathsf{i}.l)$ where j is a fresh identifier. The predicate $\mathsf{j} = \mathsf{i}.l$ is used to name an otherwise anonymous node within the structure rooted in i.

The lifting of predicates to stores allows placing side conditions on the root elements of predicates in formulas. This has many useful consequences. We summarise a few in the following.

Lemma 7.3. *Let* $\mathsf{i}, \mathsf{j}, \mathsf{k}$ *be identifiers and* $\{\square\} \not\subseteq P, Q, R \subseteq \mathsf{tree}$. *Then*

$$(P(\mathsf{i}) \circledR Q(\mathsf{j})) \circledR R(\mathsf{k}) = P(\mathsf{i}) \circledR (Q(\mathsf{j}) \circledR R(\mathsf{k})) \qquad \textit{if } \exists\, l \in L^+ : \mathsf{j}.l = \mathsf{k}\ , \qquad (5)$$

$$(P(\mathsf{i}) \circledR Q) \circledast R(\mathsf{j}) = P(\mathsf{i}) \circledR (Q \circledast R(\mathsf{j})) \qquad \textit{if } \forall\, l \in L^* : \mathsf{i}.l \neq \mathsf{j}\ , \qquad (6)$$

$$(P(\mathsf{i}) \circledR Q(\mathsf{j})) \circledR R(\mathsf{k}) = P(\mathsf{i}) \circledR (Q(\mathsf{j}) \circledast R(\mathsf{k})) \qquad \textit{if } \mathsf{j} = \mathsf{i}.l \wedge \mathsf{k} = \mathsf{i}.m \wedge l, m \in L\ . \qquad (7)$$

$$(P(\mathsf{i}) \circledast Q(\mathsf{j})) \circledR R(\mathsf{k}) = P(\mathsf{i}) \circledast (Q(\mathsf{j}) \circledR R(\mathsf{k})) \qquad \textit{if } \exists\, l \in L^+ : \mathsf{j}.l = \mathsf{k}\ . \qquad (8)$$

Proof. Assume $a_1 \in P(\mathsf{i}) \wedge a_2 \in Q(\mathsf{j}) \wedge a_3 \in R(\mathsf{k})$. By assumption $a_i \neq \square$.
(5) We only show the $\subseteq$-direction, since $\supseteq$ was shown in Lemma 6.12. By the definitions it remains to show that $(a_1 + a_2) \rhd a_3 \wedge a_1 \rhd a_2$ implies $a_1 \rhd (a_2 + a_3) \wedge a_2 \rhd a_3$. The assumption $(a_1 + a_2) \rhd a_3$ resolves to

$$\ulcorner a_1 \cdot \overline{\ulcorner a_3 \urcorner} \leq 0 \wedge \ulcorner a_2 \cdot \overline{\ulcorner a_3 \urcorner} \leq 0 \wedge a_1^{\urcorner} \cdot a_3^{\urcorner} \leq \square \wedge a_2^{\urcorner} \cdot a_3^{\urcorner} \leq \square \wedge a_1^{\urcorner} \cdot \ulcorner a_3 + a_2^{\urcorner} \cdot \ulcorner a_3 = root(a_3)\ . \qquad (*)$$

The last conjunct implies $a_2^{\urcorner} \cdot \ulcorner a_3 \leq root(a_3)$. Moreover, note that the side condition of (5) implies $root(a_3) \leq \overline{\ulcorner a_2 \urcorner}$. Hence, $root(a_3) = root(a_3) \cdot \ulcorner a_3 \leq \overline{\ulcorner a_2 \urcorner} \cdot \ulcorner a_3 = \ulcorner a_2 \cdot \ulcorner a_3 + a_2^{\urcorner} \cdot \ulcorner a_3 = a_2^{\urcorner} \cdot \ulcorner a_3$ and therefore $root(a_3) = a_2^{\urcorner} \cdot \ulcorner a_3$. This shows $a_2 \rhd a_3$, which further by Lemma 6.3 implies $root(a_2 + a_3) = root(a_2)$ and $a_1^{\urcorner} \cdot \ulcorner a_3 \leq root(a_3)$. From this we obtain by $(*)$, since $root(a_3) \neq \square$ is an atom and $a_1^{\urcorner} \cdot a_2^{\urcorner} \leq \square$ by $a_1 \rhd a_2$, that $a_1^{\urcorner} \cdot \ulcorner a_3 = 0$ as well. Hence, again by $a_1 \rhd a_2$, we obtain $root(a_2) = a_1^{\urcorner} \cdot \ulcorner a_2 + a_1^{\urcorner} \cdot \ulcorner a_3$, which establishes $a_1 \rhd (a_2 + a_3)$.
(6) The $\subseteq$-direction was again shown in Lemma 6.12. Now assume $a_1 \rhd (a_2 + a_3)$ and $a_2 \textcircled{\#} a_3$. The side condition implies $\overline{a_1^{\urcorner}} \cdot root(a_3) \leq 0$ which in turn implies $a_1^{\urcorner} \cdot \ulcorner a_3 \leq \neg root(a_3)$. Therefore $a_1 \rhd a_3$ does not hold and consequently $a_1 \rhd a_2$ and $a_1 \textcircled{\#} a_3$ need to be true by the definition of $\rhd$ for forests.
(7) We assume $(a_1 + a_2) \rhd a_3 \wedge a_1 \rhd a_2$ and show $a_1 \rhd (a_2 + a_3) \wedge a_2 \textcircled{\#} a_3$. As for (5), $(a_1 + a_2) \rhd a_3$ implies $a_1^{\urcorner} \cdot \ulcorner a_3 + a_2^{\urcorner} \cdot \ulcorner a_3 = root(a_3)$. We calculate $a_2^{\urcorner} \cdot \ulcorner a_3 \leq a_2^{\urcorner} \cdot root(a_3) = a_2^{\urcorner} \cdot \overline{a_1^{\urcorner}} \cdot root(a_3) = a_2^{\urcorner} \cdot a_1^{\urcorner} \cdot root(a_3) \leq \square \cdot \ulcorner a_3 \leq$

0 by assumptions and the side condition. Hence, $a_2 \triangleright a_3$ and $a_1^{\urcorner} \cdot {}^{\ulcorner}a_3 = \mathit{root}(a_3)$ which by the assumption $(a_1 + a_2) \triangleright a_3$ further implies $a_1 \triangleright a_3$. Next, the reverse direction is shown by $\mathit{root}(a_i) \leq \ulcorner\overline{a_1}\urcorner \Rightarrow \neg(a_1 \circledast a_i)$, which in turn implies by $a_1 \triangleright (a_2 + a_3)$ and Definition 6.4 that $a_1 \triangleright a_i$ for $i = 2, 3$. Now, using assumption $a_2 \triangleright a_3$ we immediately get $(a_1 + a_2) \triangleright a_3$ from Definition 6.4 again.
(8) Again $\supseteq$ was proved in Lemma 6.12 while $\subseteq$ holds , since the side condition implies $\mathit{root}(a_3) \leq \ulcorner\overline{a_2}\urcorner$ and hence $a_1 \triangleright a_3$ can not hold by $a_1 \circledast a_2$. Therefore by definition we can only have $a_1 \circledast a_3 \wedge a_2 \triangleright a_3$. Now the claim follows by bilinearity of $\circledast$. $\Box$

We now consider the special case of chains.

Corollary 7.4. *For arbitrary $P, Q, R \subseteq$ list and identifier i we have*

$$(P(\mathsf{i}) \circledgtr Q(\mathsf{i.next})) \circledgtr R(\mathsf{i.next.next}) = P(\mathsf{i}) \circledgtr (Q(\mathsf{i.next}) \circledgtr R(\mathsf{i.next.next})),$$

i.e., $\circledgtr$ is associative on lists.

Proof. This follows from Lemma 7.3, Equation (5) by setting $\mathsf{j} = \mathsf{i.next}$ and $\mathsf{j.next} = \mathsf{k}$. $\Box$

Next we want to give the semantics of program commands, in particular, of assignments of the form $\mathsf{i}.l := e$. To this end we enrich our algebra by another ingredient, namely by *twigs*, i.e., abstract representations of single edges in the graph corresponding to a linked structure. Special assignments of the above form will add or delete such twigs.

Definition 7.5. Assuming atomic tests with $p \cdot q = 0 \wedge p \cdot \Box = 0$, 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* of a linked structure a is $(p \mapsto q) \,|\, a =_{df} (p \mapsto q) + \neg p \cdot a$. We assume that $|$ binds tighter than $+$ but less tight than $\cdot$.

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.

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 $(p \mapsto q)^{\urcorner} = q$ and ${}^{\ulcorner}(p \mapsto q) = p$.

Lemma 7.6. $\ulcorner\overline{p \mapsto q}\urcorner = p + q$ *and* $\mathit{root}(p \mapsto q) = p$.

Proof. The first result is trivial. Second, $\mathit{root}(p \mapsto q) = {}^{\ulcorner}(p \mapsto q) \cdot \neg(p \mapsto q)^{\urcorner} = p \cdot \neg q = p$, since $p \cdot q = 0 \Leftrightarrow p \leq \neg q$ by shunting. $\Box$

Note that by $a = 0 \Leftrightarrow {}^{\ulcorner}a = 0$, cells are always non-empty.

Lemma 7.7. *For a cell a we have $\mathit{root}(a) = {}^{\ulcorner}a$, hence $\neg \mathit{root}(a) \cdot a = 0$.*

Proof. By definition $\mathit{root}(a) \leq {}^{\ulcorner}a$ and $\mathit{root}(a) \neq 0$. Thus $\mathit{root}(a) = {}^{\ulcorner}a$. $\Box$

Lemma 7.8. *Twigs $p \mapsto q$ are cells.*

Proof. By assumption, ${}^{\ulcorner}(p \mapsto q) = p$ is atomic and $\neq \Box$, hence proper. Moreover, $\mathit{reach}(p, p \mapsto q) = \ulcorner\overline{p \mapsto q}\urcorner = p + q$, acyclicity holds by $p \cdot q = 0$. To show determinacy we conclude for arbitrary tests s: $q \cdot s \leq q \Rightarrow q \cdot s = 0 \vee q \cdot s = q \Leftrightarrow q \cdot s = 0 \vee q \leq s$. Hence, $\langle p \mapsto q |\, | p \mapsto q \rangle\, s \leq \langle p \mapsto q | \, p \leq q \leq s$. The calculation for injectivity is analogous. $\Box$

Now, we can summarise a few consequences that will be used in the examples to come.

Corollary 7.9. $(\mathsf{i} \neq \Box) \cap \mathsf{list}(\mathsf{i}) = \mathsf{cell}(\mathsf{i}) \circledgtr \mathsf{list}$ *and* $(\mathsf{i} = \Box) \cap \mathsf{list}(\mathsf{i}) = \{\Box\}$.

Proof. We only show $\mathsf{list}(\mathsf{i}) = \mathsf{cell}(\mathsf{i}) \circledgtr \mathsf{list}$, since the second result is obvious. The $\supseteq$-direction follows from Lemma 6.7. For $\subseteq$ we know by the assumption $\mathsf{i} \neq \Box$ and the definitions that $a \neq \Box$ for all $(s, a) \in \mathsf{list}(\mathsf{i})$. Since a is a chain and therefore acyclic, we can write $a = \mathit{root}(a) \mapsto \mathit{root}(b) + b$ for a $b =_{df} \neg \mathit{root}(a) \cdot a$. Note that by Lemma 7.8 $\mathit{root}(a) \mapsto \mathit{root}(b) \in \mathsf{cell}$. By this one can show $b \in \mathsf{list}$ and $\mathit{root}(a) \mapsto \mathit{root}(b) \triangleright b$. $\Box$

Corollary 7.10. $(\mathsf{i.left} \neq \Box) \cap (\mathsf{i.right} \neq \Box) \cap \mathsf{tree(i)} \;=\; \mathsf{cell\,(i)} \mathbin{\circledR} (\mathsf{tree(i.left)} \circledast \mathsf{tree(i.right)})$.

Proof. A proof can be constructed similarly as in the case of Corollary 7.9. □

Now, we are ready to provide definitions for concrete program *commands*. They are modelled in our approach as relations between states.

To treat assignments $\mathsf{i}.l := e$, we use twigs (cf. Definition 7.5) to describe updates of linked structures by adding or changing links.

We use expressions e of the form $\langle var \rangle.l$ where var is an arbitrary variable and $l \in L^+$.

Definition 7.11. In the following we assume an identifier i, a selector set L, a selector name $l \in L$ and an expression e for which $[\![e]\!]_{(s,a)}$ is always an atomic test. For a linked structure $a \in S_L$ we abbreviate the subfamily $(a_k)_{k \in L-\{l\}}$ by a_{L-l}. Then we set

$$
\begin{array}{lll}
\mathsf{i} := e & =_{df} & \{((s,a),(s[\mathsf{i} \leftarrow p],a)) : \mathsf{i} \in dom(s),\ p = [\![e]\!]_{(s,a)}\},\\
\mathsf{i}.l := e & =_{df} & \{((s,a),(s,(s(\mathsf{i}) \mapsto [\![e]\!]_{(s,a)})|a_l + a_{L-l}) : \mathsf{i} \in dom(s),\ s(\mathsf{i}) \neq \Box,\ s(\mathsf{i}) \leq \ulcorner a_l\},\\
\mathsf{i} := \mathsf{new\ cell\,()} & =_{df} & \{((s,a),(s[\mathsf{i} \leftarrow p],(p \mapsto \Box)|a) : \mathsf{i} \in dom(s),\ p \text{ is an atomic test},\ p \leq \neg\ulcorner a,\ p \neq \Box\},\\
\mathrm{delete}(\mathsf{i}) & =_{df} & \{((s,a),(s,\neg p \cdot a)) : p = s(\mathsf{i}),\ \mathsf{i} \in dom(s),\ p \neq \Box\}.
\end{array}
$$

In general selector assignments do not preserve treeness. We provide sufficient conditions for that in the form of Hoare triples in the next section.

8. Inference Rules

As already mentioned in Section 2, one can encode subsets or predicates as sub-identity relations. This way we can view state predicates P as commands of the form $\{(\sigma,\sigma) : \sigma \in P\}$ where $\sigma = (s,a)$ for some store s and linked structure a. We will not distinguish predicates and their corresponding commands notationally. Following [6, 15] we encode Hoare triples with state predicates P, Q and command C as

$$\{P\}\ C\ \{Q\} \;\Leftrightarrow_{df}\; P\,;C \subseteq C\,;Q \;\Leftrightarrow\; P\,;C \subseteq U\,;Q,$$

where U is the universal relation on states.

8.1. Rules for Selector Assignments

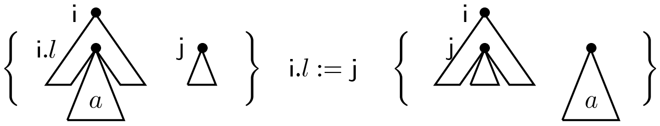
For better readability of concrete rules, we introduce some syntactic sugar and abbreviate, for expressions e, e' and operators $\circ \in \{*, \circledast, \mathbin{\circledR}\}$, formulas of the form $Q \circ P(e) \wedge e' = e$ by $Q \circ P(e,e')$. By this we can explicitly list expressions that are aliases for the same root node. For instance, we can abbreviate the rule

$$
\begin{array}{lcl}
\{\ P(\mathsf{j}) \mathbin{\circledR} Q(\mathsf{j}.l)\ \} & & \{\ P(\mathsf{j}) \mathbin{\circledR} Q(\mathsf{j}.l)\ \}\\
\ \mathsf{i} := \mathsf{j}.l; & \text{to} & \ is := \mathsf{j}.l;\\
\{\ P(\mathsf{j}) \mathbin{\circledR} Q(\mathsf{j}.l) \ \wedge\ \mathsf{i} = \mathsf{j}.l\ \} & & \{\ P(\mathsf{j}) \mathbin{\circledR} Q(\mathsf{j}.l,\mathsf{i})\ \}\ .
\end{array}
$$

Lemma 8.1. *For predicates $P, Q, R \subseteq \mathsf{tree}$, identifiers i, j and link $l \in L$ we have*

$$
\begin{array}{lll}
\{\ (P(\mathsf{i}) \mathbin{\circledR} Q(\mathsf{i}.l)) \circledast R(\mathsf{j})\ \} & \{\ P(\mathsf{i}) \circledast R(\mathsf{j}) \ \wedge\ \mathsf{i}.l = \Box\ \} & \{\ P(\mathsf{i}) \mathbin{\circledR} Q(\mathsf{i}.l)\ \}\\
\ \mathsf{i}.l := \mathsf{j}; & \ \mathsf{i}.l := \mathsf{j}; & \ \mathsf{i}.l := \Box;\\
\{\ (P(\mathsf{i}) \mathbin{\circledR} R(\mathsf{j},\mathsf{i}.l)) \circledast Q\ \} & \{\ P(\mathsf{i}) \mathbin{\circledR} R(\mathsf{j},\mathsf{i}.l)\ \} & \{\ P(\mathsf{i}) \circledast Q \ \wedge\ \mathsf{i}.l = \Box\ \}
\end{array}
$$

For the proof see below. The conjuncts $\mathsf{i}.l = \Box$ are useful, since they show that the assignments involved do not introduce memory leaks. Note that $\cap$ on predicates corresponds to their logical conjunction $\wedge$. To provide more intuition of what is happening in the leftmost rule of Lemma 8.1, we depict the shapes of the trees in the pre- and postcondition:



Note that after the assignment the subtree a still resides untouched in memory; however, unless there are links to it from elsewhere, it is inaccessible and hence garbage. The other rules can be illustrated similarly.

Proof. We only give a proof of the leftmost rule. The remaining ones can be proved similarly. Assume trees $a_1 \in P \wedge a_2 \in Q \wedge a_3 \in R$ with $a_1 \triangleright a_2 \wedge a_1 \mathbin{\#\!\!\!\!\bigcirc} a_3 \wedge a_2 \mathbin{\#\!\!\!\!\bigcirc} a_3 \wedge a = a_1 + a_2 + a_3$.

We decompose each a_i into its l-part $b_i =_{df} (a_i)_l$ and the rest $c_i =_{df} (a_i)_{L-l}$ and show $((root(a_1) \mapsto root(a_3))|b_1 + c_1) \mathbin{\#\!\!\!\!\bigcirc} a_2$. This is equivalent to $c_1 \mathbin{\#\!\!\!\!\bigcirc} c_2 \wedge (root(a_1) \mapsto root(a_3)) \mathbin{\#\!\!\!\!\bigcirc} b_2 \wedge (\neg root(a_1) \cdot b_1) \mathbin{\#\!\!\!\!\bigcirc} b_2$.

By assumption we know $(root(a_1) \cdot b_1)^{\urcorner} = root(a_2)$. This implies by the injectivity property of trees and atomicity that $(\neg root(a_1) \cdot b_1)^{\urcorner} \cdot {}^{\ulcorner}a_2 = 0$. Hence, together with $a_1 \mathbin{\triangleright\!\!\!\!\bigcirc} a_2$ we have $(\neg root(a_1) \cdot b_1) \mathbin{\#\!\!\!\!\bigcirc} b_2$.

By determinacy and again the assumption on the roots, $a_1^{\urcorner} \cdot {}^{\ulcorner}a_2 = root(a_2)$ is equivalent to $b_1^{\urcorner} \cdot {}^{\ulcorner}a_2 = root(a_2) \wedge c_1^{\urcorner} \cdot {}^{\ulcorner}a_2 = 0$. Hence, $c_1 \mathbin{\#\!\!\!\!\bigcirc} c_2$.

The rest follows from $a_1 \mathbin{\triangleright\!\!\!\!\bigcirc} a_2$ and it remains to show $((root(a_1) \mapsto root(a_3))|b_1 + c_1) \mathbin{\triangleright\!\!\!\!\bigcirc} a_3$. This can be calculated by similar considerations as above using $a_1 \mathbin{\#\!\!\!\!\bigcirc} a_3$. Therefore, $((root(a_1) \mapsto root(a_3))|b_1) + a_{L-l} \in (P(\mathsf{i}) \mathbin{\triangleright\!\!\!\!\bigcirc} R(\mathsf{j}, \mathsf{i}.l)) \circledast Q$.

□

8.2. Frame Rules

For an algebraic proof of the frame rules with the new operators we follow precursor ideas of [15, 16]. Proofs are treated there in a general and relational setting, so that we can easily adapt these results for the present work. The $\circledast$ and $\mathbin{\triangleright\!\!\!\!\bigcirc}$ operators are lifted to commands in the following by

$$(s, a)\ C \circ D\ (s', a') \quad \Leftrightarrow \quad \exists\, a_1, a_2, a'_1, a'_2 : a = a_1 + a_2 \wedge a_1 \# a_2 \wedge a' = a'_1 + a'_2 \wedge a'_1 \# a'_2 \\ \wedge (s, a_1)\ C\ (s', a'_1) \wedge (s, a_2)\ D\ (s', a'_2)$$

where $\circ \in \{\circledast, \mathbin{\triangleright\!\!\!\!\bigcirc}\}$ and $\# \in \{\mathbin{\#\!\!\!\!\bigcirc}, \triangleright\}$ resp.

Lemma 8.2. *Assume the following conditions for command C and predicates $P \subseteq dom(C)$ and R:*

$$(P \circledast R)\,;C \subseteq (P\,;C) \circledast R\ , \qquad C \circledast R \subseteq C\ .$$

Then for all predicates Q we have the $\circledast$ frame rule

$$\frac{\{P\}\,C\,\{Q\}}{\{P \circledast R\}\,C\,\{Q \circledast R\}}\ .$$

The assumptions restrict the behaviour of the command C, s.t. it can at most modify linked structures in P and leaves those in R untouched, i.e., C disregards linked structures in R.

The proof is a direct translation of the corresponding one for the $*$ frame rule in [16].

Lemma 8.3. *The $\circledast$ frame rule is valid for all predicates R and commands C that do not modify or reference any expression occurring in R.*

Proof. By Lemma 8.2 it suffices to show that all such commands satisfy the assumptions made there. We only consider the base cases in Definition 7.11. A proof for commands of the form $C_1\,; \ldots ; C_n$ can be constructed inductively from them. The cases for allocation and deallocation are obvious. For simple variable assignments, only the store component is modified and the argumentation is the same as in standard separation logic. Therefore we now concentrate on selector assignments $C = (\mathsf{i}.l := e)$. For the reader's benefit we repeat the semantic definition:

$$\mathsf{i}.l := e =_{df} \{((s, a), (s, (s(\mathsf{i}) \mapsto [\![e]\!]_{(s,a)})|a_l + a_{L-l}) : \mathsf{i} \in dom(s),\ s(\mathsf{i}) \neq \square,\ s(\mathsf{i}) \leq {}^{\ulcorner}a_l\}\ .$$

We outline a proof for the first assumption of Lemma 8.2; for the second one the argumentation is analogous. For given states (s_i, a_i), the premise of the rule resolves pointwise to

$$((s_1, a_1), (s_2, a_2)) \in C \wedge (s_1, a_p) \in P \wedge (s_1, a_r) \in R \wedge a_p \mathbin{\bigcirc\!\!\!\!\!\#} a_r \wedge a_1 = a_p + a_r$$

for suitable a_p, a_r. Since $P \subseteq \mathit{dom}(C)$, there exists a transition $((s_1, a_p), (s_1, b_p)) \in C$ where $b_p = (s_1(\mathsf{i}) \mapsto [\![e]\!]_{(s_1,a_p)})|(a_p)_l + (a_p)_{L-l}$ with $s_1(\mathsf{i}) \leq {}^{\ulcorner}(a_p)_l \wedge s_1(\mathsf{i}) \neq \Box$ and $((s_1, b_p), (s_1, b_p)) \in Q$.

We assume $a_p \mathbin{\bigcirc\!\!\!\!\!\#} a_r$ and show $b_p \mathbin{\bigcirc\!\!\!\!\!\#} a_r$. By bilinearity of $\bigcirc\!\!\!\!\!\#$ we have

$$b_p \mathbin{\bigcirc\!\!\!\!\!\#} a_r \Leftrightarrow (s_1(\mathsf{i}) \mapsto [\![e]\!]_{(s_1,a_p)})|(a_p)_l \mathbin{\bigcirc\!\!\!\!\!\#} a_r \;\;\wedge\;\; (a_p)_{L-l} \mathbin{\bigcirc\!\!\!\!\!\#} a_r \,.$$

The second conjunct follows by downward closedness of $\bigcirc\!\!\!\!\!\#$ from $a_p \mathbin{\bigcirc\!\!\!\!\!\#} a_r$ while the first is equivalent to $\overline{(s_1(\mathsf{i}) + [\![e]\!]_{(s_1,a_p)})} \cdot \overline{a_r} \leq \Box \;\wedge\; \overline{(\neg s_1(\mathsf{i}) \cdot (a_p)_l)} \cdot \overline{a_r} \leq \Box$. Again the latter conjunct follows from downward closedness of $\bigcirc\!\!\!\!\!\#$. For the former we calculate $s_1(\mathsf{i}) \cdot \overline{a_r} \leq {}^{\ulcorner}a_p \cdot \overline{a_r} \leq 0$ by $a_p \mathbin{\bigcirc\!\!\!\!\!\#} a_r$ and $[\![e]\!]_{(s_1,a_p)} \cdot \overline{a_r} \leq \Box$, since C does not reference any expression of R. ◻

Lemma 8.4. *Assume the following conditions hold for command C and predicates $P, R \subseteq \mathsf{tree}$ where additionally $P \subseteq \mathit{dom}(C)$:*

$$(P \mathbin{\bigcirc\!\!\!\!\!\triangleright} R)\,;C \subseteq (P\,;C) \mathbin{\bigcirc\!\!\!\!\!\triangleright} R\;, \qquad C \mathbin{\bigcirc\!\!\!\!\!\triangleright} R \subseteq C\;.$$

Then for all predicates Q we have the $\bigcirc\!\!\!\!\!\triangleright$ frame rule

$$\frac{\{P\}\,C\,\{Q\}}{\{P \mathbin{\bigcirc\!\!\!\!\!\triangleright} R\}\,C\,\{Q \mathbin{\bigcirc\!\!\!\!\!\triangleright} R\}}\;.$$

Lemma 8.5. *The $\bigcirc\!\!\!\!\!\triangleright$ frame rule is valid for all predicates $R \subseteq \mathsf{tree}$ and commands C that do not modify any expression occurring in R and reference at most the roots of the trees in R.*

Proof. The proof is similar as for Lemma 8.3. Again we only consider selector assignments and assume a transition $((s_1, a_p), (s_1, b_p)) \in C$ with $b_p = c|(a_p)_l + (a_p)_{L-l}$ where $c =_{df} (s_1(\mathsf{i}) \mapsto [\![e]\!]_{(s_1,a_p)})$ and $l \in L$.

The assumptions on C, R induce the following conditions for c: $s_1(\mathsf{i}) \cdot \overline{a_r} \leq 0 \wedge [\![e]\!]_{(s_1,a_p)} \cdot a_r^{\urcorner} \leq \Box \wedge [\![e]\!]_{(s_1,a_p)} \cdot {}^{\ulcorner}a_r \leq \mathit{root}(a_r) \wedge \mathit{root}(a_r) \leq (\neg s_1(\mathsf{i}) \cdot (a_p)_l)^{\urcorner} + (a_p)_{L-l}^{\urcorner}$. The last conjunct but one states that at most the root of a_r is referenced by C. The last conjunct describes that the root of a_r either remains unmodified in $(a_p)_l$ or was reachable via another link $\neq l$ anyway. Assuming $a_p \triangleright a_r$ it is not difficult to show $b_p \triangleright a_r$ by similar calculations as in the proof of Lemma 8.3. ◻

Note that this property can also be extended to forests like the following one. In the present paper it is only needed for trees.

Lemma 8.6. *Assume the following conditions hold for command C and predicates $P, R, \mathit{dom}(C), \mathit{cod}(C) \subseteq \mathsf{forest}$ where additionally $P \subseteq \mathit{dom}(C)$:*

$$(R \mathbin{\bigcirc\!\!\!\!\!\triangleright} P)\,;C \subseteq R \mathbin{\bigcirc\!\!\!\!\!\triangleright} (P\,;C)\;, \qquad R \mathbin{\bigcirc\!\!\!\!\!\triangleright} C \subseteq C$$

Then for all predicates $Q \subseteq \mathsf{forest}$ we have the symmetric $\bigcirc\!\!\!\!\!\triangleright$ frame rule

$$\frac{\{P\}\,C\,\{Q\}}{\{R \mathbin{\bigcirc\!\!\!\!\!\triangleright} P\}\,C\,\{R \mathbin{\bigcirc\!\!\!\!\!\triangleright} Q\}}\;.$$

Lemma 8.7. *The symmetric $\bigcirc\!\!\!\!\!\triangleright$ frame rule is valid for all predicates $R \subseteq \mathsf{forest}$ and commands C that do not modify and reference any expression occurring in R and do not delete the root of any tree in P.*

Proof. The proof is similar as for Lemma 8.3. We consider selector assignments and assume a subexecution $((s_1, a_p), (s_1, b_p)) \in C$. By assumption $a_r \triangleright a_p$ the command C either modifies a trees $t_p =_{df} (a_p)_j$ for which there exists another tree $t_r =_{df} (a_r)_i$ with $t_r \triangleright t_p$ or C modifies a disjoint tree t_p with $t_p \circledast t_r$ for arbitrary trees $t_r \subseteq a_r$.

Again we set $b_p = c|(t_p)_l + (t_p)_{L-l}$ with $c =_{df} (s_1(\mathsf{i}) \mapsto [\![e]\!]_{(s_1,a_p)})$ and $l \in L$. We assume the following conditions for c: $\ulcorner t_r \cdot \ulcorner c \urcorner \leq 0 \wedge t_r^{\urcorner} \cdot [\![e]\!]_{(s_1,a_p)} \leq \square \wedge t_r^{\urcorner} \cdot s_1(\mathsf{i}) \leq \mathit{root}(t_p) \wedge \mathit{root}(t_p) = \mathit{root}(b_p)$. The last conjunct states that the root in t_p remains the same in b_p, i.e., it was not deleted.

Now, assuming $t_r \triangleright t_p$ one can again show $t_r \triangleright b_p$. Moreover by definition of selector assignments, we have $s_1(\mathsf{i}) \leq t_p$. Together with $t_r \circledast t_p$ this implies that $t_r^{\urcorner} \cdot s_1(\mathsf{i}) = 0$. By this, it is not difficult to prove $t_r \circledast b_p$. $\square$

9. Examples

In this section we present the new operations and predicates in action by means of some examples.

9.1. List Reversal

This example is mainly intended to show the basic ideas of our approach. The algorithm is well known. It uses variables $\mathsf{i}, \mathsf{j}, \mathsf{k}$. The initial list is headed in i, while j heads the gradually accumulated result list. Finally, k is an auxiliary variable that remembers single list nodes while they are transferred from the original list to the result list:

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

To prove functional correctness of in-situ reversal we introduce the concept of *abstraction functions* [17]. They are used, e.g., to state invariant properties.

Definition 9.1. Assume $a \in \mathsf{list}$ and an atom $p \in \ulcorner a \urcorner$. We define the abstraction function li_a w.r.t. a which collects the nodes of the sublist of a starting in node p in a word consisting of these nodes in traversal order. Moreover, we define the semantics of the expression $\mathsf{i}^{\rightarrow}$ for a program identifier i:

$$li_a(p) =_{df} \begin{cases} \langle\rangle & \text{if } p \cdot \ulcorner a \leq \square\ , \\ \langle p \rangle \bullet\ li_a(\langle a|\, p) & \text{otherwise}\ , \end{cases} \qquad [\![\mathsf{i}^{\rightarrow}]\!]_{(s,a)} =_{df} li_a(s(\mathsf{i}))\,. \tag{9}$$

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

Now using Hoare logic proof rules for variable assignment and while-loops, we can provide a full correctness proof of the in-situ list reversal algorithm. As our invariant predicate of the algorithm we use $\mathrm{I} \Leftrightarrow_{df} (\mathsf{j}^{\rightarrow})^{\dagger} \bullet \mathsf{i}^{\rightarrow} = \alpha$, where $\_^{\dagger}$ denotes word reversal. Its set-based semantics is defined by $(s, a) \in \mathrm{I} \Leftrightarrow [\![(\mathsf{j}^{\rightarrow})^{\dagger} \bullet \mathsf{i}^{\rightarrow}]\!]_{(s,a)} = \alpha$ where α represents a word. For this example we assume $L = \{\mathsf{next}\}$.

$$\begin{array}{l}
\{\ \mathsf{list}(\mathsf{i}) \wedge \mathsf{i}^{\rightarrow} = \alpha\ \} \\
\ \mathsf{j} := \square\,; \\
\{\ \mathsf{list}(\mathsf{i}) \circledast \mathsf{list}(\mathsf{j}) \wedge \mathrm{I}\ \} \\
\ \mathsf{while}\ (\mathsf{i} \neq \square)\ \mathsf{do}\ \big(\\
\quad \{\ (\mathsf{cell}(\mathsf{i}) \rhd \mathsf{list}) \circledast \mathsf{list}(\mathsf{j}) \wedge \mathrm{I}\ \} \\
\quad\ \mathsf{k} := \mathsf{i.next}\,; \\
\quad \{\ (\mathsf{cell}(\mathsf{i}) \rhd \mathsf{list}(\mathsf{k})) \circledast \mathsf{list}(\mathsf{j}) \wedge (\mathsf{j}^{\rightarrow})^{\dagger} \bullet \mathsf{i} \bullet \mathsf{k}^{\rightarrow} = \alpha\ \} \\
\quad \{\ (\mathsf{cell}(\mathsf{i}) \rhd \mathsf{list}(\mathsf{k})) \circledast \mathsf{list}(\mathsf{j}) \wedge (\mathsf{i} \bullet \mathsf{j}^{\rightarrow})^{\dagger} \bullet \mathsf{k}^{\rightarrow} = \alpha\ \} \\
\quad\ \mathsf{i.next} := \mathsf{j}\,; \\
\quad \{\ (\mathsf{cell}(\mathsf{i}) \rhd \mathsf{list}(\mathsf{j})) \circledast \mathsf{list}(\mathsf{k}) \wedge (\mathsf{i} \bullet \mathsf{j}^{\rightarrow})^{\dagger} \bullet \mathsf{k}^{\rightarrow} = \alpha\ \} \\
\quad \{\ \mathsf{list}(\mathsf{i}) \circledast \mathsf{list}(\mathsf{k}) \wedge (\mathsf{i}^{\rightarrow})^{\dagger} \bullet \mathsf{k}^{\rightarrow} = \alpha\ \} \\
\quad\ \mathsf{j} := \mathsf{i}\,; \mathsf{i} := \mathsf{k}\,; \\
\quad \{\ \mathsf{list}(\mathsf{j}) \circledast \mathsf{list}(\mathsf{i}) \wedge \mathrm{I}\ \} \\
\big) \\
\{\ \mathsf{list}(\mathsf{j}) \wedge (\mathsf{j}^{\rightarrow})^{\dagger} = \alpha\ \} \\
\{\ \mathsf{list}(\mathsf{j}) \wedge \mathsf{j}^{\rightarrow} = \alpha^{\dagger}\ \}
\end{array}$$

Each assertion consists of a structural part and a part connecting the concrete and abstract levels of reasoning. The same pattern will also occur in the example algorithms of the following sections.

Compared to [10] we hide in the $\triangleright$ operator 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 and predicates. Quantifiers to state functional correctness are not needed due to the use of the abstraction function. Hence the formulas become easier to read and more concise.

For a variant (inspired by [18]), if one would, e.g., exchange the first two commands in the while loop of the list reversal algorithm, it could possibly leave a memory leak. It can be seen that after the assignment i.next := j one would get in the postcondition as the structural part the formula $(\mathsf{cell}\,(\mathsf{i}) \triangleright \mathsf{list}\,(\mathsf{j})) \circledast \mathsf{list}$. The list memory part separated out by the second argument of $\circledast$ can neither be reached from i nor from j. Moreover, there is no program variable containing a reference to the root of that part.

9.2. Tree Rotation

As already mentioned, for binary trees we use the selector names left and right. We set $L = \{\mathsf{left}, \mathsf{right}\}$ and $a =_{df} a_{\mathsf{left}} + a_{\mathsf{right}}$.

To define an abstraction function $^{\leftrightarrow}$ similar to the $^{\rightarrow}$ function in Equation (9), we view abstract trees as being inductively defined: An *abstract tree* is either the empty tree $\langle\rangle$ or it is a triple $\langle T_l, p, T_r\rangle$, consisting of an atomic test p that represents the root node and abstract trees T_l, T_r, the left and right subtrees, resp. Now we set

$$tr_a(p) =_{df} \begin{cases} \langle\rangle & \text{if } p \cdot \ulcorner a \leq \square\ , \\ \langle tr_a(\langle a_{\mathsf{left}} | \, p), p, tr_a(\langle a_{\mathsf{right}} | \, p)\rangle & \text{otherwise}\ , \end{cases} \qquad (10)$$
$$[\![\mathsf{i}^{\leftrightarrow}]\!]_{(s,a)} =_{df} tr_a(s(\mathsf{i}))\ .$$

For a concrete example, we now present the correctness proof of an algorithm for tree rotation as known from the data structure of AVL trees. The algorithms starts with the left tree in the following Figure 4 and ends with the rotated one on the right.

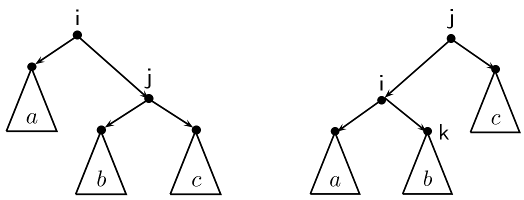


Figure 4: Tree rotation at the beginning and at the end

Using our basic tree predicates a formula for the left tree of Figure 4 would read

$$\mathsf{cell}\,(\mathsf{i}) \triangleright (\mathsf{tree}(\mathsf{i.left}) \circledast (\mathsf{cell}\,(\mathsf{i.right}) \triangleright (\mathsf{tree}(\mathsf{i.right.left}) \circledast \mathsf{tree}(\mathsf{i.right.right}))))\ . \qquad (11)$$

Unfortunately, this formula is hard to read and difficult to understand. To overcome this issue we define some auxiliary predicates that will make the assertions easier to read and more concise. The resulting formulas will exactly describe the required components of the considered tree.

Concretely for trees we set

$$\begin{aligned} \mathsf{left\_tree\_context}(\mathsf{i}) &=_{df} \mathsf{cell}\,(\mathsf{i}) \triangleright \mathsf{tree}(\mathsf{i.right})\ , \\ \mathsf{right\_tree}(\mathsf{i}) &=_{df} \mathsf{left\_tree\_context}(\mathsf{i}) \cap (\mathsf{i.left} = \square)\ , \\ \mathsf{right\_tree\_context}(\mathsf{i}) &=_{df} \mathsf{cell}\,(\mathsf{i}) \triangleright \mathsf{tree}(\mathsf{i.left})\ , \\ \mathsf{left\_tree}(\mathsf{i}) &=_{df} \mathsf{right\_tree\_context}(\mathsf{i}) \cap (\mathsf{i.right} = \square)\ . \end{aligned}$$

By this we can transform Formula (11) using Lemma 7.3 into

$$\mathsf{right\_tree\_context}(\mathsf{i}) \triangleright (\mathsf{left\_tree\_context}(\mathsf{i.right}) \triangleright \mathsf{tree}(\mathsf{i.right.right}))\ . \qquad (12)$$

We now give a "clean" version of the tree rotation algorithm, in which all occurring subtrees are separated. After that we will show an optimised version, however, with sharing in an intermediate state. With the above new predicates, a correctness proof reads as follows:

$\{\ \mathsf{right\_tree\_context(i)} \mathbin{\textcircled{$\triangleright$}} (\mathsf{left_tree_context(i.right)} \mathbin{\textcircled{$\triangleright$}} \mathsf{tree(i.right.left)}) \ \wedge\ \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle T_k, q, T_r\rangle\rangle\ \}$
$\mathsf{j := i.right;}$
$\{\ \mathsf{right\_tree\_context(i)} \mathbin{\textcircled{$\triangleright$}} (\mathsf{left_tree_context(i.right, j)} \mathbin{\textcircled{$\triangleright$}} \mathsf{tree(j.left)}) \wedge$
$\quad \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle T_k, q, T_r\rangle\rangle \ \wedge\ \mathsf{j}^{\leftrightarrow} = \langle T_k, q, T_r\rangle\ \}$
$\{\ (\mathsf{right\_tree\_context(i)} \mathbin{\textcircled{$\triangleright$}} \mathsf{left_tree_context(i.right, j)}) \mathbin{\textcircled{$\triangleright$}} \mathsf{tree(j.left)} \wedge$
$\quad \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle T_k, q, T_r\rangle\rangle \ \wedge\ \mathsf{j}^{\leftrightarrow} = \langle T_k, q, T_r\rangle\ \}$
$\mathsf{i.right} := \square;$
$\{\ (\mathsf{left\_tree(i)} \circledast \mathsf{left\_tree\_context(j)}) \mathbin{\textcircled{$\triangleright$}} \mathsf{tree(j.left)} \ \wedge\ \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle\rangle\rangle \ \wedge\ \mathsf{j}^{\leftrightarrow} = \langle T_k, q, T_r\rangle\ \}$
$\{\ \mathsf{left\_tree(i)} \circledast (\mathsf{left\_tree\_context(j)} \mathbin{\textcircled{$\triangleright$}} \mathsf{tree(j.left)}) \ \wedge\ \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle\rangle\rangle \ \wedge\ \mathsf{j}^{\leftrightarrow} = \langle T_k, q, T_r\rangle\ \}$
$\mathsf{k := j.left;}$
$\{\ \mathsf{left\_tree(i)} \circledast (\mathsf{left\_tree\_context(j)} \mathbin{\textcircled{$\triangleright$}} \mathsf{tree(j.left, k)}) \ \wedge\ \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle\rangle\rangle \ \wedge\ \mathsf{j}^{\leftrightarrow} = \langle T_k, q, T_r\rangle \ \wedge\ \mathsf{k}^{\leftrightarrow} = T_k\ \}$
$\mathsf{j.left} := \square;$
$\{\ \mathsf{left\_tree(i)} \circledast \mathsf{right\_tree(j)} \circledast \mathsf{tree(k)} \ \wedge\ \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle\rangle\rangle \ \wedge\ \mathsf{j}^{\leftrightarrow} = \langle\langle\rangle, q, T_r\rangle \ \wedge\ \mathsf{k}^{\leftrightarrow} = T_k\ \}$
$\mathsf{j.left := i;}$
$\{\ (\mathsf{left\_tree\_context(j)} \mathbin{\textcircled{$\triangleright$}} \mathsf{left_tree(i, j.left)}) \circledast \mathsf{tree(k)} \wedge$
$\quad \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle\rangle\rangle \ \wedge\ \mathsf{j}^{\leftrightarrow} = \langle\langle T_l, p, \langle\rangle\rangle, q, T_r\rangle \ \wedge\ \mathsf{k}^{\leftrightarrow} = T_k\ \}$
$\{\ \mathsf{left\_tree\_context(j)} \mathbin{\textcircled{$\triangleright$}} (\mathsf{left_tree(i, j.left)} \circledast \mathsf{tree(k)}) \wedge$
$\quad \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle\rangle\rangle \ \wedge\ \mathsf{j}^{\leftrightarrow} = \langle\langle T_l, p, \langle\rangle\rangle, q, T_r\rangle \ \wedge\ \mathsf{k}^{\leftrightarrow} = T_k\ \}$
$\mathsf{i.right := k;}$
$\{\ \mathsf{left\_tree\_context(j)} \mathbin{\textcircled{$\triangleright$}} (\mathsf{right_tree_context(i, j.left)} \mathbin{\textcircled{$\triangleright$}} \mathsf{tree(k, i.right)}) \wedge$
$\quad \mathsf{j}^{\leftrightarrow} = \langle\langle T_l, p, T_k\rangle, q, T_r\rangle \ \wedge\ \mathsf{i}^{\leftrightarrow} = \langle T_l, p, T_k\rangle \ \wedge\ \mathsf{k}^{\leftrightarrow} = T_k\ \}$

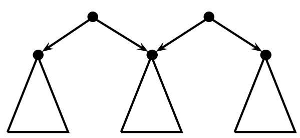
Note that the predicate $(\mathsf{i}.l = \square)$ satisfies the equation $(P(\mathsf{i}) \circledast Q) \cap (\mathsf{i}.l = \square) = (P(\mathsf{i}) \cap (\mathsf{i}.l = \square)) \circledast Q$ for $P, Q \subseteq \mathsf{tree}$. Therefore we can use Lemma 8.1 for the proof.

The next version of the algorithm uses fewer assignments, but shows sharing within an intermediate state. Its verification requires the definition of a new predicate, since one of the intermediate states cannot be described with the operators we have defined so far.

Definition 9.2. For predicates $P, R \subseteq \mathsf{forest}$ and $Q \subseteq \mathsf{tree}$ we define

$$P \mathbin{\textcircled{$\triangleright$}} Q \mathbin{\textcircled{$\triangleleft$}} R =_{df} \{\ a_1 + a_2 + a_3 :\ a_1 \in P,\ a_2 \in Q,\ a_3 \in R,\ a_1 \triangleright a_2,\ a_3 \triangleright a_2,\ \overline{a_1} \cdot \overline{a_3} = \mathit{root}(a_2)\ \}\ .$$

Clearly, $P \mathbin{\textcircled{$\triangleright$}} Q \mathbin{\textcircled{$\triangleleft$}} R = R \mathbin{\textcircled{$\triangleright$}} Q \mathbin{\textcircled{$\triangleleft$}} P$. The linked structures characterised by the predicate can be depicted as follows:



For using this predicate in a verification of our second variant of tree rotation algorithm we have the following inference rules.

Lemma 9.3. *Assume predicates $P \subseteq l\mathsf{\_tree\_context}$ and $Q, R \subseteq \mathsf{tree}$, identifiers i, j and selectors $l, m \in L$ then*

$\{\ (P(\mathsf{i}) \mathbin{\textcircled{$\triangleright$}} (Q(\mathsf{j}, \mathsf{i}.l) \mathbin{\textcircled{$\triangleright$}} R(\mathsf{j}.m))\ \}$
$\mathsf{i}.l := \mathsf{j}.m;$
$\{\ P(\mathsf{i}) \mathbin{\textcircled{$\triangleright$}} R(\mathsf{j}.m, \mathsf{i}.l) \mathbin{\textcircled{$\triangleleft$}} Q(\mathsf{j})\ \}\ ,$

$\{\ P(\mathsf{i}) \mathbin{\textcircled{$\triangleright$}} S(\mathsf{j}.m, \mathsf{i}.l) \mathbin{\textcircled{$\triangleleft$}} R(\mathsf{j})\ \}$
$\mathsf{i}.l := \mathsf{j};$
$\{\ P(\mathsf{i}) \mathbin{\textcircled{$\triangleright$}} (R(\mathsf{j}, \mathsf{i}.l) \mathbin{\textcircled{$\triangleright$}} S(\mathsf{j}.m))\ \}\ .$

The latter rule also works for $P \subseteq \mathsf{tree}$.

Proof. We outline a proof of the first rule; a proof for the second one can be obtained similarly. Assume $a = a_1 + a_2 + a_3$ with $a_i \in P(\mathsf{i}) \wedge a_2 \in Q(\mathsf{j}, \mathsf{i}.l) \wedge a_3 \in R(\mathsf{j}.m)$. We know $a_1 \triangleright (a_2 + a_3) \wedge a_2 \triangleright a_3$ and from the identifiers $s(\mathsf{i}) \mapsto [\![\mathsf{j}.m]\!]_{(s,a)} = root(a_1) \mapsto root(a_3)$.

Note that $a_1 \in P(\mathsf{i})$. This immediately implies $(root(a_1) \mapsto root(a_3))|(a_1)_l = root(a_1) \mapsto root(a_3)$ and we set $b_1 =_{df} (root(a_1) \mapsto root(a_3)) + (a_1)_{L-l}$. From the assumption we get $(a_1)_{L-l} \textcircled{\#} a_2$. Using Lemma 6.12, we also know $a_1 \textcircled{\#} a_3$ and can further infer $(a_1)_{L-l} \textcircled{\#} a_3$. Now we can conclude $b_1 \triangleright a_3 \wedge \overline{b_1} \cdot \overline{a_2} = root(a_3)$. □

By Lemma 9.3 we can verify the following shorter form of the tree rotation algorithm that uses sharing.

{ right_tree_context(i) $\circledtriangleright$ (left_tree_context(i.right) $\circledtriangleright$ tree(i.right.left)) $\wedge$ $\mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle T_k, q, T_r\rangle\rangle$ }
j := i.right;
{ right_tree_context(i) $\circledtriangleright$ (left_tree_context(i.right, j) $\circledtriangleright$ tree(j.left)) $\wedge$
$\mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle T_k, q, T_r\rangle\rangle \wedge \mathsf{j}^{\leftrightarrow} = \langle T_k, q, T_r\rangle$ }
i.right := j.left,
{ right_tree_context(i) $\circledtriangleright$ tree(j.left, i.right) $\circledtriangleleft$ left_tree_context(j) $\wedge$ $\mathsf{i}^{\leftrightarrow} = \langle T_l, p, T_k\rangle \wedge \mathsf{j}^{\leftrightarrow} = \langle T_k, q, T_r\rangle$ }
j.left := i;
{ left_tree_context(j) $\circledtriangleright$ (right_tree_context(i, j.left) $\circledtriangleright$ tree(i.right)) $\wedge$
$\mathsf{j}^{\leftrightarrow} = \langle\langle T_l, p, T_k\rangle, q, T_r\rangle \wedge \mathsf{i}^{\leftrightarrow} = \langle T_l, p, T_k\rangle \wedge \mathsf{k}^{\leftrightarrow} = T_k$ }

The third assertion, that uses the new predicate, can be depicted as in Figure 5.

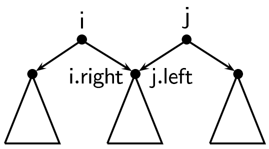


Figure 5: Tree rotation with sharing in an intermediate state

10. A Treatment for Overlaid Data Structures

To further underpin the practicality of our approach, we consider as a concrete example for the treatment of overlaid data structures so-called *threaded trees*. We consider trees where the threads enable a fast inorder traversal of the whole tree (cf. Figure 6 where the dashed lines denote threads).

First, all predicates and operations defined up to now consider non-reachability or directed reachability only on complete access elements, i.e., the operators work on all selectors. This is far too strict, especially in the case of threaded trees. As an example, $\triangleright$ completely excludes the existence of cycles in the whole tree while e.g., links and threads together might form cycles within such a tree. In Figure 6 we can directly reach a cycle from j to its successor via the thread and back via the left selector.

Hence, we need a weaker variant of $\triangleright$ that works on a specific set of links $M \subseteq L$. For a linked structure c over L we set $c_M =_{df} \sum_{l \in M} c_l$ and define

$$a \triangleright_M b \Leftrightarrow_{df} a_M \triangleright b_M$$

and its corresponding operator on predicates by

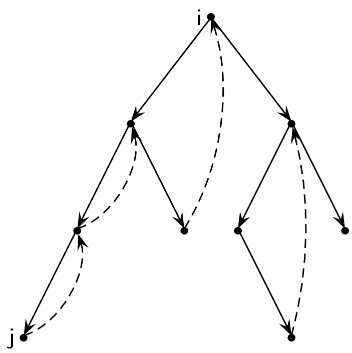


Figure 6: Example of a threaded tree

$$P \circledtriangleright_M Q =_{df} \{ a + b : a \in P,\, b \in Q,\, a \triangleright_M b \} \ .$$

We will omit the set braces when M is a singleton set.

The same generalisations apply to $\circledast$ and $\#\!\!\!\!\bigcirc$. Note that, by $M \subseteq L$ and downward closedness of $\#\!\!\!\!\bigcirc$, also $\#\!\!\!\!\bigcirc \subseteq \#\!\!\!\!\bigcirc_M$ and hence $P \circledast Q \subseteq P \circledast_M Q$. Note that our laws for $\#\!\!\!\!\bigcirc$ and $\rhd\!\!\!\!\bigcirc$ hold also for $\#\!\!\!\!\bigcirc_M$ and $\rhd\!\!\!\!\bigcirc_M$, resp., assuming a set of links $M \subseteq L$.

For a threaded three we define the access relation by $a = a_{\mathsf{left}} + a_{\mathsf{right}} + a_{\mathsf{marked}}$, i.e., $L = \{\mathsf{left}, \mathsf{right}, \mathsf{marked}\}$. Clearly the access elements a_{left} and a_{right} need to be disjoint, while a_{marked} is a test with $a_{\mathsf{marked}} \leq \ulcorner a_{\mathsf{right}}$. It represents a set of nodes from which threads emanate, i.e., where the right links represent pointers from the respective node to its successor in the inorder traversal of the corresponding unthreaded tree. In addition, we require the following structural properties of a:

1. $a_{\mathsf{LR}} =_{df} a_{\mathsf{left}} + \neg a_{\mathsf{marked}} \cdot a_{\mathsf{right}}$ forms a tree;
2. $a_{\mathsf{thread}} =_{df} a_{\mathsf{marked}} \cdot a_{\mathsf{right}} + a_{\mathsf{RLm}}$, where $a_{\mathsf{RLm}} =_{df} (\neg a_{\mathsf{marked}} \cdot a_{\mathsf{right}}) \cdot a^*_{\mathsf{left}} \cdot \neg \ulcorner a_{\mathsf{left}}$, forms a chain;
3. the inorder sequence of the a_{LR} equals the traversal sequence of a_{thread}.

The element a_{RLm} connects a non-marked node x, i.e., a node without any threads, with the leftmost node in the right subtree of x, i.e., its successor node in the inorder traversal. The subexpression $a^*_{\mathsf{left}} \cdot \neg \ulcorner a_{\mathsf{left}}$ occurring in a_{RLm} is an algebraic representation of the loop $\mathsf{while}\, \ulcorner a_{\mathsf{left}}\, \mathsf{do}\, a_{\mathsf{left}}$. It has been shown in [19] that determinacy of a loop body is inherited by the corresponding while loop.

Note that a_{thread} is a virtual access relation, i.e., its selector thread is not in L, but it is formed using selectors of L.

Next, we relax the definition for some predicates, so that they take the new linked structures into account:

$$
\begin{aligned}
\mathsf{u\_cell} &=_{df} \{a : a_{\mathsf{LR}} \text{ is a cell, } a_{\mathsf{marked}} \leq 0\}\,, \\
\mathsf{m\_cell} &=_{df} \{a : a_{\mathsf{LR}} \text{ is a cell, } a_{\mathsf{marked}} = \mathit{root}(a)\}\,, \\
\mathsf{thread\_list} &=_{df} \{a : a_{\mathsf{thread}} \text{ is a chain}\}\,, \\
\mathsf{lr\_tree} &=_{df} \{a : a_{\mathsf{LR}} \text{ is a tree}\}\,.
\end{aligned}
$$

The predicate $\mathsf{u\_cell}$ characterises unmarked cells while cells in $\mathsf{m\_cell}$ are marked. This is realised by setting its marked component to its root. Moreover, the predicate $\mathsf{thread\_list}$ is restricted to all marked right selectors and connections from unmarked nodes to left-most nodes while $\mathsf{lr\_tree}$ considers only the left and unmarked right selectors. We further define

$$
[\![\mathsf{j}^{\rightarrow}]\!]_{(s,a)} =_{df} li_{a_{\mathsf{thread}}}(s(\mathsf{j})) \quad \text{and} \quad [\![\mathsf{i}^{\rightsquigarrow}]\!]_{(s,a)} =_{df} \mathit{inorder}(tr_{a_{\mathsf{LR}}}(s(\mathsf{i}))) \tag{13}
$$

where $tr_a(p)$ for a tree a is defined in Equation (10) and $\mathit{inorder}(T)$ returns the word consisting of the nodes of T in the sequence of an inorder traversal of T.

A threaded tree can now defined by the predicate

$$
\mathsf{th\_tree}(\mathsf{i},\mathsf{j}) =_{df} \mathsf{lr\_tree}(\mathsf{i}) \wedge \mathsf{thread\_list}(\mathsf{j}) \wedge \mathsf{j}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow}
$$

where i points to the root of the underlying tree and j points to the head of the list formed by a_{thread} (cf. Figure 6). Note that $\mathsf{j}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow}$ implies that $\mathsf{j} = \mathsf{leftmost}(\mathsf{i})$ where

$$
lm_a(p) =_{df} \begin{cases} \square & \text{if } p = \square\,, \\ p & \text{if } (\langle a_{\mathsf{left}} | \, p) \cdot \ulcorner a = 0\,, \\ lm_a(\langle a_{\mathsf{left}} | \, p) & \text{otherwise}\,, \end{cases} \qquad [\![\mathsf{leftmost}(\mathsf{i})]\!]_{(s,a)} =_{df} lm_a(s(\mathsf{i}))\,.
$$

Next, we give a verification example and therefore sum up a few consequences.

Lemma 10.1. *Assume predicates* $P, Q \subseteq \mathsf{tree}$ *and identifiers* i, j. *Moreover assume selector sets* $K, M \subseteq L$

and a selector $l \in K - M$*. Then*

$$\{\ P(\mathsf{i}) \circledast_K Q(\mathsf{j})\ \}\quad \mathsf{i}.l := \mathsf{j};\quad \{\ P(\mathsf{i}) \circledast_{K-l} Q(\mathsf{j}) \wedge P(\mathsf{i}) \rhd_l Q(\mathsf{j}, \mathsf{i}.l)\ \}\ ,$$

$$\{\ P(\mathsf{i}) \rhd Q(\mathsf{j})\ \}\quad \mathsf{i}.l := \mathsf{j};\quad \{\ P(\mathsf{i}) \rhd Q(\mathsf{j}) \wedge P(\mathsf{i}) \rhd_l Q(\mathsf{j}, \mathsf{i}.l)\ \}$$

and

$$\{\ P(\mathsf{i}) \rhd_M Q(\mathsf{j})\ \}\quad \mathsf{j}.l := \mathsf{i};\quad \{\ P(\mathsf{i}) \rhd_M Q(\mathsf{j}) \wedge Q(\mathsf{j}) \rhd_l P(\mathsf{i}, \mathsf{j}.l)\ \}\ .$$

Proofs for these rules can be constructed similar to that of Lemma 9.3.

All rules make use of the generalised operators. The first rule describes that after the selector assignment P and Q remain strongly disjoint on all selectors in $K - l$ while it is now possible to reach Q from P via l. This is similarly mimicked in the second rule. It describes that Q is reachable from P; especially one can use the selector l to reach Q from P. The third rule describes that all links from P to Q mentioned in the precondition will remain unchanged by assigning via a selector $l \notin M$.

Note that these rules also extend to forests but suffice in this form for the present paper.

To mark nodes we define a command that appropriately sets the marked selector of the considered access elements and redefine allocation of nodes to ignore the marked selector:

$$\begin{array}{rcl} \mathsf{mark(i)} & =_{df} & \{ ((s,a),(s,(s(\mathsf{i}) + a_{\mathsf{marked}}) + a_{L-\mathsf{marked}})) : \mathsf{i} \in dom(s) \}\ , \\ \mathsf{i} := \mathsf{new\,cell}\,() & =_{df} & \{ ((s,a),(s[\mathsf{i} \leftarrow p],(p \mapsto \Box)|a_{L-\mathsf{marked}} + a_{\mathsf{marked}}) : \mathsf{i} \in dom(s), \\ & & \quad p \text{ is an atomic test, } p \leq \neg\ulcorner a,\ p \neq \Box \}\ . \end{array}$$

Before we can use it in the verification of the concrete example we give further inference rules.

Lemma 10.2. *Assume identifiers* $\mathsf{i}, \mathsf{j}, \mathsf{k}$ *and* $\mathsf{i} \neq \Box \wedge \mathsf{k} \neq \Box$ *then*

$$\{\ \mathsf{th\_tree}(\mathsf{i},\mathsf{j}) \circledast \mathsf{u\_cell}(\mathsf{k})\ \}\quad \mathsf{j.left} := \mathsf{k};\quad \{\ (\mathsf{lr\_tree}(\mathsf{i}) \rhd_{\mathsf{LR}} \mathsf{u\_cell}(\mathsf{k})) \wedge \mathsf{thread\_list}(\mathsf{j}) \wedge \mathsf{k} \bullet \mathsf{j}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow}\ \}\ ,$$

$$\{\ \mathsf{u\_cell}(\mathsf{k})\ \}\quad \mathsf{mark(k)};\quad \{\ \mathsf{m\_cell}(\mathsf{k})\ \}$$

and

$$\{\ \mathsf{lr\_tree}(\mathsf{i}) \wedge (\mathsf{u\_cell}(\mathsf{k}) \rhd_{\mathsf{right}} \mathsf{thread\_list}(\mathsf{j}, \mathsf{k.right})) \wedge \mathsf{k} \bullet \mathsf{j}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow}\ \}\quad \mathsf{mark(k)};\quad \{\ \mathsf{lr\_tree}(\mathsf{i}) \wedge (\mathsf{m\_cell}(\mathsf{k}) \rhd_{\mathsf{thread}} \mathsf{thread\_list}(\mathsf{j}, \mathsf{k.right})) \wedge \mathsf{k}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow}\ \}\ .$$

These laws are direct consequences of the definition of mark and the abstraction functions in Equation (13). The first rule expresses that after making k the left subtree of j the inorder list of the resulting overall tree now starts with k and continues with that headed by j. The meaning of the second rule is obvious. The third rule states that after marking the right-link of k must be interpreted as a thread link, so that the thread list is now headed by k.

We can now give another verification example to view the new predicates and operators in action. For simplicity, we do not treat balancing so that we can simply add a new node as the left subtree of the leftmost node. We assume a non-empty threaded tree with root in i and $\mathsf{j} \neq \mathsf{i}$ heading the thread list. Then we can reason as follows.

$$
\begin{array}{l}
\{\ \mathsf{lr\_tree}\,(\mathsf{i}) \circledtriangleright_{\mathsf{LR}} \mathsf{u\_cell}\,(\mathsf{j}) \ \wedge\ \mathsf{thread\_list}\,(\mathsf{j}) \ \wedge\ \mathsf{j}^{\rightarrow} = \mathsf{i}^{\leadsto} \ \wedge\ \mathsf{j}^{\rightarrow} = \alpha \ \}\\
\ \mathsf{k} := \mathsf{new\ cell}\,();\\
\{\ (\mathsf{lr\_tree}\,(\mathsf{i}) \circledtriangleright_{\mathsf{LR}} \mathsf{u\_cell}\,(\mathsf{j})) \circledast \mathsf{u\_cell}\,(\mathsf{k}) \ \wedge\ \mathsf{thread\_list}\,(\mathsf{j}) \circledast \mathsf{u\_cell}\,(\mathsf{k}) \ \wedge\ \mathsf{j}^{\rightarrow} = \mathsf{i}^{\leadsto} \ \wedge\ \mathsf{j}^{\rightarrow} = \alpha \ \}\\
\ \mathsf{j.left} := \mathsf{k},\\
\{\ \mathsf{lr\_tree}\,(\mathsf{i}) \circledtriangleright_{\mathsf{LR}}(\mathsf{u\_cell}\,(\mathsf{j}) \circledtriangleright_{\mathsf{LR}} \mathsf{u\_cell}\,(\mathsf{k},\mathsf{j.left})) \ \wedge\ \mathsf{thread\_list}\,(\mathsf{j}) \circledast_{\mathsf{right}} \mathsf{u\_cell}\,(\mathsf{k},\mathsf{j.left}) \wedge\\
\quad \mathsf{k} \bullet \mathsf{j}^{\rightarrow} = \mathsf{i}^{\leadsto} \ \wedge\ \mathsf{j}^{\rightarrow} = \alpha \ \}\\
\ \mathsf{k.right} := \mathsf{j};\\
\{\ \mathsf{lr\_tree}\,(\mathsf{i}) \circledtriangleright_{\mathsf{LR}}(\mathsf{u\_cell}\,(\mathsf{j}) \circledtriangleright_{\mathsf{LR}} \mathsf{u\_cell}\,(\mathsf{k},\mathsf{j.left})) \ \wedge\ \mathsf{u\_cell}\,(\mathsf{k}) \circledtriangleright_{\mathsf{right}} \mathsf{thread\_list}\,(\mathsf{j},\mathsf{k.right}) \wedge\\
\quad \mathsf{k} \bullet \mathsf{j}^{\rightarrow} = \mathsf{i}^{\leadsto} \ \wedge\ \mathsf{k} \bullet \mathsf{j}^{\rightarrow} = \mathsf{k} \bullet \alpha \ \}\\
\ \mathsf{mark}(\mathsf{k});\\
\{\ \mathsf{lr\_tree}\,(\mathsf{i}) \circledtriangleright_{\mathsf{LR}}(\mathsf{u\_cell}\,(\mathsf{j}) \circledtriangleright_{\mathsf{LR}} \mathsf{m\_cell}\,(\mathsf{k},\mathsf{j.left})) \ \wedge\ \mathsf{m\_cell}\,(\mathsf{k}) \circledtriangleright_{\mathsf{thread}} \mathsf{thread\_list}\,(\mathsf{j},\mathsf{k.right})\\
\quad \wedge\, \mathsf{k}^{\rightarrow} = \mathsf{i}^{\leadsto} \ \wedge\ \mathsf{k}^{\rightarrow} = \mathsf{k} \bullet \alpha \ \}\\
\{\ \mathsf{lr\_tree}\,(\mathsf{i}) \ \wedge\ \mathsf{thread\_list}\,(\mathsf{k}) \ \wedge\ \mathsf{k}^{\rightarrow} = \mathsf{i}^{\leadsto} \ \wedge\ \mathsf{k}^{\rightarrow} = \mathsf{k} \bullet \alpha \ \}\\
\ \mathsf{j} := \mathsf{k};\\
\{\ \mathsf{lr\_tree}\,(\mathsf{i}) \ \wedge\ \mathsf{thread\_list}\,(\mathsf{j}) \ \wedge\ \mathsf{j}^{\rightarrow} = \mathsf{i}^{\leadsto} \ \wedge\ \mathsf{j}^{\rightarrow} = \mathsf{k} \bullet \alpha \ \}
\end{array}
$$

We conclude this section by sketching a similar idea for treating doubly linked lists. An adequate access relation can be defined by $a = a_{\mathsf{next}} + a_{\mathsf{prev}}$. The characterising predicate for this data structure then reads

$$\mathsf{dl\_list}(\mathsf{i},\mathsf{j}) \ =_{df}\ \mathsf{next\_list}\,(\mathsf{i}) \ \wedge\ \mathsf{prev\_list}\,(\mathsf{j}) \ \wedge\ \mathsf{i}^{\rightarrow} = ({}^{\leftarrow}\mathsf{j})^{\dagger}$$

where

$$\mathsf{next\_list} \ =_{df}\ \{a : a_{\mathsf{next}} \text{ is a chain}\}\,, \qquad \mathsf{prev\_list} \ =_{df}\ \{a : a_{\mathsf{prev}} \text{ is a chain}\}\,.$$

and

$$[\![\mathsf{i}^{\rightarrow}]\!]_{(s,a)} \ =_{df}\ li_{a_{\mathsf{next}}}(s(\mathsf{j}))\,, \qquad [\![{}^{\leftarrow}\mathsf{j}]\!]_{(s,a)} \ =_{df}\ li_{a_{\mathsf{prev}}}(s(\mathsf{j}))\ .$$

11. 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. [20] defined an extension called *Confined Separation Logic* and provided a relational model for it. They defined various 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 [18]. Generally, there heaps are viewed as labelled object graphs. Starting from an abstract foundation the authors define a decidable logic, e.g. for lists, with domain-specific predicates and operations suitable for automated reasoning.

By contrast, our approach enables abstract derivations in a largely first-order algebraic approach, called pointer Kleene algebra [12]. The given simple (in-)equational laws allow a direct usage of automated theorem proving systems as Prover9 [21] or any other systems through the TPTP Library [22] at the level of the underlying resource algebra [23]. This supports and helpfully guides the development of domain specific predicates and operations. The assertions we have presented are simple and still suitable for expressing shapes of linked structures without the need of any arithmetic as in [18]. Part of such assertions can be automatically verified using Smallfoot [24].

A novel approach to sharing in data structures can be found in [25]. This approach can be directly used with arbitrary separation logics and introduces, differing from our approach, an operation called overlapping conjunction. This operator in contrast to the separating conjunction allows unspecified overlapping of the resources characterised by predicates. It enables impressive reasoning about sharing in combination with the separating implication. However, the formulas involved unfortunately become very complex and difficult to understand. We hope that the approach of the present paper can also capture complex examples like the garbage collecting algorithm given in [25] with easier and more concise formulas.

12. 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 about 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. Finally, we combined the obtained results in a treatment for threaded trees and presented the predicates and operators in a verification of an element insertion algorithm on such trees.

For future work, it will be interesting to explore more complex object structures and verify garbage collecting algorithms like the *Schorr-Waite Graph Marking* or treat concurrent garbage collection algorithms.

Acknowledgements: We thank reviewers of RAMiCS 2012 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.*

13. Appendix: Proofs

Proof of Lemma 3.3.

1. First, ${}^{\ulcorner}a \leq \mathit{reach}({}^{\ulcorner}a, a)$ by the reach induction rule from Section 2.
   Second, by a domain property, $a^{\urcorner} = ({}^{\ulcorner}a \cdot a)^{\urcorner} = \langle a|\, {}^{\ulcorner}a \leq \mathit{reach}({}^{\ulcorner}a, a)$.

2. For ($\leq$) we know by diamond star induction that $\mathit{reach}({}^{\ulcorner}a, a+b) \leq \ulcorner a \urcorner \Leftarrow {}^{\ulcorner}a \leq \ulcorner a \urcorner \wedge \langle (a+b)|\, \ulcorner a \urcorner \leq \ulcorner a \urcorner$. ${}^{\ulcorner}a \leq \ulcorner a \urcorner$ holds by definition of $\ulcorner\,\urcorner$, while $\langle (a+b)|\, \ulcorner a \urcorner \leq \ulcorner a \urcorner$ resolves by diamond distributivity to $\langle a|\, \ulcorner a \urcorner \leq \ulcorner a \urcorner \wedge \langle b|\, \ulcorner a \urcorner \leq \ulcorner a \urcorner$. Finally, the claim holds by $(\ulcorner a \urcorner \cdot a)^{\urcorner} \leq a^{\urcorner}$ and the assumption. The direction ($\geq$) follows from Part 1, $a \leq a+b$ and isotony of *reach*. □

Proof of Lemma 5.4.

We first show the auxiliary result

$$p \leq a^{\urcorner} \wedge |a\rangle\, p = 0 \Rightarrow p = 0 \ . \tag{14}$$

We have, by the definition of diamond, full strictness of domain and (gra),

$$|a\rangle\, p = 0 \Leftrightarrow {}^{\ulcorner}(a \cdot p) = 0 \Leftrightarrow a \cdot p = 0 \Leftrightarrow p \leq \neg a^{\urcorner} \ .$$

Since by assumption $p \leq a^{\urcorner}$, we get $p \leq a^{\urcorner} \cdot \neg a^{\urcorner} = 0$.

Now we continue with the proof of Lemma 5.4. Suppose $a^{\urcorner} = 0$. Then by full strictness also $a = 0$ and hence ${}^{\ulcorner}a = 0$, contradicting atomicity of ${}^{\ulcorner}a$. Hence $a^{\urcorner} \neq 0$.

Now assume $p \leq a^{\urcorner} \wedge p \neq 0$. By Equation 14 we have $0 \neq |a\rangle\, p = {}^{\ulcorner}(a \cdot p) \leq {}^{\ulcorner}a$. Hence, atomicity of ${}^{\ulcorner}a$ implies $|a\rangle\, p = {}^{\ulcorner}a$. Now, by definition of codomain and determinacy of a,

$$a^{\urcorner} = \langle a|\, {}^{\ulcorner}a = \langle a|\, |a\rangle\, p \leq p \ ,$$

so that altogether we have $p = a^{\urcorner}$, which, by the assumptions and the definition of atomicity, shows the claim. □

Proof of Equation (11) $\Leftrightarrow$ Equation (12):

$$
\begin{array}{ll}
& \mathsf{cell}\,(\mathsf{i}) \rhd (\mathsf{tree}(\mathsf{i.left}) \circledast (\mathsf{cell}\,(\mathsf{i.right}) \rhd (\mathsf{tree}(\mathsf{i.right.left}) \circledast \mathsf{tree}(\mathsf{i.right.right})))) \\
= & \quad \{\!\![\text{ Lemma 7.3 (7) }]\!\!\} \\
& (\mathsf{cell}\,(\mathsf{i}) \rhd \mathsf{tree}(\mathsf{i.left})) \rhd (\mathsf{cell}\,(\mathsf{i.right}) \rhd (\mathsf{tree}(\mathsf{i.right.left}) \circledast \mathsf{tree}(\mathsf{i.right.right}))) \\
= & \quad \{\!\![\text{ definition of } \mathsf{right\_tree\_context}]\!\!\} \\
& \mathsf{right\_tree\_context}(\mathsf{i}) \rhd (\mathsf{cell}\,(\mathsf{i.right}) \rhd (\mathsf{tree}(\mathsf{i.right.left}) \circledast \mathsf{tree}(\mathsf{i.right.right}))) \\
= & \quad \{\!\![\text{ commutativity of } \circledast]\!\!\} \\
& \mathsf{right\_tree\_context}(\mathsf{i}) \rhd (\mathsf{cell}\,(\mathsf{i.right}) \rhd (\mathsf{tree}(\mathsf{i.right.right}) \circledast \mathsf{tree}(\mathsf{i.right.left}))) \\
= & \quad \{\!\![\text{ Lemma 7.3 (7) }]\!\!\} \\
& \mathsf{right\_tree\_context}(\mathsf{i}) \rhd ((\mathsf{cell}\,(\mathsf{i.right}) \rhd \mathsf{tree}(\mathsf{i.right.right})) \rhd \mathsf{tree}(\mathsf{i.right.left})) \\
= & \quad \{\!\![\text{ definition of } \mathsf{left\_tree\_context}]\!\!\} \\
& \mathsf{right\_tree\_context}(\mathsf{i}) \rhd (\mathsf{left\_tree\_context}(\mathsf{i.right}) \rhd \mathsf{tree}(\mathsf{i.right.left}))
\end{array}
$$

The same calculation can be done for the final state, i.e., the equation

$$\mathsf{cell}\,(\mathsf{j}) \rhd ((\mathsf{cell}\,(\mathsf{i}, \mathsf{j.left}) \rhd (\mathsf{tree}(\mathsf{i.left}) \circledast \mathsf{tree}(\mathsf{k}, \mathsf{i.right}))) \circledast \mathsf{tree}(\mathsf{j.right}))$$

equals the following

$$\mathsf{left\_tree\_context}(\mathsf{j}) \rhd (\mathsf{right\_tree\_context}(\mathsf{i}, \mathsf{j.left}) \rhd \mathsf{tree}(\mathsf{k}, \mathsf{i.right}))\ .$$

□

References

[1] H.-H. Dang, B. Möller, Transitive Separation Logic, in: W. Kahl, T. G. Griffin (Eds.), 13th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 13), volume 7560 of *LNCS*, Springer, 2012, pp. 1–16.

[2] J. Desharnais, B. Möller, G. Struth, Kleene algebra with domain, ACM Transactions on Computational Logic 7 (2006) 798–833.

[3] T. Ehm, Pointer Kleene algebra, in: R. Berghammer, B. Möller, G. Struth (Eds.), RelMiCS/AKA 2003, volume 3051 of *LNCS*, Springer, 2004, pp. 99–111.

[4] P. Höfner, G. Struth, Automated reasoning in Kleene algebra, in: F. Pfenning (Ed.), Automated Deduction — CADE-21, volume 4603 of *Lecture Notes in Artificial Intelligence*, Springer, 2007, pp. 279–294.

[5] E. Manes, D. Benson, The inverse semigroup of a sum-ordered semiring, Semigroup Forum 31 (1985) 129–152.

[6] D. Kozen, Kleene algebra with tests, ACM Transactions on Programming Languages and Systems 19 (1997) 427–443.

[7] D. Kozen, A completeness theorem for Kleene algebras and the algebra of regular events, Information and Computation 110 (1994) 366–390.

[8] B. Möller, Calculating with acyclic and cyclic lists, Information Sciences 119 (1999) 135–154.

[9] C. A. R. Hoare, B. Möller, G. Struth, I. Wehrman, Foundations of concurrent Kleene algebra, in: R. Berghammer, A. Jaoua, B. Möller (Eds.), Relations and Kleene Algebra in Computer Science, volume 5827 of *LNCS*, Springer, 2009, pp. 166–186.

[10] J. C. Reynolds, An introduction to separation logic, in: M. Broy (Ed.), In Engineering Methods and Tools for Software Safety and Security, IOS Press, 2009, pp. 285–310.

[11] E.-J. Sims, Extending Separation Logic with Fixpoints and Postponed Substitution, Theoretical Computer Science 351 (2006) 258–275.

[12] T. Ehm, The Kleene algebra of nested pointer structures: Theory and applications, PhD Thesis, `http://www.opus-bayern.de/uni-augsburg/frontdoor.php?source_opus=89`, 2003.

[13] B. Möller, Some applications of pointer algebra, in: M. Broy (Ed.), Programming and Mathematical Method, number 88 in NATO ASI Series, Series F: Computer and Systems Sciences, Springer, 1992, pp. 123–155.

[14] J. Desharnais, B. Möller, Characterizing determinacy in Kleene algebra, Information Sciences 139 (2001) 253–273.

[15] H.-H. Dang, P. Höfner, B. Möller, Algebraic separation logic, Journal of Logic and Algebraic Programming 80 (2011) 221–247.

[16] H.-H. Dang, B. Möller, Reverse Exchange for Concurrency and Local Reasoning, in: J. Gibbons, P. Nogueira (Eds.), MPC, volume 7342 of *LNCS*, Springer, 2012, pp. 177–197.

[17] C. A. R. Hoare, Proofs of correctness of data representations, Acta Informatica 1 (1972) 271–281.

[18] Y. Chen, J.-W. Sanders, Abstraction of Object Graphs in Program Verification, in: C. Bolduc, J. Desharnais, B. Ktari (Eds.), Proc. of 10th Intl. Conference on Mathematics of Program Construction, volume 6120 of *LNCS*, Springer, 2010, pp. 80–99.

[19] J. Desharnais, B. Möller, Characterizing Determinacy in Kleene Algebra (Revised version), Technical Report 2001-03, Institute of Computer Science, University of Augsburg, April 2001.

[20] S. Wang, L.-S. Barbosa, J.-N. Oliveira, A Relational Model for Confined Separation Logic, in: Proc. of the 2nd IFIP/IEEE Intl. Symposium on Theoretical Aspects of Software Engineering, TASE '08, IEEE Press, 2008, pp. 263–270.

[21] W. W. McCune, Prover9 and Mace4, `http://www.cs.unm.edu/~mccune/prover9`, 2005.

[22] G. Sutcliffe, C. Suttner, The TPTP problem library: CNF release v1.2.1, Journal of Automated Reasoning 21 (1998) 177–203.

[23] P. Höfner, G. Struth, Can refinement be automated?, in: E. Boiten, J. Derrick, G. Smith (Eds.), Refine 2007, volume 201 of *ENTCS*, Elsevier, 2008, pp. 197–222.

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

[25] A. Hobor, J. Villard, The Ramifications of Sharing in Data Structures, in: R. Giacobazzi, R. Cousot (Eds.), Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL, ACM Press, 2013, pp. 523–536.