Extended Transitive Separation Logic

H.-H. Dang*, B. Möller
Institut für Informatik, Universität Augsburg, D-86135 Augsburg, Germany

Abstract
Separation logic (SL) is an extension of Hoare logic by operators and formulas for reasoning more flexibly about heap portions or linked object/record structures. In the present paper we give an algebraic extension of SL at the data structure level. At the same time we step beyond standard SL by studying not only domain disjointness of heap portions but also disjointness along transitive links. To this end we define operations that allow expressing 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 that includes operators and formulas to reason more flexibly about heap portions (heaplets) or linked object/record structures. The central connective of this logic is the separating conjunction $P_1 \ast P_2$ of formulas $P_1, P_2$. It guarantees that 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 when considering a dereferencing of $x$ like in $\ast x = y$. For a concrete example consider

```
x 1 2 3 4 5
y 7 8
```

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, … , 5 and $P_2$ those of the records with contents 7, 8. Note that $P_1 \ast P_2$ 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 unintendendly 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 $\ast$ that ensure the absence of sharing or that restrict sharing in such a way that unintended changes can be ruled...

*Corresponding author.
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].

Preprint submitted to Elsevier December 31, 2014
out. 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 termination in such algorithms. Since we treat such transitive dependences within data structures, we call the approach transitive separation logic. In particular, 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 concise and readable reasoning. The approach is illustrated with examples of in-situ list reversal, tree rotation and overlaid data structures like threaded trees. We elaborate on ideas of previous work on algebras for pointer structures [2, 3, 4] and abstract formalisations of separation logic [5, 6] to combine them in a separation logical setting which enables, in addition to semi-automated reasoning [7], also more flexible reasoning about linked object structures.

2. Basics and Definitions

Our basic algebraic structure is that of a modal Kleene algebra [8], since it allows simple proofs in a calculational style and has proved to enable a suitable abstraction for pointer structures [9]. It further allows the application of first-order theorem provers [7] and captures a lot of models such as relations, regular languages or finite traces. We will introduce its constituents in several steps.

The basic layer is an idempotent semiring \((\mathbb{S}, +, \cdot, 0, 1)\), where \((\mathbb{S}, +, 0)\) forms an idempotent commutative monoid and \((\mathbb{S}, \cdot, 1)\) a monoid. An intuitive example of an idempotent semiring is provided by taking \(\mathbb{S}\) 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 \(x \leq y \Leftrightarrow x + y = y\). In the relational interpretation, \(\leq\) coincides with inclusion \(\subseteq\). We assume the existence of a greatest element that we denote by \(\top\). In the relational semiring, it is given by the universal relation. Moreover, + is the supremum operation of an upper semilattice and hence satisfies the sum-split rule:

\[
x + y \leq z \Leftrightarrow x \leq z \land y \leq z.
\]

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 general semirings, this approach is mimicked by using sub-identity elements \(p \leq 1\), called tests [10, 11]. 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 \(p \cdot \neg p = 0\). Thus, tests have to form a Boolean subalgebra. This implies that on tests + coincides with the binary supremum \(\cup\) and \(\cdot\) with the binary infimum \(\cap\). Every semiring contains at least the greatest test 1 and the least test 0.

When using tests, the product \(p \cdot a\) can be used to restrict an element \(a\) to links that start in nodes of \(p\) while, symmetrically, \(a \cdot p\) restricts \(a\) to links ending in nodes of \(p\). Following [8], this behaviour is used to axiomatise the operators \(\cap\) and \(\cup\) that represent the domain and codomain of a semiring element as tests. Consistent with 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:

\[
\begin{align*}
\gamma a \leq a \cdot a, & \quad \gamma (p \cdot a) \leq p, & \quad \gamma (a \cdot b) = \gamma (a \cdot \bar{b}), & \quad a \leq a \cdot a, & \quad (a \cdot p) \leq p, & \quad (a \cdot b) \leq (a \cdot \bar{b}).
\end{align*}
\]

These imply fundamental properties such as additivity and isotony, among others, see [8]. We mention the following ones:

\[
\begin{align*}
a = 0 & \Leftrightarrow \gamma a = 0 \Leftrightarrow a^2 = 0, & \quad \text{(full strictness)}
\end{align*}
\]

\[
\begin{align*}
\gamma a \leq p & \Leftrightarrow a \leq p \cdot a, & \quad \text{(lp)}
\end{align*}
\]

\[
\begin{align*}
p \leq \gamma a & \Leftrightarrow p \cdot a \leq 0, & \quad \text{(gla)}
\end{align*}
\]

\[
\begin{align*}
\gamma a \leq p & \Leftrightarrow a \leq a \cdot p, & \quad \text{(lrp)}
\end{align*}
\]

\[
\begin{align*}
p \leq \gamma a & \Leftrightarrow a \cdot p \leq 0. & \quad \text{(gra)}
\end{align*}
\]
Formula (llp) says that $\overline{a}$ is the least left preserver of $a$, while (gla) says that $\neg \overline{a}$ is the greatest left annihilator of $a$. Symmetric explanations apply to the codomain laws.

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

$$\langle a \rangle p =_{df} (p \cdot a)^\top.$$ 

Since this is an abstract version of the diamond operator from modal logic, an idempotent semiring with it is called modal. The diamond $\langle a \rangle p$ 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 $+$ 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 [12] by an iteration operator $\ast$. It can be axiomatised by the following unfold and induction laws:

$$1 + x \cdot x^\ast \leq x^\ast, \quad x \cdot y + z \leq y \Rightarrow x^\ast \cdot z \leq y,$$
$$1 + x^\ast \cdot x \leq x^\ast, \quad y \cdot x + z \leq y \Rightarrow z \cdot x^\ast \leq y.$$

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

$$\text{reach}(p,a) =_{df} \langle a^\ast \rangle p.$$ 

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

$$\text{reach}(p,a) = p + \langle a \rangle \text{reach}(p,a), \quad \text{reachunfold}$$
$$p \leq q \land \langle a \rangle q \leq q \Rightarrow \text{reach}(p,a) \leq q. \quad \text{reachinduct}$$

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

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

Using $\Box$ we also characterise the subset of elements that have no links emanating from the pseudo-reference $\Box$ to any other address $\neq \Box$. This is a natural requirement, since the general purpose of $\Box$ is to denote a terminator reference. We refer to this property as properness. Formally, an element $a$ is called proper iff $\Box \cdot a \leq \Box$. This implies that $\Box$ is proper and for proper elements $\Box \cdot a$ is a test. Moreover, properness is downward closed, i.e., if $a$ is proper and $b \leq a$ then $b$ is proper, too. We summarise a few more consequences.

**Corollary 2.1.** $a_1, a_2$ are proper iff $a_1 + a_2$ is proper.

**Lemma 2.2.** For an element $a$ with $\Box \cdot \overline{a} = 0$ the following properties hold:

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

**Proof.** For 1 and 2 we calculate $\Box \cdot a = \Box \cdot \overline{a} \cdot a = \overline{a} \cdot a = 0 \leq \Box$ by domain axioms and assumption. Finally, for 3 we have $a = 1 \cdot a = (\Box + \overline{\Box}) \cdot a = \Box \cdot a + \overline{\Box} \cdot a = \Box \cdot a$ by neutrality, $\Box$ being a test, distributivity and 2.

**3. A Stronger Notion of Separation**

Following the example given in Section 1, we now continue to define an adequate operation that excludes sharing. We start by two simple sharing patterns in data structures that cannot be excluded by $\ast$ only.
In both cases $h_1$ and $h_2$ satisfy the disjointness property, since $\mathcal{h}_1 \cap \mathcal{h}_2 = \emptyset$. 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 data structures is a main correctness property needed for many algorithms that work, e.g., with linked lists or tree structures.

Hence, in many cases the separation expressed by $\mathcal{h}_1 \cap \mathcal{h}_2 = \emptyset$ 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 [9, 3] we give a stronger separation relation $\oplus$ on access elements.

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

$$a_1 \oplus a_2 \iff_d \text{reach}(\mathcal{a}_1, a) \cdot \text{reach}(\mathcal{a}_2, a) \leq \Box .$$

Intuitively, $a$ is strongly separated into $a_1$ and $a_2$ if each address except $\Box$ reachable from $a_1$ is unreachable from $a_2$ w.r.t. $a$, and vice versa. However, since $\Box$ 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 can be used to rule out the data structures in Figure 1.

Clearly, $\oplus$ is symmetric and $\Box \oplus a$ and $\ominus \oplus a$ hold. Moreover, since by definition we have for all $p, b$ that $p \leq \text{reach}(p, b)$, the new separation condition indeed implies the analogue of the old one, i.e., both parts are disjoint: $a_1 \oplus a_2 \Rightarrow \mathcal{a}_1 \cdot \mathcal{a}_2 = 0$.

Finally, $\oplus$ is downward closed by isotony of reach: $a_1 \oplus a_2 \land b_1 \leq a_1 \land b_2 \leq a_2 \Rightarrow b_1 \oplus b_2$.

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

Definition 3.2. The nodes $\overline{a}$ of an access element $a$ are given by $\overline{a} = a \cdot \mathcal{a} + a$. A node in $\overline{a} - \mathcal{a}$ 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 $\overline{\Box} = 0$. Note that we consider $\Box$ as a node; this saves some case distinctions in calculations. Therefore we stay with Definition 3.2. We show two further properties that link the nodes operator with reachability.

Lemma 3.3.

1. $\overline{a} \leq \text{reach}(\mathcal{a}, a)$,

2. $(\mathcal{b} \overline{a}) \leq \overline{a} \Rightarrow \text{reach}(\mathcal{a}, a + b) = \overline{a}$; in particular, $\overline{a} = \text{reach}(\mathcal{a}, a)$.
Proof.

1. We use the definition of $\langle\rangle$ and (sum-split). First, $\forall a \leq \text{reach}(\langle a \rangle, a)$ by (reachinduct).
   Second, by a domain property, $\forall d = \langle a \cdot a \rangle = \langle a \rangle \leq \text{reach}(\langle a \rangle, a)$.

2. For $\leq$ we know by diamond star induction that $\text{reach}(\langle a \rangle, a + b) \leq \langle a \rangle \iff \forall a \leq \langle a \rangle \wedge \langle (a + b) \rangle \leq \langle a \rangle$. $\forall a \leq \langle a \rangle \wedge \langle (a + b) \rangle \leq \langle a \rangle$. These two conjunts hold by $\langle a \rangle$ and the assumption. The direction $\geq$ follows from Part 1, $a \leq a + b$ and isotony of $\text{reach}$.
   Within this proof we have also shown that $\langle a \rangle \leq \langle a \rangle$, which justifies the final claim.

Trivially, the first law states that all nodes in the domain and range of an access element $a$ are reachable from $a$, 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 this lemma we can give a simpler equivalent characterisation of $\otimes$.

Lemma 3.4. If $a, b$ are proper then $a \otimes b \iff a \cdot b \leq \Box$.

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

($\Leftarrow$) $a \cdot b \leq \Box$ implies $a \cdot b \leq \Box$. Hence $\langle b | a \rangle = (\langle a \cdot a \rangle \cdot b \cdot b) \leq (\langle a \cdot a \rangle \cdot b \cdot b) \leq (\langle a \cdot a \rangle \cdot b \cdot b) \leq (\langle a \cdot a \rangle \cdot b \cdot b) \equiv (\langle a \rangle \cdot b)$, since $a$ is proper and $\Box$ are tests. Symmetrically $\langle a \rangle \bar{b} \leq \Box$ holds. Now, Lemma 3.3.2 tells us $\text{reach}(\langle a \rangle, a + b) \cdot \text{reach}(\langle b \rangle, a + b) = \langle a \rangle \cdot b$, from which the claim is again immediate.

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 $\otimes$. 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 $\otimes$ is bilinear, i.e., satisfies

$$(a + b) \otimes c \iff a \otimes c \wedge b \otimes c \quad \text{and} \quad a \otimes (b + c) \iff a \otimes b \wedge a \otimes c.$$

Proof. We use the characterisation of $\otimes$ from Lemma 3.4. First, we calculate $(a + b) \otimes c \iff a \cdot b \cdot c \leq \Box \iff (a + b) \cdot c \leq \Box \iff a \cdot c \leq \Box \iff a \otimes c \wedge b \otimes c$. The other equivalence follows from commutativity of $\otimes$.

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. A predicate is a set of access elements. For predicates $P_1$ and $P_2$, we define the separating conjunction $*$ and the strongly separating conjunction $\oplus$ by

$$P_1 \ast P_2 = \{ a + b : a \in P_1, b \in P_2, a \ast b \} ,$$

$$P_1 \oplus P_2 = \{ a + b : a \in P_1, b \in P_2, a \oplus b \} .$$

where $a \ast b = \langle a \rangle \cdot \bar{b}$. Moreover, we call a predicate proper if all its elements are proper. By Cor. 2.1 the operators $*$ and $\oplus$ preserve properness.

Lemma 3.7. $\oplus$ is commutative and associative. Moreover, $P \oplus \text{emp} = P$ where $\text{emp} = \{ 0 \}$. 

5
Proof. Commutativity is immediate from the definition. Neutrality of \( \text{emp} \) follows from \( 0 \oplus a \) and by neutrality of \( 0 \) w.r.t. \( + \).

For associativity, assume \( a \in (P_1 \oplus P_2) \oplus P_3 \), say \( a = a_{12} + a_3 \) with \( a_{12} \oplus a_3 \) and \( a_{12} \in P_1 \oplus P_2 \) and \( a_3 \in P_3 \). Then there are \( a_1, a_2 \) with \( a_1 \oplus a_2 \) and \( a_{12} = a_1 + a_2 \) and \( a_{12} \in P_i \). By Lemma 3.5 \( a_{12} \oplus a_3 \) is equivalent to \( a_{1} \oplus a_3 \land a_{23} \oplus a_3 \). Using Lemma 3.5 again, \( a_{1} \oplus a_2 \land a_{13} \oplus a_3 \iff a_{1} \oplus a_{23} \) where \( a_{23} = a_2 + a_3 \). Therefore \( a \in P_1 \oplus (P_2 \oplus P_3) \). Hence \( (P_1 \oplus P_2) \oplus P_3 \subseteq P_1 \oplus (P_2 \oplus P_3) \). The reverse inclusion is shown symmetrically. \( \square \)

The defined connectives are structurally similar to operations given in [13]. Although that paper presented them with another application, they still can be interpreted for our applications due to abstraction. We present some of their properties and use them to characterise the interplay between separating conjunction and our stronger connective.

Lemma 3.8 (Exchange [13]). Assume a semigroup \((A, +)\). Then for bilinear relations \( R \) and \( S \) with \( R \subseteq S \) we have

\[
P_1 \oplus P_2 \subseteq P_1 \oplus P_2,
(P_1 \oplus P_2) \oplus P_3 \subseteq P_1 \oplus (P_2 \oplus P_3),
(P_1 \oplus P_2) \ominus (P_3 \ominus P_4) \subseteq (P_1 \ominus P_3) \ominus (P_2 \ominus P_4),
\]

with \( P_i \subseteq A \) and \( P \ominus Q \) \(=\) \( \{a + b : a \in P, b \in Q, aRb \} \).

Since \( \ominus \) and the standard separation condition \( *\) are bilinear and \( a_1 \ominus a_2 \Rightarrow a_1 * a_2 \) as mentioned above, results from [13] immediately yield:

Corollary 3.9. For proper predicates \( P_i \) the following inequations hold:

\[
P_1 \oplus P_2 \subseteq P_1 * P_2,
(P_1 * P_2) \oplus P_3 \subseteq P_1 * (P_2 \oplus P_3),
P_1 \ominus (P_2 \ominus P_3) \subseteq (P_1 \ominus P_2) \ominus P_3,
(P_1 * P_2) \ominus (P_3 * P_4) \subseteq (P_1 \ominus P_3) \ominus (P_2 \ominus P_4).
\]


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 in SL implicitly weld into recursive data type predicates. To explain this, we concentrate on singly linked lists. In [14] the predicate \( \text{list}(x) \) 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:

\[
h \models \text{list}(\text{nil}) \iff_h h = \emptyset,
x \neq \text{nil} \Rightarrow (h \models \text{list}(x) \iff_h \exists y : h \models [x \mapsto y] * \text{list}(y)).
\]

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

Note that using \( \ominus \) instead of \( * \) would not work, because the heaplets in the definition 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 \( a^2 \leq \text{emp} a \).
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 $\emptyset$.

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 $\overline{a} \cdot \overline{a} \leq \emptyset$.

**Proof.** As tests form a Boolean subalgebra we conclude $\overline{a} \cdot \overline{a} \leq \emptyset \iff a \cdot \overline{a} \leq \emptyset \iff a \leq \emptyset + \emptyset$. □

**Lemma 4.4.** For proper and closed $a_1, a_2$ with $a_1 \ast a_2$ we have $a_1 \ominus a_2$.

**Proof.** By distributivity and order theory we know

$$\overline{a_1} \cdot \overline{a_2} \leq \emptyset \iff \overline{a_1} \cdot \overline{a_2} \leq \emptyset \wedge \overline{a_1} \cdot \overline{a_2} \leq \emptyset \wedge a_1 \cdot \overline{a_2} \leq \emptyset \wedge a_1 \cdot a_2 \leq \emptyset .$$

The first conjunct holds by assumption and leastness of 0. Since properness implies $\emptyset \cdot \overline{a} \leq \emptyset$ we calculate by closedness for the second, and analogously for the third, conjunct $\overline{a_1} \cdot \overline{a_2} \leq \overline{a_1} \cdot (a_2 + \emptyset) = \overline{a_1} \cdot a_2 + \overline{a_1} \cdot \emptyset = \overline{a_1} \cdot \emptyset \leq \emptyset$. The last conjunct again reduces by distributivity and the assumptions to $\emptyset \cdot \emptyset \leq \emptyset$ which is trivial, since $\emptyset$ 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 [15], 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 [9, 4, 3, 16] 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 $p \neq \emptyset$ we have $p \cdot (a^+|p = 0$, 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 with at least one edge from $x$ to $y$ within $a$. Atomicity of $p$ is needed to represent a single node; the definition would not work for arbitrary sets of nodes. For instance, if $a = \{(1, 2), (3, 4)\}$ then $a$ is acyclic, but for the non-atomic test $p = \{(1, 1), (2, 2), (3, 3)\}$ we have $q =_{df} \{a^+|p = \{(2, 2), (4, 4)\}$ and $p \cdot q = \{(2, 2)\} \neq \emptyset$. The element $\emptyset$ is excluded from consideration, since it is only 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 $p \neq \emptyset$ we have $p \cdot a^+ \cdot p = 0$.\]
Proof. By definition of diamond, a codomain property and (full strictness),
\[ p \cdot (a^+) p = 0 \iff (p \cdot a^+) \cdot p = 0 \iff (p \cdot a^+ \cdot p) = 0 \iff p \cdot a^+ \cdot p = 0. \]

Next, since certain access operations are deterministic, we need an algebraic characterisation of determinacy. We borrow it from [17]:

Definition 5.3. An access element \( a \) is deterministic iff for all tests \( p : \langle a \rangle p \leq p \), where the dual diamond is defined by \( \langle a \rangle p = \langle (a \cdot p) \rangle \).

A relational characterisation of determinacy of \( a \) is \( a^\rightarrow \cdot a \leq 1 \), where \( ^\rightarrow \) 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 structures other than relations. The roles of the expressions \( a^\rightarrow \) and \( a \) are now played by \( \langle a \rangle \) and \( |a| \), respectively.

Lemma 5.4. If \( a \) is deterministic and \( \vec{a} \) is an atom then also \( \vec{a}^\rightarrow \) is an atom.

Proof. We first show the auxiliary result
\[ p \leq \vec{a} \land |a| p = 0 \Rightarrow p = 0. \]

We have, by the definition of diamond, (full strictness) and (gra),
\[ |a| p = 0 \iff \vec{a} = 0 \iff a \cdot p = 0 \iff p \leq \vec{a}. \]

Since by assumption \( p \leq \vec{a} \), we get \( p \leq \vec{a} \land \vec{a} = 0 \).

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

Now assume \( p \leq \vec{a} \land p \neq 0 \). By Equation (*) we have \( 0 \neq |a| p = \vec{a} \cdot p \leq \vec{a} \). Hence, atomicity of \( \vec{a} \) implies \( |a| p = \vec{a} \). Now, by definition of diamond and determinacy of \( a \),
\[ \vec{a} = \langle a \rangle \vec{a} = \langle a \rangle |a| p \leq p, \]
so that altogether we have \( p = \vec{a} \), which, by the assumptions and the definition of atomicity, shows the claim.

Interestingly, this proof is independent of the set of all tests being 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 in a record structure 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 \( L \) is denoted by \( S_L \). Since \( \circ \) is proper and deterministic we will also view it as an element of \( S_L \) although it has no selectors.

- A linked structure \( a \) is a forest iff \( a \) is acyclic and injective, i.e., has maximal in-degree 1 except possibly for \( \circ \). Algebraically this is expressed by the dual of the formula for determinacy, namely
\[ \forall p : |a'| (a'|p \leq p, \quad \text{where } a' =_{df} a \cdot \neg \circ. \]

By distributivity of the diamonds, for a linked structure \( a = (a_l)_{l \in L} \) injectivity of \( \Sigma_{l \in L} a_l \) is equivalent to pairwise (relative) injectivity of the \( a_l \), i.e., to \( \forall k,l \in L : \forall p : |a'_k| (a'_l|p \leq p \). The case \( k = l \) expresses injectivity of every component \( a_k \).
Moreover, we define for forests $a$

$$\text{roots}(a) = df \left(\gamma - a\right) + \square \cdot a.$$  

By properness and since $\square$ is atomic, the term $\square \cdot a$ equals $\square$ when $\square \subseteq a$ and is 0 otherwise.

- A forest $a$ is called a tree iff $r = df$ roots$(a)$ is atomic and $\overline{\gamma} = \{a^*\} r$; in this case $r$ is called the root of the tree and denoted by root$(a)$. If additionally $L = \{\text{left, right}\}$ 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 $\overline{\gamma} a$ 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

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

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 combinable iff the root of $a_2$ equals one of the leaves of $a_1$.

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

$$a_1 \triangleright a_2 \iff \gamma a_1 \cdot a_2 = 0 \land a_1 \cdot a_2 \leq \square \land a_1 \cdot a_2 = \text{root}(a_2).$$

This relation guarantees domain disjointness and excludes occurrences of cycles, since $\gamma a_1 \cdot a_2 = 0 \iff a_1 \psi a_2 = 0 \land a_1 \cdot a_2 = 0$. 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 by injectivity the root of $a_2$ cannot occur more than once in $a_1$.

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 $\square \triangleright a \iff \text{FALSE}$ and $a \triangleright \square \iff \square \subseteq \overline{a}$.

**Proof.** First, we have $\square \triangleright a \iff \square \cdot a = 0 \land \square \cdot a \leq \square \land \square \cdot a = \text{root}(a)$. Now, $\square \cdot \overline{a} = \text{root}(a)$ implies root$(a) \subseteq \square$ and, since root$(a)$ is atomic and hence $\neq 0$, it must equal $\square$. By definition also $a = \square$ which immediately contradicts $\square \cdot a = 0$.

Second, $a \triangleright \square$ implies $\gamma a \cdot \square = 0 \land a \cdot \square \leq 0 \land a \cdot \square = 0$. 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 $\square \subseteq \overline{a}$. $\square$

**Lemma 6.3.** For trees $a_1$ and $a_2$ with $a_1 \triangleright a_2$ we have root$(a_1 + a_2) = \text{root}(a_1)$.

**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$(a_1 + a_2) = a_1 \cdot \neg a_1 \cdot \neg a_2 + a_2 \cdot \neg a_1 \cdot \neg a_2$.

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

Since the directed disjointness relation $\triangleright$ is defined only on tree-like structures, we extend it now to arbitrary forests. For this we assume in the following that any forest $a$ can be represented as a finite summation of trees $a_i$, i.e., $a = \sum a_i$. 9
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 $a_{i_1} \oplus a_{i_2}$ ($i_1 \neq i_2$) and $b_{j_1} \oplus b_{j_2}$ ($j_1 \neq j_2$). Then we define directed combinability by

$$a \triangleright b \iff \exists i, j : a_i \triangleright b_j \land \left( \sum_{k \neq i} a_k \right) \oplus b_j .$$

Note that $\triangleright$ on the constituent trees $a_i, b_j$ is given by Definition 6.1. We refer to such components by numbers $i \in \mathbb{N}$ and to a particular selector $l \in L$ by the access element $a_l$. The above definition requires at least two constituent trees of forests $a$ and $b$ to be connected w.r.t. $\triangleright$ while all unconnected trees must be strongly disjoint.

![Figure 3: $\triangleright$-combination of two forests $a, b$](image)

We now show that $\triangleright$ guarantees preservation of linked structures under $\oplus$.

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

1. If the $a_i$ are deterministic and $\overline{a_1} \cdot \overline{a_2} = 0$ then also $a_1 + a_2$ is deterministic.

2. If the $a_i$ are injective and $a_1 \cdot a_2 \leq \circ$ then also $a_1 + a_2$ is injective.

3. If the $a_i$ are acyclic and $a_2 \cdot \overline{a_1} = 0$ then also $a_1 + a_2$ is acyclic.

Proof.

1. By distributivity, $\langle a_1 + a_2 \rangle | a_1 + a_2 \rangle p \leq p$, since $\langle a_1 \rangle | a_1 \rangle p \leq p$ and $\langle a_2 \rangle | a_1 \rangle p \leq 0 \land \langle a_1 \rangle | a_2 \rangle p \leq 0$ by $\overline{a_1} \cdot \overline{a_2} = 0$.

2. By definition and distributivity we have $(a_1 + a_2)' = (a_1 + a_2) \cdot \overline{\circ} = a_1 + a_2$. Now we can reason symmetrically to Part 1.

3. Assume an arbitrary atomic test $p \neq \circ$. We show $p \cdot (a_1 + a_2)^+ \cdot p = 0$. First note that if $a_2 \cdot \overline{a_1} = 0$ then $(a_1 + a_2)^+ = a_1^+ + a_2^+ + a_2^\circ$. This follows using $(x + y)^+ = x^+ \cdot (y \cdot x)^+$, domain properties and the definition of $\cdot$.

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

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

Corollary 6.6. Consider a selector set $L$. If $a_1, a_2 \in S_L$ are linked structures with $\overline{a_1} \overline{a_2} = 0$ and $a_1 \cdot \overline{a_2} \leq \circ$ 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 $a_1 \triangleright a_2$ then $a_1 + a_2$ is again a tree whose root is that of $a_1$.

Proof. Let, for abbreviation $r_i =_{df} \text{root}(a_i)$. 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 $\text{roots}(a_1 + a_2) = \text{roots}(a_1)$ and thus is atomic. It remains to show $\overline{a_1} + a_2 = \{(a_1 + a_2)^*|r_1\}$. We know that $\overline{a_1} + a_2 = \overline{a_1} + a_2^*$. First, since $a_1 \triangleright a_2$ implies $a_2^* \cdot r_1 = 0$ and hence $a_2 \cdot a_1 = 0$, we have $(a_1 + a_2)^* = a_1^* \cdot a_2^*$. Therefore,

\[
\{(a_1 + a_2)^*|r_1 = (a_1^* \cdot a_2^*)|r_1 = (a_1^*|a_1^*) = (a_1^*|a_1^*) = (a_1^*|a_1^*) \cdot (a_1^*|a_1^*) . \quad (*)
\]

For the second summand we have

\[
\{a_2^*|a_1^* = (a_2^*|a_2^*) = (a_2^*|a_2^*) = (a_2^*|a_2^*) = (a_2^*|a_2^*) .
\]

Moreover, since $r_2 \leq a_1$, we obtain from $(*)$

\[
\{(a_1 + a_2)^*|r_1 = \overline{a_1} + r_2 + \{a_2^*|r_2 = \overline{a_1} + \{a_2^*|r_2 = \overline{a_1} + a_2 .
\]

\[\square\]

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 $a_1 \triangleright a_2$ or $a_1 \odot a_2$ holds then also $a_1 + a_2$ is a forest.

Proof. Immediate from Lemma 6.7 and the definition of $\triangleright$ on forests. \[\square\]

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

\[
cell =_{df} \{ a : a \text{ is a cell } \},
list =_{df} \{ a : a \text{ is a chain } \},
tree =_{df} \{ a : a \text{ is a tree } \},
forest =_{df} \{ a : a \text{ is a forest } \} .
\]

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

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

\[
P \odot Q =_{df} \{ a_1 + a_2 : a_1 \in P, a_2 \in Q, a_1 \triangleright a_2 \} .
\]

This allows, conversely, also talking about decomposability: If $a \in P_1 \odot P_2$ then $a$ can be split into two disjoint parts $a_1, a_2$ such that $a_1 \triangleright a_2$ holds. To avoid excessive notation, in the sequel we tacitly assume that, as in this definition, all predicates involved in our formulas are restricted to the same set of selectors.

Lemma 6.11. forest $\odot$ forest $\subseteq$ forest, tree $\odot$ tree $\subseteq$ tree and list $\odot$ list $\subseteq$ list. As particular cases cell $\odot$ list $\subseteq$ list, tree $\odot$ cell $\subseteq$ tree and cell $\odot$ tree $\subseteq$ tree.

Lemma 6.12. Let $P, Q, R \subseteq$ tree. Then

\[
P \odot (Q \odot R) \subseteq (P \odot Q) \odot R ,
\]

\[
P \odot (Q \odot R) \subseteq (P \odot R) \ast Q ,
\]

\[
(P \odot Q) \odot R \subseteq P \odot (Q \odot R) ,
\]

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

By this we can immediately derive from distributivity and the definitions that $a_1 \triangleright a_2 \wedge (a_1 + a_2) \triangleright a_3$ and $a_1 \oplus a_3 \wedge (a_1 + a_3) \cdot a_2 \leq 0$, which shows the first two laws.

For the third law, assume $a_1 \triangleright a_2$ and $(a_1 + a_2) \oplus a_3$ which is equivalent to $a_1 \oplus a_3 \wedge a_2 \oplus a_3$. 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 $\oplus$ and the definition of $\triangleright$ on forests. □

7. Expressions and Assertions

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 variable identifiers to nodes, i.e., atomic tests. The domain of a store $s$ is denoted by dom$(s)$. For a store $s$, an identifier $i$ and atomic test $P$ we denote by $s[i \leftarrow p]$ the store that assigns $p$ to $i$ and $s[j]$ to all identifiers $j \in \text{dom}(s) - \{i\}$. A state is a pair $(s,a)$ consisting of a store $s$ and a linked structure $a$. The set of all states is denoted by $\Sigma$.

Based on states we can define selector expressions and their semantics.

Definition 7.2. Consider a set $L$ of selector names. A selector expression has the form $i.l$ with variable identifier $i$ and a sequence $l = l_1 \ldots l_n \in L^+$ of selector names. Its semantics w.r.t. a state $(s,a)$ is defined as

$$[i.l](s,a) = \begin{cases} \{a_{l_1} \ldots a_{l_n} | (s(i)) \} & \text{if } i \in \text{dom}(s), \\ 0 & \text{otherwise}. \end{cases}$$

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

Selector expressions and identifiers are the atomic parts of expressions, which arise by arbitrary nesting of functions from a given set that will vary from application to application. Also $\Box$ and $0$ are expressions with the obvious semantics. The set $\text{FV}(e)$ of variable identifiers occurring in an expression $e$ is defined as usual.

Next we define ways of forming predicates.

Definition 7.3. For an identifier $i$ and a predicate $P \subseteq \text{tree}$ we define its extension $P(i)$ to states by

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

Syntactically, we will assume that $P$ is denoted by some identifier $P$ and call $P(i)$ an atomic formula. To ease reading, we will, however, just write $P$ instead of $P$. Further atomic formulas are formed by applying $=$ and $\neq$ to expressions. General formulas are formed from the atomic ones using the connectives $\wedge$, $\vee$, $\neg$, $\otimes$, $\otimes$, with the obvious semantics. The set $\text{FV}(P)$ of variable identifiers occurring in a formula $P$ is the union of all sets $\text{FV}(e)$ for the expressions $e$ occurring in $P$.

Using predicate extension 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., $P = \{ (s,a) : a \in P \}$. In particular, for operator $\circ \in \{ =, \neq \}$ and $l, m \in L^+$, we define special formulas by

$$\begin{align*}
( i \circ \Box) &= \{ (s,a) : i \in \text{dom}(s), \text{root}(a) \circ \Box \}, \\
( i.l \circ \Box) &= \{ (s,a) : 0 \neq [i.l](s,a) \circ \Box \}, \\
( i.l = j.m) &= \{ (s,a) : 0 \neq [i.l](s,a) = [j.m](s,a) \neq 0 \}. 
\end{align*}$$
The mechanism of predicate extension cannot be used with expressions $e$ involving selector chains. Simply setting $P(e) = _{eff} \{ (s, a) : a \in P, \text{root}(a) = [e]_{(s, a)} \}$ would, for instance, not work in a formula like $P(i) \otimes Q(i, i)$, since by the definition of $\otimes$ the expression $i, i$ would have to be evaluated in a disjoint portion of the underlying access element than $i$ itself. Instead, we use a syntactic solution: we view $P(i) \otimes Q(i, i)$ as an abbreviation for $(P(i) \otimes Q(j)) \cap (j = i, i)$ where $j$ is a fresh identifier. The predicate $j = i, i$ 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 7.4.** Let $i, j, k$ be identifiers and $\{ \varnothing \} \subseteq P, Q, R \subseteq \text{tree}$. Then

$$(P \otimes Q(j)) \otimes R(k) = P \otimes (Q(j) \otimes R(k))$$

if $\exists l \in L^+ : j, l = k$ ,

(5)

$$(P(i) \otimes Q(j)) \otimes R(j) = P(i) \otimes (Q \otimes R(j))$$

if $\forall l \in L^* : i, l \neq j$ ,

(6)

$$(P \otimes Q(j)) \otimes R(k) = P \otimes (Q(j) \otimes R(k))$$

if $\exists l \in L^+ : j, l = k$ .

(7)

**Proof.** Assume $a_1 \in P(i) \land a_2 \in Q(j) \land a_3 \in R(k)$. By assumption $a_i \neq \varnothing$.

(5) We only show $\subseteq$, since $\supseteq$ was shown in Lemma 6.12. By the definitions it remains to show that $(a_1 + a_2) \triangleright a_3 \land a_1 \triangleright a_2 \triangleright a_3$. The assumption $(a_1 + a_2) \triangleright a_3$ resolves to

$$(a_1 \cdot a_2) \subseteq a_3 \leq \varnothing \land a_1 \cdot a_3 \leq \varnothing \land a_2 \cdot a_3 \leq \varnothing \land a_3 = \text{root}(a_3) .$$

(6) The $\subseteq$-direction was again shown in Lemma 6.12. Now assume $a_1 \triangleright (a_2 + a_3)$ and $a_2 \otimes a_3$. The side condition implies $\varnothing \cdot \text{root}(a_3) \leq \varnothing$ which in turn implies $a_1 \cdot a_3 \leq \neg \text{root}(a_3)$. Therefore $a_1 \triangleright a_3$ does not hold and consequently $a_1 \triangleright a_2$ and $a_1 \otimes a_3$ need to be true by the definition of $\triangleright$ for forests.

(7) We assume $(a_1 + a_2) \triangleright a_3 \land a_1 \triangleright a_2$ and show $a_1 \triangleright (a_2 + a_3) \land a_2 \otimes a_3$. As for (5), $(a_1 + a_2) \triangleright a_3$ implies $a_1 \cdot a_3 \leq a_2 \cdot a_3 = \text{root}(a_3)$. We calculate $a_2 \cdot a_3 \subseteq a_2 \cdot a_1 \cdot a_3 = \text{root}(a_3)$ which by the assumption $(a_1 + a_2) \triangleright a_3$ further implies $a_1 \triangleleft a_3$. Next, the reverse direction is shown by $\text{root}(a_3) \leq a_3 \Rightarrow \neg (a_1 \otimes a_3)$, which in turn implies by $a_1 \triangleright (a_2 + a_3)$ and Definition 6.4 that $a_1 \triangleright a_3$ 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 $\text{root}(a_3) \leq a_3$ and hence $a_1 \triangleright a_3$ can not hold by $a_1 \otimes a_2$. Therefore by definition we can only have $a_1 \otimes a_3 \land a_2 \triangleright a_3$.

Now the claim follows by bilinearity of $\otimes$. □

We now consider the special case of chains.

**Corollary 7.5.** For arbitrary $P, Q, R \subseteq \text{list}$ and identifier $i$ we have

$$(P(i) \otimes Q(i, \text{next})) \otimes R(i, \text{next}, \text{next}) = P(i) \otimes (Q(i, \text{next}) \otimes R(i, \text{next}, \text{next})) ,$$

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

**Proof.** This follows from Lemma 7.4, Equation (5) by setting $j = i, \text{next}$ and $j, \text{next} = k$. □
8. Programming Constructs

Next we want to give the semantics of program commands, in particular, of assignments of the form \(i.d := e\).

8.1. Twigs

To this end, we enrich our algebra by another ingredient, namely by \textit{twigs}, i.e., abstract representations of single edges in the graph corresponding to a linked structure. Special assignments of the above form will modify such twigs.

\textbf{Definition 8.1.} Assuming atomic tests with \(p \cdot q = 0 \land p \cdot \top = 0\), we define a \textit{twig} by \(p \mapsto q \equiv df \ p \cdot \top \cdot q\). The corresponding update of a linked structure \(a\) is \((p \mapsto q) | a \equiv df \ (p \mapsto q) + \neg p \cdot a\). We assume that | binds tighter than + but less tight than \cdot.

We call the elements \(p \mapsto q\) twigs as they intuitively correspond to the least non-nil components in trees or forests. In the literature, twigs are also called rectangles. 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. The operator \mapsto occurs in various forms in the literature, e.g., as relational override in the Z notation [18].

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 \((p \mapsto q) = q\) and \((p \mapsto q) = p\).

\textbf{Lemma 8.2.} \(\overrightarrow{p \mapsto q} = p + q\) and roots\((p \mapsto q) = p\).

\textbf{Proof.} The first result is trivial. Second, \(\text{root}(p \mapsto q) = \overline{(p \mapsto q)} \cdot \neg(p \mapsto q) = p \cdot \neg q = p\), since \(p \cdot q = 0 \iff p \leq \neg q\) by shunting.

\hspace{1cm} Note that by \(a = 0 \iff \overline{a} = 0\), cells are always non-empty.

\textbf{Lemma 8.3.} For a cell \(a\) we have root\((a) = \overline{a}\), hence \(\neg\text{root}(a) \cdot a = 0\).

\textbf{Proof.} By definition root\((a) \leq \overline{a}\) and root\((a) \neq 0\). Thus root\((a) = \overline{a}\).

\textbf{Lemma 8.4.} Twigs \(p \mapsto q\) are cells.

\textbf{Proof.} By assumption, \(\overline{(p \mapsto q)}\) is atomic and \(\not\in \top\), hence proper. Moreover, reach\((p,p \mapsto q) = p \mapsto q = p + q\), acyclicity holds by \(p \cdot q = 0\). To show determinacy we conclude for arbitrary tests \(s: q \cdot s \leq q \Rightarrow q \cdot s = 0 \lor q \cdot s = q \iff q \cdot s = 0 \lor q \leq s\). Hence, \(\{p \mapsto q\} | p \mapsto q \} s \leq \{p \mapsto q\} | p \leq s\). The calculation for injectivity is analogous.

We summarise a few consequences that will be used in the examples to come.

\textbf{Corollary 8.5.} \((i \not\in \top) \cap \text{list}(i) = \text{cell}(i) \oplus \text{list}\) and \((i = \top) \cap \text{list}(i) = \{\top\}\).

\textbf{Proof.} We only show the first claim, since the second one is obvious. The 2-direction follows from Lemma 6.7. For \(\subseteq\) we know by the assumption \(i \neq \top\), the definitions and Lemma 6.2 that \(a \not\in \top\) for all \((s,a) \in \text{list}(i)\). Since \(a\) is a chain and therefore acyclic, we write \(a = (\text{root}(a) \mapsto \text{root}(b)) + b\) for \(b =_df \neg\text{root}(a) \cdot a\). Note that by Lemma 8.4 root\((a) \mapsto \text{root}(b) \in \text{cell}\). By this, one can show \(b \in \text{list}\) and \((\text{root}(a) \mapsto \text{root}(b)) > b\).

\textbf{Corollary 8.6.} \((i.\text{left} \not\in \top) \cap (i.\text{right} \not\in \top) \cap \text{tree}(i) = \text{cell}(i) \oplus (\text{tree}(i.\text{left}) \oplus \text{tree}(i.\text{right}))\).

\textbf{Proof.} A proof can be constructed similar to Corollary 8.5.
8.2. Commands and Programs

Now we are ready to define the meaning of concrete programming constructs. Semantically they are modelled by relations between states.

**Definition 8.7.** A *command* \( C \) is a relation \( C \subseteq \Sigma \times \Sigma \). *Programs* are defined by the following grammar, where the nonterminals \( \text{var}, \text{sel}, \text{exp} \) and \( \text{pred} \) specifying variable identifiers, selectors, expressions and formulas are assumed to have the obvious definitions:

\[
\text{prog} ::= \text{skip} | \text{prog} ; \text{prog} \\
| \text{if} \ pred \ \text{then} \ \text{prog} \ \text{else} \ \text{prog} \\
| \text{while} \ pred \ \text{do} \ \text{prog} \\
| \text{var} ::= \text{exp} \\
| \text{var} . \text{sel}^+ ::= \text{exp} \\
| \text{var} ::= \text{new cell} () \\
| \text{delete} (\text{var})
\]

The sets \( \text{MV}(S) \) of (potentially) modified and \( \text{RV}(S) \) of referenced variables of a program \( S \) are defined as follows:

\[
\begin{align*}
\text{MV}(\text{skip}) &= \emptyset, & \text{RV}(\text{skip}) &= \emptyset, \\
\text{MV}(S ; S') &= \text{df} \ \text{MV}(S) \cup \text{MV}(S'), & \text{RV}(S ; S') &= \text{df} \ \text{RV}(S) \cup \text{RV}(S'), \\
\text{MV}(\text{if} \ P \ \text{then} \ S \ \text{else} \ S') &= \text{df} \ \text{MV}(S) \cup \text{MV}(S'), & \text{RV}(\text{if} \ P \ \text{then} \ S \ \text{else} \ S') &= \text{df} \ \text{RV}(S) \cup \text{RV}(S'), \\
\text{MV}(i := E) &= \text{df} \ \{ i \}, & \text{RV}(i := E) &= \text{df} \ \text{FV}(E), \\
\text{MV}(\text{if} . \text{ll} := E) &= \text{df} \ \{ i, \text{ll} \}, & \text{RV}(\text{if} . \text{ll} := E) &= \text{df} \ \text{FV}(E), \\
\text{MV}(i := \text{new cell} ()) &= \text{df} \ \{ i \}, & \text{RV}(i := \text{new cell} ()) &= \text{df} \ \emptyset, \\
\text{MV}(\text{delete}(i)) &= \text{df} \ \{ i \}, & \text{RV}(\text{delete}(i)) &= \text{df} \ \emptyset.
\end{align*}
\]

Now we define the semantics \([S] \subseteq \Sigma \times \Sigma \) of program \( S \) by induction on the structure of \( S \).

**Definition 8.8.** In the following we assume an identifier \( i \), a selector set \( L \), a selector name \( l \in L \) and an expression \( e \) for which the semantics \([e]_{(s,a)}\) as defined above 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

\[
\begin{align*}
[i := e] =_{df} & \{(s,a), (s, [s[i \leftarrow p], a]) : i \in \text{dom}(s), p = [e]_{(s,a)}\}, \\
[l := e] =_{df} & \{(s,a), (s, [s[i \leftarrow p], a]) : i \in \text{dom}(s), s(i) \neq \ominus, s(i) \leq \lceil a \rceil\}, \\
[i := \text{new cell}()] =_{df} & \{(s,a), (s, [s[i \leftarrow p], (p \mapsto \ominus)]) : i \in \text{dom}(s), p \text{ is an atomic test, } p \leq \lceil a \rceil, p \neq \ominus\}, \\
[\text{delete}(i)] =_{df} & \{(s,a), (s, [s[i \leftarrow p], a]) : p = s(i), i \in \text{dom}(s), p \neq \ominus\}.
\end{align*}
\]

Finally,

\[
\begin{align*}
\text{[skip]} =_{df} I, & \quad [S : S'] =_{df} [S] : [S'], \\
\text{[if} \ P \ \text{then} \ S \ \text{else} S'\text{]} =_{df} [P] ; [S] \cup \neg [P] ; [S'], \\
\text{[while} \ P \ \text{do} \ S\text{]} =_{df} ([P] ; [S]^* ; \neg [P]),
\end{align*}
\]

where \( I \) is the identity relation on states.

In general, selector assignments do not preserve treeness. We provide sufficient conditions for that in the form of Hoare triples in the next section.
9. 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 $a$. For better readability we now omit the brackets $[\_\_\_]$ for the semantics of syntactic commands and predicates and do not distinguish predicates and their corresponding commands notationally. Following [11, 5] 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.

9.1. Rules for Selector Assignments

For better readability of concrete rules, we introduce some syntactic sugar and abbreviate, for expressions $e, e'$ and operators $\circ \in \{ *, \otimes, \oplus \}$, formulas of the form $Q \circ P(e) \land e' = e$ 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

$$\{ P(j) \oplus Q(j, l) \} \quad \text{to} \quad \{ P(\#) \oplus Q(\#) \}.$$  \hspace{1cm} (9)

**Lemma 9.1.** For predicates $P, Q, R \subseteq \text{tree}$, identifiers $i, j$ and link $l \in L$ we have

$$\{ (P(i) \oplus Q(i, l)) \oplus R(j) \} \quad \text{to} \quad \{ P(i) \oplus R(i) \land i.l = \Box \} \quad \{ P(i) \oplus Q(i, l) \}$$

$$\{ (P(i) \oplus R(j = i, l)) \oplus Q \} , \quad \{ P(i) \oplus R(j = i, l) \} , \quad \{ P(i) \oplus Q \land i.l = \Box \} .$$

For the proof see below. The conjunct $i.l = \Box$ in the precondition of the middle rule enforces that the execution of the assignment does not introduce memory leaks. In the postcondition of the right rule it additionally asserts that no object structure is linked to $i$ via the $l$-selector. Note that $\cap$ on predicates corresponds to their logical conjunction $\land$. To provide more intuition of what is happening in the leftmost rule of Lemma 9.1, we depict the shapes of the trees in the pre- and postcondition:

$$\{ \{ i.l \} \quad \text{to} \quad \{ i.l := j \} \}$$

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 $a_1 \in P \land a_2 \in Q \land a_3 \in R$ with $a_1 \triangleright a_2 \land a_1 \oplus a_3 \land a_2 \oplus a_3 \land a = a_1 + a_2 + a_3$.

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 $((\text{root}(a_1)) \mapsto \text{root}(a_3))[b_1 + c_1] \oplus a_2$. This is equivalent to $c_1 \oplus c_2 \land (\neg\text{root}(a_1) \cdot b_1) \oplus b_2 \land (\text{root}(a_3) \mapsto \text{root}(a_3)) \oplus b_2.

We show the three conjuncts in turn.

First, by determinacy and the assumption on the roots, $a_1 \cdot b_2 = \text{root}(a_2)$ is equivalent to $b_1 \cdot a_2 = \text{root}(a_2) \land c_1 \cdot a_2 = 0$. Hence, $c_1 \oplus c_2$.

Second, by assumption we know $(\text{root}(a_1) \cdot b_1) = \text{root}(a_2)$. This implies by injectivity of trees and hence of $a_1$ and atomicity of $\text{root}(a_2)$ that $(\neg\text{root}(a_1) \cdot b_1) \cdot a_2 = 0$. Hence, together with $a_1 \oplus a_2$ we have $(\neg\text{root}(a_1) \mapsto b_1) \oplus b_2$.

The third conjunct follows from $a_1 \oplus a_2$ and $a_1 \oplus a_3$.

It remains to show $((\text{root}(a_1) \mapsto \text{root}(a_3))[b_1 + c_1] \oplus a_3$. This can be calculated by similar considerations as above using again $a_1 \oplus a_3$.

In sum, $((\text{root}(a_1) \mapsto \text{root}(a_3))[b_1] + a_{L-l} \in (P(i) \ominus R(j, i, l)) \oplus Q$. \hfill \sqr
9.2. Frame Rules

An essential ingredient of all of SL are frame rules of various kinds. They allow modular verification by extending a result about a program with a smaller set of resources into a larger context without the need of re-proving it there.

We want to use our approach to formulate several such rules for our new operators and show their validity in an algebraic manner. We follow precursor ideas of [5, 19], where proofs are performed in a general and relational setting, so that we can easily adapt these results to the present work.

First, we lift the ⊙ and ⊖ operators to commands:

\[(s, a) \circ D (s', a') ⇔ \exists a_1, a_2, a_1', a_2' : a = a_1 + a_2 \land a_1 \circ a_2 \land a' = a_1' + a_2' \land a_1' \circ a_2'
\]
\[\land (s, a_1) C (s', a_1') \land (s, a_2) D (s', a_2')\]

where \(\circ = ⊕\) and \(\circ = ⊖\) or \(\circ = ⊖\) and \(\circ = ⊖\).

Now we give a number of results about validity of certain frame rules under additional side conditions. The each of these is specialised to a subclass of programs where these side conditions are automatically satisfied.

**Lemma 9.2.** Assume the following conditions for a command \(C\) and predicates \(P \subseteq \text{dom}(C)\) and \(R\):

\[(P \circ R) : C \subseteq (P : C) ⊕ R \quad \text{and} \quad C \circ R \subseteq C.\]

Then for all predicates \(Q\) we have the \(⊕\) frame rule

\[
\frac{\{P\} C \{Q\}}{\{P \circ R\} C \{Q \circ R\}}.
\]

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 [19].

**Lemma 9.3.** The \(⊕\) frame rule is valid for all predicates \([R]\) and commands \([S]\) where \(S\) is a program that does not modify or reference any subexpression of formula \(R\), i.e., for which \((\text{MV}(S) \cup \text{RV}(S)) \cap \text{FV}(R) = ∅\).

**Proof.** By Lemma 9.2 it suffices to show that all such commands satisfy the assumptions made there. We only consider the base cases in Definition 8.7. A proof for commands involving sequential composition, union and reflexive transitive closure 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 \(C = [i.l := e]\). For the reader’s benefit we repeat the semantic definition:

\[
[i.l := e] =_{df} \{(s, a), (s, (s(i) \to [e](s,a)))a_i + a_{L−i}) : i \in \text{dom}(s), a(i) \neq ∅, s(i) \subseteq \{a_i\}\}
\]

We outline a proof for the first assumption of Lemma 9.2; for the second one the argumentation is analogous.

For given states \((s_1, a_1)\), the premise of the rule resolves pointwise to

\[
((s_1, a_1), (s_2, a_p)) \subseteq C \land (s_1, a_p) \subseteq P \land (s_1, a_r) \subseteq R \land a_p ⊕ a_r \land a_1 = a_p + a_r
\]

for suitable \(a_p, a_r\). Since \(P \subseteq \text{dom}(C)\), there exists a transition \(((s_1, a_p), (s_1, b_p)) \in C\) where \(b_p = (s_1(i) \to \{e\}(s_1, a_p))(a_p) + (a_p)\)\_L\_i\_t, with \(s_1(i) \subseteq (a_p)\) \land \(s_1(i) \neq ∅\) and \(((s_1, b_p), (s_1, b_p)) \in Q\).

We assume \(a_p ⊕ a_r\), and show \(b_p ⊕ a_r\). By bilinearity of \(⊕\) we have

\[
b_p ⊕ a_r ⇔ (s_1(i) \to [e](s_1, a_p))(a_p) ⊕ a_r \land (a_p)\_L\_i\_t \oplus a_r.
\]

The second conjunct follows by downward closedness of \(⊕\) from \(a_p ⊕ a_r\) while the first is equivalent to

\[
\frac{(s_1(i) + \{e\}(s_1, a_p)) \cdot a_r \leq ∅ \land \neg (s_1(i) \cdot (a_p)) \cdot a_r \leq ∅.}{\text{Again the latter conjunct follows from downward closedness of } ⊕. \text{ For the former we infer } s_1(i) \cdot a_r \leq a_p \cdot a_r \leq 0 \text{ by } a_p ⊕ a_r \text{ and } [e](s_1, a_p) \cdot a_r \leq ∅, \text{ since } i.l := e \text{ does not reference any expression of } R.}
\]

\[\square\]
Lemma 9.4. Assume the following conditions for command $C$ and predicates $P, R \subseteq \text{tree}$ where additionally $P \subseteq \text{dom}(C)$:

$$(P \otimes R) \cup C \subseteq (P ; C) \otimes R,$$  
$$C \otimes R \subseteq C.$$

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

$$\frac{\{P\} \cup \{Q\}}{(P \otimes R) \cup \{Q \otimes R\} \subseteq C}.$$

Lemma 9.5. The $\otimes$ frame rule is valid for all predicates $[R]$ and commands $[S]$ where $S$ is a program that does not modify any subexpression of formula $R$, i.e., for which $\text{MV}(S) \cap \text{FV}(R) = \emptyset$.

Proof. The proof is similar as for Lemma 9.3. Again we only consider selector assignments $S$ and assume a transition $((s_1, a_p), (s_1, b_p)) \in [S]$ with $b_p = c((a_p)_l) + (a_p)_{L-t}$ where $c =_{df} (s_1(i) \mapsto [c]_{(s_1, a_p)})$ and $l \in L$.

The assumptions on $S, R$ induce the following conditions for $c$:

$$s_1(i) \cdot a_p \leq 0,$$
$$[c]_{(s_1, a_p)} : a_p \leq \emptyset,$$
$$[c]_{(s_1, a_p)} : a_p \leq 0,$$
$$\text{root}(a_p) \leq (-s_1(i) \cdot (a_p)_{L-t} + (a_p)_{L-t})^-.$$

The third condition states that no node of $a_p$ is referenced by $S$ while the last one describes that the root of $a_p$ 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 9.3. \qed

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 9.6. Assume the following conditions hold for command $C$ and predicates $P, R, \text{dom}(C), \text{cod}(C) \subseteq \text{forest}$ where $\text{cod}(C)$ denotes the codomain of $C$ and additionally $P \subseteq \text{dom}(C)$:

$$(R \otimes P) \cup C \subseteq R \otimes (P ; C), R \otimes C \subseteq C,$$

Then for all predicates $Q \subseteq \text{forest}$ we have the swapped $\otimes$ frame rule

$$\frac{\{P\} \cup \{Q\}}{(R \otimes P) \cup \{Q \otimes P\} \subseteq C}.$$

Lemma 9.7. The swapped $\otimes$ frame rule is valid for all predicates $[R] \subseteq \text{forest}$ and commands $[S]$ where $S$ is a program that does not delete the root of any tree in $P$ and does not modify or reference any subexpression of formula $R$, i.e., for which $\text{MV}(S) \cap \text{FV}(R) = \emptyset$.

Proof. The proof is similar as for Lemma 9.3. We consider selector assignments $S$ and assume a transition $((s_1, a_p), (s_1, b_p)) \in [S]$. By assumption $a_r \triangleright a_p$ the program $S$ either modifies a tree $t_p =_{df} (a_p)_l$ for which there exists another tree $t_r =_{df} (a_r)_l$, with $t_r \triangleright t_p$, or $S$ modifies a disjoint tree $t_p$ with $t_p \oplus t_r$ for all trees $t_r \leq a_r$.

Again we set $b_p = c((t_p)_l) + (t_p)_{L-t}$ with $c =_{df} (s_1(i) \mapsto [c]_{(s_1, a_p)})$ and $l \in L$. We assume the following conditions for $c$: $l \cdot r \leq 0 \land r \cdot l \leq \emptyset \land l \cdot s_1(i) \leq \text{root}(t_p) \land \text{root}(t_p) = \text{root}(b_p)$. 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 the definition of selector assignments, we have $s_1(i) \leq t_p$. Together with $t_r \oplus t_p$ this implies that $l \cdot s_1(i) = 0$. By this, it is not difficult to prove $t_r \oplus b_p$. \qed
10. Examples

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

10.1. In-Situ List Reversal

This example is mainly intended to show the basic ideas of our approach. We assume $L = \{\text{next}\}$. The algorithm is well known. It uses variables $i, j, k$. 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:

$$j := \emptyset; \text{ while } (i \not= \emptyset) \text{ do } (k := i.next; i.next := j; j := i; i := k).$$

To prove functional correctness of in-situ reversal we introduce the concept of abstraction functions [20]. They are used, e.g., to state invariant properties.

Definition 10.1. Assume $a \in \text{list}$ and an atom $p \in \mathbb{A}$. We define the abstraction function $\alpha_a$ 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 $i \rightarrow$ for a program identifier $i$:

$$\alpha_a(p) = df \begin{cases} \emptyset & \text{if } p \cdot a \leq \emptyset, \\ \alpha_a((a)p) & \text{otherwise}, \end{cases} \quad [i \rightarrow](s,a) = df \alpha_a(s(i)). \quad (10)$$

Here $\bullet$ stands for concatenation of words and $\emptyset$ 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 for functional correctness of the algorithm we use $I \iff_{df} (j^{-}) \bullet i \rightarrow = \alpha$, where $^{-}$ denotes word reversal. Its set-based semantics is defined by $(s,a) \in I \iff [\langle j^{-}\rangle] \bullet i \rightarrow ](s,a) = \alpha$ where $\alpha$ represents a word.

$$\begin{cases} \{ \text{list } i \land i \rightarrow = \alpha \} \\
    j := \emptyset; \\
    \{ \text{list } i \land j \land I \}
\end{cases} \quad \text{(Corollary 8.5)}$$

$$\begin{cases} \{ (\text{cell } i \odot \text{list } j) \land I \} \\
    \{ \text{cell } (i \odot \text{list } k) \odot \text{list } j \land (j^{-}) \bullet k \rightarrow = \alpha \} \\
    \{ \text{cell } (i \odot \text{list } k) \odot \text{list } j \land (i \bullet j^{-}) \bullet k \rightarrow = \alpha \} \\
    \{ \text{cell } (i \odot \text{list } k) \odot \text{list } j \land \langle \rangle \bullet k \rightarrow = \alpha \}
\end{cases} \quad \text{property of } \langle \rangle$$

$$\begin{cases} \{ \text{cell } (i \odot \text{list } j) \odot \text{list } k \land i \land I \} \\
    \{ \text{cell } (i \odot \text{list } j) \odot \text{list } k \land (i \bullet j^{-}) \bullet k \rightarrow = \alpha \} \\
    \{ \text{cell } (i \odot \text{list } k) \odot \text{list } j \land \langle \rangle \bullet k \rightarrow = \alpha \}
\end{cases} \quad \text{variable assignment}$$

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 [14] we hide in the $\odot$ 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 [21]), if one would, e.g., exchange the first two commands in the while loop above, this 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 \((\text{cell } (i) \oplus \text{list } (j)) \oplus \text{list}\). The list memory part separated out by the second argument of \(\oplus\) can neither be reached from \(i\) nor from \(j\). Moreover, there is no program variable containing a reference to the root of that part.

10.2. Tree Rotation

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

To define an abstraction function \(\leftrightarrow\) similar to the \(\rightarrow\) function in Equation (10), 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

\[
\begin{align*}
\text{tr}_a(p) &= df \begin{cases} 
\langle \rangle & \text{if } p \cdot \cdot a \leq \square , \\
\langle \text{tr}_a(\langle a_{\text{left}}|p\rangle), p, \text{tr}_a(\langle a_{\text{right}}|p\rangle) \rangle & \text{otherwise,}
\end{cases} \\
[i \leftrightarrow](s, a) &= df \text{tr}_a(s(i)) . 
\end{align*}
\]

(11)

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 algorithm starts with the left tree in the following Figure 4 and ends with the rotated one on the right.

\[
\begin{align*}
\text{cell } (i) \oplus (\text{tree } (i, \text{left}) \oplus (\text{cell } (i, \text{right}) \oplus (\text{tree } (i, \text{right}, \text{left}) \oplus \text{tree } (i, \text{right}, \text{right})))) .
\end{align*}
\]

(12)

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 trees.

Concretely for trees we set

\[
\begin{align*}
\text{left_tree_context}(i) &= df \text{cell } (i) \oplus \text{tree } (i, \text{right}) , \\
\text{right_tree}(i) &= df \text{left_tree_context}(i) \cap (i, \text{left} = \square) , \\
\text{right_tree_context}(i) &= df \text{cell } (i) \oplus \text{tree } (i, \text{left}) , \\
\text{left_tree}(i) &= df \text{right_tree_context}(i) \cap (i, \text{right} = \square) .
\end{align*}
\]

The left and right tree contexts can be depicted as follows:

Using these definitions we can transform Formula (12) using Lemma 7.4(7) into

\[
\begin{align*}
\text{right_tree_context}(i) \oplus \text{left_tree_context}(i, \text{right}) \oplus \text{tree } (i, \text{right}, \text{left}) .
\end{align*}
\]

(13)

20
Proof.

\[
\text{cell}(i) \odot (\text{tree}(i.\text{left}) \odot (\text{cell}(i.\text{right}) \odot (\text{tree}(i.\text{right}.\text{left}) \odot \text{tree}(i.\text{right}.\text{right}))))
\]

\[
= \ \\
(\text{cell}(i) \odot \text{tree}(i.\text{left})) \odot (\text{cell}(i.\text{right}) \odot (\text{tree}(i.\text{right}.\text{left}) \odot \text{tree}(i.\text{right}.\text{right})))
\]

\[
= \ \\
\text{right_tree_context}(i) \odot (\text{cell}(i.\text{right}) \odot (\text{tree}(i.\text{right}.\text{left}) \odot \text{tree}(i.\text{right}.\text{right})))
\]

\[
= \ \\
\text{left_tree_context}(i) \odot (\text{cell}(i.\text{right}) \odot (\text{tree}(i.\text{right}.\text{left}) \odot \text{tree}(i.\text{right}.\text{right})))
\]

\[
(\text{Lemma 7.4 (7)})
\]

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

\[
\{ \text{right_tree_context}(i) \odot (\text{left_tree_context}(i.\text{right}) \odot \text{tree}(i.\text{right}.\text{left})) \land i^* = \langle T_i, p, \langle T_k, q, T_r \rangle \rangle \}
\]

\[
j := i.\text{right};
\]

\[
\{ \text{right_tree_context}(i) \odot (\text{left_tree_context}(i.\text{right} = j) \odot \text{tree}(j.\text{left})) \land
i^* = \langle T_i, p, \langle T_k, q, T_r \rangle \rangle \land j^* = \langle T_k, q, T_r \rangle \}
\]

\[
\{ \text{right_tree_context}(i) \odot \text{left_tree_context}(i.\text{right} = j) \odot \text{tree}(j.\text{left}) \land
i^* = \langle T_i, p, \langle T_k, q, T_r \rangle \rangle \land j^* = \langle T_k, q, T_r \rangle \}
\]

\[
i.\text{right} := 0;
\]

\[
\{ \text{left_tree}(i) \odot \text{left_tree_context}(j) \odot \text{tree}(j.\text{left}) \land i^* = \langle T_i, p, \langle T_k, q, T_r \rangle \rangle \land j^* = \langle T_k, q, T_r \rangle \}
\]

\[
k := j.\text{left};
\]

\[
\{ \text{left_tree}(i) \odot \text{right_tree}(j) \odot \text{tree}(k) \land i^* = \langle T_i, p, \langle T_k, q, T_r \rangle \rangle \land j^* = \langle T_k, q, T_r \rangle \land k^* = T_k \}
\]

\[
j.\text{left} := 0;
\]

\[
\{ \text{left_tree}(i) \odot \text{right_tree}(j) \odot \text{tree}(k) \land i^* = \langle T_i, p, \langle T_k, q, T_r \rangle \rangle \land j^* = \langle T_k, q, T_r \rangle \land k^* = T_k \}
\]

\[
j.\text{left} := i;
\]

\[
\{ \text{left_tree_context}(j) \odot \text{left_tree}(i = j.\text{left}) \odot \text{tree}(k) \land
i^* = \langle T_i, p, \langle T_k, q, T_r \rangle \rangle \land j^* = \langle T_k, q, T_r \rangle \land k^* = T_k \}
\]

\[
\{ \text{left_tree_context}(j) \odot \text{left_tree}(i = j.\text{left}) \odot \text{tree}(k) \land
i^* = \langle T_i, p, \langle T_k, q, T_r \rangle \rangle \land j^* = \langle T_k, q, T_r \rangle \land k^* = T_k \}
\]

\[
i.\text{right} := k;
\]

\[
\{ \text{left_tree_context}(j) \odot \text{right_tree_context}(i = j.\text{left}) \odot \text{tree}(k) \land
j^* = \langle T_j, p, \langle T_k, q, T_r \rangle \rangle \land i^* = \langle T_i, p, \langle T_k, q, T_r \rangle \rangle \land k^* = T_k \}
\]

Note that the predicate \((i = \circ)\) satisfies the equation \((P(i) \odot Q) \cap (i.L = \circ) = (P(i) \cap (i.L = \circ)) \odot Q\) for \(P, Q \subseteq \text{tree}\). Therefore we can use Lemma 9.1 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 expressed with the operators we have defined so far.

21
Definition 10.2. For predicates $P, R \subseteq \text{forest}$ and $Q \subseteq \text{tree}$ we define
\[ P \oplus Q \oplus R = df \{ \ a_1 + a_2 + a_3 : a_1 \in P, a_2 \in Q, a_3 \in R, a_1 \triangleright a_2, a_3 \triangleright a_2, a_1 \mapsto a_3 = \text{root}(a_2) \ \}. \]

Clearly, $P \oplus Q \oplus R = R \oplus Q \oplus P$. 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 10.3. Assume predicates $P \subseteq \text{tree-context}$ and $Q, R \subseteq \text{tree}$, identifiers $i, j$ and selectors $l, m \in L$ then
\[
\begin{align*}
\{ \ (P(i) \oplus (Q(j = i.l) \oplus R(j.m))) \} & \quad \{ \ P(i) \oplus S(j.m = i.l) \oplus R(j) \} \\
& \quad \{ \ P(i) \oplus R(j.m = i.l) \oplus Q(j) \} , \quad \{ \ P(i) \oplus (R(j = i.l) \oplus S(j.m)) \} .
\end{align*}
\]
The latter rule also works for $P \subseteq \text{tree}$.

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 $a_1 \in P(i) \land a_2 \in Q(j = i.l) \land a_3 \in R(j.m)$. We know $a_1 \triangleright (a_2 + a_3) \land a_2 \triangleright a_3$ and from the identifiers $s(i) \mapsto j, m \mapsto \text{root}(a_1) \mapsto \text{root}(a_3)$.

Note that $a_1 \in P(i)$. This immediately implies $(\text{root}(a_1) \mapsto \text{root}(a_3))(a_1) = \text{root}(a_1) \mapsto \text{root}(a_3)$ and we set $b_1 = df \ (\text{root}(a_1) \mapsto \text{root}(a_3)) + (a_1)_{L \mapsto i}$. From the assumption we get $(a_1)_{L \mapsto i} \oplus a_2$. Using Lemma 6.12, we also know $a_1 \oplus a_3$ and can further infer $(a_1)_{L \mapsto i} \oplus a_3$. Now we can conclude $b_1 \triangleright a_3 \land b_1 \cdot a_2 = \text{root}(a_3)$.

Using Lemma 10.3 we can verify the following shorter form of the tree rotation algorithm that uses sharing.
\[
\begin{align*}
\{ \ \text{right_tree_context}(i) \oplus (\text{left_tree_context}(i.\text{right}) \oplus \text{tree}(i.\text{right.left})) \land \ i^* = \langle T_i, p, \langle T_k, q, T_r \rangle \rangle \} \\
& \quad \{ \ \text{right_tree_context}(i) \oplus (\text{left_tree_context}(i.\text{right} = j) \oplus \text{tree}(i.\text{left})) \land \\
& \quad \quad \quad \ i^* = \langle T_i, p, \langle T_k, q, T_r \rangle \rangle \land \ j^* = \langle T_k, q, T_r \rangle \} \\
& \quad \{ \ \text{right_tree_context}(i) \oplus \text{tree}(j.\text{left} = i.\text{right}) \oplus \text{left_tree_context}(j) \land \ i^* = \langle T_i, p, T_k \rangle \land \ j^* = \langle T_k, q, T_r \rangle \} \\
& \quad \quad \quad \ j.\text{left} = i; \\
& \quad \{ \ \text{left_tree_context}(j) \oplus (\text{right_tree_context}(i = j.\text{left}) \oplus \text{tree}(i.\text{right})) \land \\
& \quad \quad \quad \ j^* = \langle T_i, p, T_k, q, T_r \rangle \land \ i^* = \langle T_i, p, T_k \rangle \land \ k^* = T_k \}.
\end{align*}
\]

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](image-url)
11. A Treatment of 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). While the tree itself is rooted at i, the traversal using regular and thread links starts at the leftmost node j of the tree. It proceeds by following thread links until a regular right link is met. Then the game repeats starting from the leftmost node of the corresponding right subtree, and so on, until the rightmost node of the overall tree is reached. In this section we formalise this algorithm and corresponding invariants to show that it indeed visits the nodes in inorder (provided the thread links are placed correctly, which we also will treat formally).

To achieve this, we need some additional definitions. 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 restrictive, especially in the case of threaded trees. As an example, now consider non-reachability or directed reachability only on complete access elements, i.e., the operators in inorder (provided the thread links are placed correctly, which we 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

\[
a \triangleright_M b \iff a_M \triangleright b_M
\]

and its corresponding operator on predicates by

\[
P \ominus_M Q =_{df} \{ a + b : a \in P, b \in Q, a \triangleright_M b \}.
\]

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

The same generalisations apply to \( \oplus \) and \( \otimes \). Note that, by \( M \subseteq L \) and downward closedness of \( \oplus \), also \( \ominus \subseteq \ominus_M \) and hence \( P \ominus Q \subseteq P \ominus_M Q \). Note that our laws for \( \ominus \) and \( \otimes \) hold also for \( \ominus_M \) and \( \otimes_M \), resp., assuming a set of links \( M \subseteq L \).

For a threaded tree we define the access element by \( a = a_{\text{left}} + a_{\text{right}} + a_{\text{marked}} \), i.e., \( L = \{ \text{left}, \text{right}, \text{marked} \} \). Clearly the access elements \( a_{\text{left}} \) and \( a_{\text{right}} \) need to be disjoint, while \( a_{\text{marked}} \) is a test with \( a_{\text{marked}} \leq a_{\text{right}} \). 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 a “real” program this test would be implemented by marking bits on the nodes; it is here treated as an access element for uniformity.

Based on this, we define a virtual access element \( a_{\text{thread}} \); this means that the selector \( \text{thread} \) is not contained in \( L \), but the element is constructed using selectors of \( L \). It reflects the two types of traversal steps mentioned in the algorithm description above. At a marked node, the right-link is a thread link which is followed directly, whereas at an unmarked node it is a regular link and the traversal has to take a “macro-step”, described by the auxiliary access element \( a_{RLm} \), to the leftmost node of the corresponding right subtree. Formally,

\[
a_{\text{thread}} =_{df} a_{\text{marked}} \cdot a_{\text{right}} + \neg a_{\text{marked}} \cdot a_{RLm} , \\
a_{RLm} =_{df} a_{\text{right}} \cdot a_{LM} , \\
a_{LM} =_{df} a_{\text{left}} \cdot \neg a_{\text{left}} .
\]

The element \( a_{LM} \) is an algebraic representation of the loop while \( a_{\text{left}} \) do \( a_{\text{left}} \) that moves to the leftmost node corresponding to a given starting node in a tree. It has been shown in [22] that determinacy of a loop body is inherited by the corresponding while loop. Moreover, \( a_{\text{thread}} \) can be viewed as an algebraic counterpart of the conditional expression \( a_{\text{thread}} = \text{if } a_{\text{marked}} \text{ then } a_{\text{right}} \text{ else } a_{\text{left}} \cdot a_{LM} \); hence it is deterministic as well.

For the correct working of the algorithm we require the following structural properties of \( a \):
1. $a_{LR} =_{df} a_{\text{left}} + \neg \text{marked} \cdot a_{\text{right}}$ forms a tree, the tree underlying the threaded structure, with standard left/right links as reflected by the index $LR$;

2. $a_{\text{thread}}$ forms a chain;

3. the inorder sequence of $a_{LR}$ equals the traversal sequence of $a_{\text{thread}}$.

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

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

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

$$
[\text{tr}^{-}]_{(s,a)} =_{df} \text{li}_{a_{\text{marked}}}(s(i)) \quad \text{and} \quad [\text{inorder}(\text{tr}_{a_{LR}}(s(i)))]
$$

where $\text{tr}_{a}(p)$ for a tree $a$ is defined in Equation (11) and $\text{inorder}(T)$ returns the word consisting of the nodes of $T$ in the sequence of an inorder traversal of $T$.

A threaded tree can now be defined by the predicate

$$
\text{th\_tree}(i,j) =_{df} \text{lr\_tree}(i) \land \text{thread\_list}(j) \land j^{-} = i^{-}
$$

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

$$
\text{lm}_a(p) =_{df} \begin{cases} \quad \square & \text{if } p = \square, \\
\quad p & \text{if } (\{a_{\text{left}}(p) \cdot a = 0 \}, \text{leftmost}(i))_{(s,a)} =_{df} \text{lm}_a(s(i)). \\
\end{cases}
$$

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

**Lemma 11.1.** Assume predicates $P, Q \subseteq \text{tree}$ and identifiers $i,j$. Moreover, assume selector sets $K, M \subseteq L$ and a selector $l \in K - M$. Then

$$
\begin{align*}
\{ P(i) \oplus_K Q(j) \} & \quad \text{i\_l} := j; \\
\{ P(i) \oplus_{K-l} Q(j) \land P(i) \ominus_l Q(j = i.l) \} & \quad \text{j\_l} := i; \\
\{ P(i) \oplus Q(j) \land P(i) \ominus_i Q(j = i.l) \} & \quad \text{and} \quad \{ P(i) \oplus_M Q(j) \land Q(j) \ominus_l P(i = j.l) \}.
\end{align*}
$$

Proofs for these rules can be constructed similar to that of Lemma 9.1.

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 $l \not\in M$.

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:

$$
\begin{align*}
\text{mark}(i) & =_{df} \{ ((s,a),(s,(s(i) + a_{\text{marked}}) + a_{L-\text{marked}})) : i \in \text{dom}(s) \}, \\
i := \text{new m\_cell}() & =_{df} \{ ((s,a),(s[i \leftarrow l].(p \mapsto \circ)(a_{L-\text{marked}} + a_{\text{marked}})) : i \in \text{dom}(s), \\
p \text{ is an atomic test, } p \leq \neg i, p \neq \circ \}.
\end{align*}
$$
Before we can use it in the verification of the concrete example we give further inference rules.

**Lemma 11.2.** Assume identifiers $i, j, k$ and $i \neq \square \land k \neq \square$ then

\[
\{ \text{th_tree}(i = j) \odot \text{u_cell}(k) \} \quad \{ \text{u_cell}(k) \} \quad \text{mark}(k); \quad \text{and}
\]

\[
\{ \text{lr_tree}(i) \odot_{LR} \text{u_cell}(k) \} \wedge \text{thread_list}(j) \wedge k \cdot j^\rightarrow = i^\leftarrow \}
\]

These laws are direct consequences of the definition of \text{mark} and the abstraction functions in Equation (14).

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 $j \neq i$ heading the thread list. Then we can reason as follows.

\[
\{ \text{lr_tree}(i) \odot_{LR} \text{u_cell}(j) \wedge \text{thread_list}(j) \wedge j^\rightarrow = i^\leftarrow \wedge j^\leftarrow = \alpha \}
\]

\[
k := \text{new cell}();
\]

\[
\{ \text{lr_tree}(i) \odot_{LR} \text{u_cell}(j) \odot \text{u_cell}(k) \wedge \text{thread_list}(j) \odot \text{u_cell}(k) \wedge j^\rightarrow = i^\leftarrow \wedge j^\leftarrow = \alpha \}
\]

\[
j \cdot \text{left} := k.
\]

\[
\{ \text{lr_tree}(i) \odot_{LR} \text{u_cell}(k) \odot_{LR} \text{u_cell}(j) \odot \text{thread_list}(k) \odot \text{right u_cell}(k = j.\text{left}) \wedge k \cdot j^\rightarrow = i^\leftarrow \wedge j^\leftarrow = \alpha \}
\]

\[
k \cdot \text{right} := j;
\]

\[
\{ \text{lr_tree}(i) \odot_{LR} \text{u_cell}(j) \odot_{LR} \text{u_cell}(k = j.\text{left}) \wedge \text{u_cell}(k) \odot_{LR} \text{right thread_list}(j = k.\text{right}) \wedge k \cdot j^\rightarrow = \alpha \}
\]

\[
\text{mark}(k);
\]

\[
\{ \text{lr_tree}(i) \odot_{LR} \text{u_cell}(j) \odot_{LR} \text{m_cell}(k = j.\text{left}) \wedge \text{m_cell}(k) \odot_{LR} \text{thread_list}(j = k.\text{right}) \wedge k \cdot j^\rightarrow = \alpha \}
\]

\[
j := k;
\]

\[
\{ \text{lr_tree}(i) \wedge \text{thread_list}(j) \wedge j^\rightarrow = i^\leftarrow \wedge j^\leftarrow = \alpha \}
\]

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

\[
\text{dl_list}(i,j) = \text{df} \ \text{next_list}(i) \wedge \text{prev_list}(j) \wedge j^\rightarrow = (\text{df})^j
\]

where

\[
\text{next_list} = \text{df} \ \{ a : a_{next} \text{ is a chain} \}, \quad \text{prev_list} = \text{df} \ \{ a : a_{prev} \text{ is a chain} \}.
\]

and

\[
[i^\rightarrow]_{(s,a)} = \text{df} \ \text{li}_{a_{next}}(s(j)), \quad [(\text{df})^j]_{(s,a)} = \text{df} \ \text{li}_{a_{prev}}(s(j)).
\]

12. Related Work

There exist several approaches to extend SL by additional constructs that rule out sharing or restrict outgoing pointers of disjoint heaps to a single direction. Wang et al. [23] 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 [21] where, generally, 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 [4]. 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 [21]. 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.

13. 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 their derivation. 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 concrete examples 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 of 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 to treat concurrent garbage collection algorithms. Moreover, another objective is to investigate program verification tasks with the presented approach involving (semi)automated theorem proving tools. This would immediately improve practicality and raise the confidence in treatments based on transitive separation logic.

However, this will require substantial additional efforts for the following reasons. The basic laws about modal operators and reachability etc. from Sections 2, 3, 5 and 6 are expressible in “propositional” and one-sorted form; thus they can be readily translated to first-order logic and automatically verified using systems like PROVER9 [26] or any other systems accessible through the TPTP Library [27], at the level of the underlying resource algebra [28]. A substantial amount of such verification tasks has been carried out. However, the rules in Sections 8–11 that deal with programs would at least need a many-sorted framework such as ISABELLE, and fully general automation for such frameworks seems not available currently. Finding a suitable combination of convenient simple formalisation and full automation will be a challenging task for which, however, the present paper hopefully lays well usable foundations.

Acknowledgements: We thank reviewers of RAMiCS 2012 and JLAMP for their fruitful comments that helped to significantly improve the paper. This research was partially funded by the DFG project MO 690/9-1 AlgSep — Algebraic Calculi for Separation Logic.
References