Universität Augsburg

Algebraic Aspects of Separation Logic

Han-Hing Dang

Report 2009-01

May 2009

Institut für Informatik
D-86135 Augsburg

Contents

Chapter 1

Introduction

A lot of software systems nowadays run through long phases of development before they can be delivered. This process is very error-prone and especially in early phases of development software bugs become more expensive if they are found too late. Moreover such bugs yield an immense decrease in prestige for a company when they happen too often. Not least due to this, it is nowadays very important to develop correct software. Over the last four decades a lot of effort has been invested to ensure correctness of programs by using formal specification and verification methods. Some approaches for imperative programming languages like `C` have already been introduced by Hoare and Dijkstra to ensure partial or total correctness. But they lack expressiveness for shared mutable data structures, i.e., structures where updatable fields can be referenced from more than one point (e.g. [Rey08a]). Therefore Reynolds, O'Hearn and others have developed a novel system called separation logic which allows reasoning about such data structures. Their approach extended Hoare logic with assertions that express assumptions about separation in storage. Furthermore the command language is enriched by some constructs which allows altering these separate ranges. Over the last few years this logic has become a large research area and also has a variety of further applications e.g. in reasoning about concurrency ([HWW09, HMSW09]).

In this technical report we take another view of separation logic, using an algebraic approach to simplify proofs in this logic and to find new results. First we will give some basic definitions and notations. Afterwards we will have a closer look at the new assertion constructs for this logic and explain how they are to be understood. As a main result we found a possibility to treat them algebraically. Especially for some classes of assertions we will present an axiomatization and algebraically prove a lot of laws of [Rey08a]. Reynolds did not give proofs for them. Next we look more closely at the new command constructs to reference heap cells. To treat the commands algebraically, we introduce a relational view for them and define an algebraic structure in which we are able to work with them. Afterwards we introduce and explain some inference rules for these commands. Finally we give a concrete example of an in-place list reversal algorithm which we will verify using our algebraic approach.

Most of the laws given in Chapter 3 have been checked by the automatic theorem prover *Prover9* which, according to [DH08], is for our purposes the best theorem prover. We added an input file for the axioms in the Appendix and listed some of the goals which we have verified using Prover9.

Chapter 2

Basics and Notations

As already mentioned in the introduction, separation logic is an extension of Hoare logic by constructs which allows reasoning about separate storage parts. In Hoare logic, states only consist of a store used to save values of program variables. Now in separation logic states do not just consist of stores; they are extended by an additional heap component. The store of a state corresponds to the currently allocated registers and the the heap to the addressable memory ([Rey08a]). In the rest of this report we write s for stores and h for heaps. In [Rey08a] stores are defined as functions from variables into values of $\mathbb{Z}$ and heaps as functions from addresses into values of $\mathbb{Z}$, while addresses are also values of $\mathbb{Z}$. We will slightly deviate from this functional view and define for an algebraic treatment the stores and heaps as relations.

For this relational view it is useful to let every store or heap just map a variable or an address into a single value. This brings us immediately to the first definition where define partial functions using a relational view.

Definition 2.0.1 (Relational view of partial functions) A relation $R \subseteq M \times N$ is called a *partial function* from M to N iff

$$\forall\, x \in M : \forall\, y, z \in N : (x, y) \in R \,\wedge\, (x, z) \in R \,\Rightarrow\, y = z.$$

The set of all partial functions from M to N is denoted by $M \rightsquigarrow N$.

Now we can define our states with store and heap components as follows.

Definition 2.0.2 Let V be the set of all variables. Then we define

$$\begin{array}{rcl} Values & = & \mathbb{Z} \\ \{\mathsf{nil}\} \,\dot\cup\, Addresses & \subseteq & Values \\ Stores & = & V \rightsquigarrow Values \\ Heaps & = & Addresses \rightsquigarrow Values \\ States & = & Stores \times Heaps \end{array}$$

where $\dot\cup$ denotes the disjoint union on sets.

We write in this report $h(a)$ for an address a and $s(x)$ for a variable x to denote its according value. The constant nil is a value for pointers that denotes an improper reference like null in programming languages like Java. By this definition nil is not an address and therefore every heap $h \in Heaps$ satifies $\mathsf{nil} \notin \mathsf{dom}(h)$, where $\mathsf{dom}(\cdot)$ is defined as follows

Definition 2.0.3 (Domain of Relations) Let $R \subseteq M \times N$ be a relation. Then its domain $\mathsf{dom}(R)$ is defined by

$$\mathsf{dom}(R) =_{\mathit{df}} \{x \in M \mid \exists\, y : (x, y) \in R\}.$$

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

As in [Möl93] and for later definitions we also need an operator called *update* operator. This operator is used to model changes in stores and heaps. First we give a definition in the following and then explain its meaning.

Definition 2.0.4 (Update on partial functions) Let R and S be partial functions. Then we define

$$R \mid S =_{\mathit{df}} R \cup \{(x, y) \mid (x, y) \in S \wedge x \notin \mathsf{dom}(R)\}.$$

Hence the relation R updates the relation S with all possible tuples of R in such a way that $R \mid S$ is again a partial function. The righthand side of $\cup$ above is disjoint from the set R. In particular, $R \mid S$ can be seen as an extension of R to $\mathsf{dom}(R) \cup \mathsf{dom}(S)$. In later definitions to denote an update on a single variable or address, we abbreviate this and do not write the brackets in the notation. So we write $(x, v) \mid S$ for $\{(x, v)\} \mid S$.

2.1 Expressions

Before giving a detailed definition for assertions we need to define how expressions need to be understood in Separation Logic. This is important because they are ubiquitous for every assertion. The reader may find them straightforward to understand but to get a better comprehension about how they need to be evaluated, we defined them more precisely. An important fact is that every expression, used in separation logic in assertions, is independent of the heap. This means that they only need the store component of a given state to be evaluated. In particular, expressions in assertions are always defined and can not cause any fault or side effects when evaluated.

They are basically used for describing conditions on stores or on heaps with the help of special constructs which will be defined later when giving a precise definition for assertions. From the definitions of stores and heaps above we are able to perform simple arithmetics on values or addresses in expressions since we are working only with values of $\mathbb{Z}$.

In separation logic there exists two kinds of expressions, namely $\langle\text{exp}\rangle$-expressions and Boolean $\langle\text{bexp}\rangle$-expressions. In the remainder we will give intuitive definitions for them to explain their structure and their evaluation on stores. Afterwards we give a small example.

Definition 2.1.1 (Syntax for expressions)

$$\begin{array}{rcl}
\langle\text{var}\rangle & ::= & x \mid y \mid z \mid ... \\
\langle\text{exp}\rangle & ::= & 0 \mid 1 \mid 2 \mid ... \\
 & & \mid \langle\text{var}\rangle \mid -\langle\text{exp}\rangle \mid \langle\text{exp}\rangle + \langle\text{exp}\rangle \mid \langle\text{exp}\rangle - \langle\text{exp}\rangle \\
 & & \mid \langle\text{exp}\rangle \cdot \langle\text{exp}\rangle \mid ... \\
\langle\text{bexp}\rangle & ::= & \mathsf{true} \mid \mathsf{false} \\
 & & \mid \langle\text{exp}\rangle = \langle\text{exp}\rangle \mid \langle\text{exp}\rangle < \langle\text{exp}\rangle \mid ...
\end{array}$$

This means that $\langle\text{exp}\rangle$-expressions can only be variables, denoted by x, y, z, or values, i.e., numbers of $\mathbb{Z}$, or terms with the usual arithmetic operators over variables or natural numbers. Furthermore $\langle\text{bexp}\rangle$-expressions are Boolean expressions which are true, false or just some comparisons.

The semantics of expressions when evaluating them, using the store of a given state, is defined as follows.

Definition 2.1.2 (Semantics of expressions) Let s be an arbitrary store on which the following expressions are intended to be evaluated. Furthermore, let e_1, e_2 be $\langle\text{exp}\rangle$-expressions and $[op]$ be an arbitrary corresponding arithmetic or Boolean operator. Then we write $\langle\cdot\rangle_{\text{exp}}(s)$ for the evaluation of an arbitrary expression on s and inductively define it as follows

$$\begin{aligned}
\langle c\rangle_{\text{exp}}(s) = c \quad \forall c \in \mathbb{Z}\,, &\qquad \langle x\rangle_{\text{exp}}(s) = s(x) \quad \forall x \in \mathsf{dom}(s)\,,\\
\langle e_1\,[op]\,e_2\rangle_{\text{exp}}(s) &= \langle e_1\rangle_{\text{exp}}(s)\,[op]\,\langle e_2\rangle_{\text{exp}}(s) \in \mathbb{Z}\,,\\
\langle \mathsf{true}\rangle_{\text{bexp}}(s) = \mathsf{true}\,, &\qquad \langle \mathsf{false}\rangle_{\text{bexp}}(s) = \mathsf{false}\,,\\
\langle e_1\,[op]\,e_2\rangle_{\text{bexp}}(s) &= \langle e_1\rangle_{\text{exp}}(s)\,[op]\,\langle e_2\rangle_{\text{exp}}(s) \in \{\mathsf{true}, \mathsf{false}\}
\end{aligned}$$

We give a small example to how this evaluation is done.

Example 2.1.3 Let $V = \{x, y, z\}$ and $s = \{(x, 1), (y, 32), (z, 59)\}$. Some examples for $\langle\text{exp}\rangle$-expressions are

$$x + 1,\ y,\ z + x - 1\,.$$

Their corresponding values w.r.t. the store s are

$$\begin{aligned}
\langle x + 1\rangle_{\text{exp}}(s) &= s(x) + 1 = 2\\
\langle y\rangle_{\text{exp}}(s) &= s(y) = 32\\
\langle z + y - 1\rangle_{\text{exp}}(s) &= s(z) + s(y) - 1 = 90
\end{aligned}$$

Two simple examples for Boolean-expressions are

$$x + 1 < y,\ y = z + x - 1\,.$$

Their results, evaluated w.r.t. s, are

$$\begin{aligned}
\langle x + 1 < y\rangle_{\text{bexp}}(s) &= \big(s(x) + 1 < s(y)\big) = \big(2 < 32\big) = \mathsf{true}\\
\langle y = z + x - 1\rangle_{\text{bexp}}(s) &= \big(s(y) = s(z) + s(x) - 1\big) = \big(32 = 59 + 1 - 1\big) = \mathsf{false}
\end{aligned}$$

□

With the use of expressions we next introduce the assertion language of separation logic.

Chapter 3

Assertions

Assertions play an important role in separation logic. They are used as predicates to describe the contents of heaps and stores using the just defined expressions. Moreover, in separation logic, assertions are used as pre- or postconditions in programs, like in Hoare logic. The assertions in separation logic can be divided into the already known ones from predicate logic and into four new ones which express properties of the heap. We first give a definition for our assertions. Afterwards go into detail explaining the semantics of our new operators and how they are used in separation logic.

3.1 Definitions and Examples

We start by a definition of what an assertion can look like. We write p, q and r in the remainder to denote assertions. Afterwards we have a closer look on the new aspects of separation logic. Furthermore we present what is new in this logic and what is the main idea behind the new predicates characterising heaps.

Definition 3.1.1 (Syntax for assertions)

$$\begin{aligned}
\langle\text{assert}\rangle \;\; ::= \;\; & \langle\text{bexp}\rangle \\
& \mid \neg\, \langle\text{assert}\rangle \mid \langle\text{assert}\rangle \;\vee\; \langle\text{assert}\rangle \mid \langle\text{assert}\rangle \;\wedge\; \langle\text{assert}\rangle \\
& \mid \langle\text{assert}\rangle \leftrightarrow \langle\text{assert}\rangle \mid \langle\text{assert}\rangle \rightarrow \langle\text{assert}\rangle \\
& \mid \forall\, \langle\text{var}\rangle.\, \langle\text{assert}\rangle \mid \exists\, \langle\text{var}\rangle.\, \langle\text{assert}\rangle \\
& \mid \mathsf{emp} \\
& \mid \langle\text{exp}\rangle \;\mapsto\; \langle\text{exp}\rangle \\
& \mid \langle\text{assert}\rangle * \langle\text{assert}\rangle \\
& \mid \langle\text{assert}\rangle \mathrel{-\!\!*} \langle\text{assert}\rangle
\end{aligned}$$

Remark 3.1.2 We write $p \in \langle\text{bexp}\rangle$ for an arbitrary assertion p if it is a $\langle\text{bexp}\rangle$-expression. □

Hence, we have the usual Boolean connectives and four new ones including an atomic assertion named emp. We will explain their usefulness when we defined their semantics. To get a connection between states and assertions we inductively define a relation $\models$ according to [Rey08a].

We write $(s, h) \models p$ or briefly $s, h \models p$, if the state (s, h) satisfies the assertion p. Hence we have the pair $((s, h), p) \in \models$ iff p holds in the state (s, h). Otherwise we will write $(s, h) \not\models p$. Moreover we call an assertion p *valid* iff p holds in every state. And, as usual, if there exists a state (s, h) which satisfies p then we say the assertion p is *satisfiable*. Sometimes we only focus the heap h of a state (s, h) that satisfies an assertion p. Therefore we abbreviate: h satisfies p.

Subsequently we define the semantics for assertions in separation logic, including the just mentioned new connectives which give separation logic its name.

Definition 3.1.3 (Semantics for Assertions) Let e be an $\langle\text{exp}\rangle$-expression, b be an $\langle\text{bexp}\rangle$-expression and p, q be assertions then

$$\begin{array}{lll}
s, h \models b & \Leftrightarrow_{\mathit{df}} & \langle b \rangle_{\text{bexp}}(s) = \mathsf{true} \\
s, h \models \neg p & \Leftrightarrow_{\mathit{df}} & s, h \not\models p \\
s, h \models p \wedge q & \Leftrightarrow_{\mathit{df}} & s, h \models p \quad \text{and} \quad s, h \models q \\
s, h \models p \vee q & \Leftrightarrow_{\mathit{df}} & s, h \models p \quad \text{or} \quad s, h \models q \\
s, h \models p \rightarrow q & \Leftrightarrow_{\mathit{df}} & s, h \models p \quad \text{implies} \quad s, h \models q \\
s, h \models p \leftrightarrow q & \Leftrightarrow_{\mathit{df}} & s, h \models p \quad \text{iff} \quad s, h \models q \\
s, h \models \forall v.\ p & \Leftrightarrow_{\mathit{df}} & \forall x \in \mathbb{Z} : (v, x) \,|\, s, h \models p \\
s, h \models \exists v.\ p & \Leftrightarrow_{\mathit{df}} & \exists x \in \mathbb{Z} : (v, x) \,|\, s, h \models p \\
\hline
s, h \models \mathsf{emp} & \Leftrightarrow_{\mathit{df}} & h = \emptyset \\
s, h \models e \mapsto e' & \Leftrightarrow_{\mathit{df}} & h = \{(\, \langle e \rangle_{\text{exp}}(s) \,,\, \langle e' \rangle_{\text{exp}}(s) \,)\} \\
s, h \models p * q & \Leftrightarrow_{\mathit{df}} & \exists h_0, h_1 \in \mathit{Heaps} : \mathsf{dom}(h_0) \cap \mathsf{dom}(h_1) = \emptyset \quad \text{and} \\
 & & \quad h = h_0 \cup h_1 \quad \text{and} \quad s, h_0 \models p \quad \text{and} \quad s, h_1 \models q \\
s, h \models p \mathrel{-\!\!*} q & \Leftrightarrow_{\mathit{df}} & \forall h_0 \in \mathit{Heaps} : (\mathsf{dom}(h_0) \cap \mathsf{dom}(h) = \emptyset \quad \text{and} \quad s, h_0 \models p) \\
 & & \quad \text{implies} \quad s, h_0 \cup h \models q
\end{array}$$

The first eight definitions are standard and should be known from predicate logic or Hoare logic. They do not make any assumptions about the heap and only carry along the heap without making any changes to it. Only the semantics of quantifications over variables might need a closer look and more explanation.

Example 3.1.4 Consider the assertion $x = 1 \wedge \exists x.\ x = 2$ and a state with $s = \{(x, 1)\}$. One can see that $x = 1$ is satisfied. In $\exists x.\ x = 2$, the variable x is bound by the $\exists$ -quantifier and therefore x is only a placeholder for a value $v \in \mathbb{Z}$ and is not evaluated by the tuple $(x, 1)$. In general this means if s can be updated with a tuple (x, v) such that $x = 2$ holds then $\exists x. x = 2$ is satisfiable. □

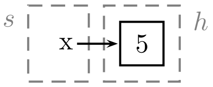
We want to avoid situations as in the example above and assume for the rest that every variable is uniquely used. That means it is either bound or free. And in the case of bound variables each quantifier uses different variable names. We call $FV(p)$, the set of all free variables of an assertion p, i.e., all variables occurring in an assertion p, that are not bound by any quantification.

In the above definition, the last four lines describe the new parts in separation logic. We explain them in the following more detailed. For an arbitrary state (s, h), emp ensures that the heap h is empty and contains no addressable cells. For a single heap cell the assertion $e \mapsto e'$ is used, where e and e' denotes expressions. It assures that there exists only one cell at the address $\langle e \rangle_{\text{exp}}(s)$ on the heap which has the value $\langle e' \rangle_{\text{exp}}(s)$. To build up more complex heaps, the novel operator $*$ is used. It is called the *separating conjunction* by Reynolds and lies at the heart of separation logic. One simple idea behind this operator is to join two disjoint heaps into a larger one. Two heaps h and h' are disjoint if $\mathsf{dom}(h) \cap \mathsf{dom}(h') = \emptyset$ holds for them, i.e., if they do not map from common addresses into values. As in each computer for example, there must be only one value assigned to each address allocated on the heap. One may wonder then why the case

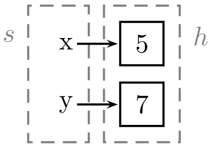
$$\mathsf{dom}(h) \cap \mathsf{dom}(h') \neq \emptyset \quad \wedge \quad \forall\, x \in \mathsf{dom}(h) \cap \mathsf{dom}(h') \subseteq \mathit{Addresses} : h(x) = h'(x)$$

is not included in the definition of $*$, since it should also be possible to unite such two heaps. But in fact, it is intended not to be able to combine such two heaps, since the general purpose for this operator is to separate the heap h of a state (s, h) into two disjoint portions. Or more exactly, let p and q be two arbitrary assertions and (s, h) a state satisfying the assertion $p * q$ then it is possible to divide h into two disjoint heaps such that p holds for one and q for the other part. But this is not possible anymore if the above case is allowed. Obviously it is not clear anymore to which heap the overlapping part might belong. This is why the $*$ operator is called *separating conjunction.*

Example 3.1.5 As a simple example to illustrate the $*$ operation, we start by a state which satisfies $x \mapsto 5$. The heap then maps the address $s(x)$ into the value 5.



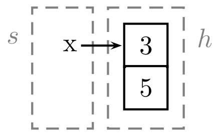
The variable x has a value $s(x)$ in the store s whereas $s(x)$ denotes an address on the heap. Using the separating conjunction $*$, we are able to combine another disjoint heap cell, satisfying $y \mapsto 7$, with the existing one. Again it is important that x and y are in $\mathsf{dom}(s)$ of the state and therefore are defined. The reader should also keep in mind that the conjunction of two disjoint heaps requires that both states on which the assertions are satisfied must agree on their store components. Then the resulting state is illustrated in the following picture



and satisfies $(x \mapsto 5) * (y \mapsto 7)$. The assertions $x \mapsto 5$ and $y \mapsto 7$ hold for two disjoint parts of the heap and imply immediately that $s(x) \neq s(y)$. Otherwise one would get a contradiction to the assumption that h is a partial function. To see that $*$ cannot be replaced by $\wedge$, consider next the assertion $x \mapsto 5 \wedge y \mapsto 7$ and a state (s, h) satisfying the assertion. By definition $h = \{(s(x), 5)\}$ and $h = \{(s(y), 7)\}$ has to hold. But this is false for every state because even if $s(x) = s(y)$, i.e., if the domain of the heap is exactly determined. We would immediately get the just mentioned contradiction.

Of course, one can guess that an assertion like $(x \mapsto 5) * (x \mapsto 5)$ similarly does not hold for any state, since it is not possible to let two disjoint heaps map from the same address $s(x)$. We can conclude that the separating conjunction is not generally an idempotent operation.

For two consecutive cells in a heap, e.g. a heap satisfying $(x \mapsto 3) * ((x+1) \mapsto 5)$, we will abbreviate this in the following and write $x \mapsto 3, 5$ instead. More intuitively, such a state is illustrated in the following picture



For the cell containing 5 it should be clear that it is reachable by $x+1$. □

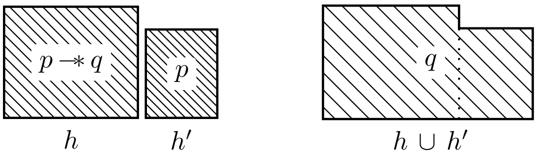
We generalize situations with adjacent heap cells as in the example above and give a notation in the following remark.

Remark 3.1.6 Let $e_1, \ldots, e_n$ and e be expressions then $e \mapsto e_1, \ldots, e_n$ abbreviates $(e \mapsto e_1) * \big(((e+1) \mapsto e_2) * \big(\ldots * ((e+n) \mapsto e_n)\big)\big)$. The expressions $e_1, \ldots, e_n$ denote the contents of n consecutive cells starting from address e.

In the following we will use the convention that $\mapsto$ binds stronger than $*$ and write $x \mapsto e * y \mapsto e'$ instead of $(x \mapsto e) * (y \mapsto e')$. □

To point out some advantages of these novel operators, we will later give a larger example when reasoning about programs.

Next we explain the *separating implication* $-\!\!*$, as it is named by Ishtiaq and O'Hearn ([Rey08a]), and give an example to get an idea of this operation. Consider a heap h in which $p -\!\!* q$ holds and a disjoint heap h' which satisfies the assertion p. This situation is illustrated on the left side of the picture below.



If we extend the heap h with the heap h', then the resulting heap $h \cup h'$ satisfies the assertion q. This is illustrated on the right side of the figure[1]. We give in the following a simple explicit example. In general cases the situation becomes more complicated.

Example 3.1.7 Consider the following assertions

$$\begin{aligned} p &= (x+1) \mapsto 5\,, \\ q &= x \mapsto 3 * (x+1) \mapsto 5\,, \end{aligned}$$

then by extending a heap in which $(x+1) \mapsto 5 -\!\!*[x \mapsto 3 * (x+1) \mapsto 5]$ holds with a disjoint heap satisfying $(x+1) \mapsto 5$, we obtain a heap that satisfies $x \mapsto 3*(x+1) \mapsto 5$. But as we know from the definitions above, a heap satisfying $(x+1) \mapsto 5$ has only one single cell and therefore is determined exactly by a given store s. Furthermore the assertions $x \mapsto 3$ and $x+1 \mapsto 5$ always describe disjoint non-overlapping portions of a given heap, so that the assertion $x \mapsto 3 * (x+1) \mapsto 5$ always describes a union of two disjoint heaps. In fact, in this case

$$(x+1) \mapsto 5 -\!\!* \big(x \mapsto 3 * (x+1) \mapsto 5\big) \Leftrightarrow x \mapsto 3$$

holds by the definition of $-\!\!*$. But of course this is just a simple example and generally $p -\!\!*(q * p) \Leftrightarrow q$ for arbitrary assertions p, q does not hold. □

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

We will discuss and sum up some general laws for the newly introduced heap operations in the next chapter.

Next we introduce an algebraic view for the just defined assertions. The main idea is to have a set-theoretic view on this logic by calculating with sets of states instead of continuing to use the predicates for the states. We will see that most of the laws axiomatized in [Rey08a] can be easily proved algebraically and be handled more concisely.

3.2 An Algebraic Treatment

To get the just mentioned mechanism under control we define sets of states on which we are going to calculate. Let p be an arbitrary assertion then we define by $[\![p]\!] =_{df} \{(s,h) : s,h \models p\}$ the set of all states that satisfy p, i.e., $(s,h) \in [\![p]\!] \Leftrightarrow s,h \models p$. By writing the brackets $[\![\cdot]\!]$ around an assertion p, it is meant to emphasize that we are using sets of states. We will later use the letters p, q to denote sets of states when calculating in our algebraic setting, so the reader should keep in mind that by writing p in later algebraic proofs, we mean a set of states that satisfy p and not the assertion itself anymore.

The usual Boolean connectives in this logic can be handled as already known. We can view the disjunction of two assertions in a set-based view for the $\models$ - semantics as a union of two sets. The same holds for the conjunction which can be modeled by set intersection. Since the complete assertion language is Boolean we are able to define the negation of an assertion as the complement on a Boolean lattice.

To model the separating conjunction $*$ algebraically let $\mathcal{P}(States)$ be the power set of $States$. Then we define an operator $\mathbin{\dot\cup} : States \times States \to \mathcal{P}(States)$ which has a similar semantics as the $*$ in separation logic. This operator is an algebraic version of $*$ with the difference that it will be used for calculating with sets of states. Consider two states (s,h) and (s',h'). The disjoint union $\mathbin{\dot\cup}$ on heaps between two states is defined by

$$(s,h) \mathbin{\dot\cup} (s',h') =_{df} \begin{cases} \{(s, h \cup h')\} & \text{if } \mathsf{dom}(h) \cap \mathsf{dom}(h') = \emptyset \wedge s = s' \\ \emptyset & \text{otherwise.} \end{cases}$$

By this definition the second case stands for an error case; if the heaps are not disjoint, their union is not defined and thus the operator will return $\emptyset$. Consider now the *pointwise extension* of this operator denoted by the same operator with functionality $\mathbin{\dot\cup} : \mathcal{P}(States) \times \mathcal{P}(States) \to \mathcal{P}(States)$ which is defined by

$$[\![p]\!] \mathbin{\dot\cup} [\![q]\!] =_{df} \bigcup_{(s,h) \in [\![p]\!]} \bigcup_{(s',h') \in [\![q]\!]} (s,h) \mathbin{\dot\cup} (s',h')$$

for $[\![p]\!], [\![q]\!] \in \mathcal{P}(States)$. By this definition, the extended operator distributes over union, i.e.,

$$\bigcup \chi_1 \mathbin{\dot\cup} \bigcup \chi_2 =_{df} \bigcup \{[\![p]\!] \mathbin{\dot\cup} [\![q]\!] : [\![p]\!] \in \chi_1, [\![q]\!] \in \chi_2\}$$

for $\chi_1, \chi_2 \subseteq \mathcal{P}(States)$. In fact, an analogous proof shows that it even distributes over arbitrary unions. By taking $\chi_1 = \emptyset$ or $\chi_2 = \emptyset$ we obtain strictness of the pointwise extension with respect to $\emptyset$ in both arguments:

$$\emptyset \mathbin{\dot\cup} [\![p]\!] = \emptyset \qquad [\![p]\!] \mathbin{\dot\cup} \emptyset = \emptyset$$

Furthermore by taking $\chi_1 = \{[\![p_1]\!], [\![p_2]\!]\}$, $\chi_2 = \{[\![q_1]\!], [\![q_2]\!]\}$ and using the equivalence

$$\llbracket p_1 \rrbracket \subseteq \llbracket p_2 \rrbracket \Leftrightarrow \llbracket p_1 \rrbracket \cup \llbracket p_2 \rrbracket = \llbracket p_2 \rrbracket$$

(analogously for q_1 and q_2) one also obtains isotony with respect to $\subseteq$, i.e.,

$$\llbracket p_1 \rrbracket \subseteq \llbracket p_2 \rrbracket \land \llbracket q_1 \rrbracket \subseteq \llbracket q_2 \rrbracket \Rightarrow \llbracket p_1 \rrbracket \uplus \llbracket q_1 \rrbracket \subseteq \llbracket p_2 \rrbracket \uplus \llbracket q_2 \rrbracket$$

Finally, bilinear equational laws for $\uplus$, i.e., laws in which each side has at most one occurence of every variable (i.e. a set of states) are inherited by the pointwise extension. Consider e.g. the assertions p, q and r, then examples of such laws are

$$\begin{array}{rl} (\llbracket p \rrbracket \uplus \llbracket q \rrbracket) \uplus \llbracket r \rrbracket = \llbracket p \rrbracket \uplus (\llbracket q \rrbracket \uplus \llbracket r \rrbracket) & \text{(associativity)} \\ \llbracket p \rrbracket \uplus \llbracket q \rrbracket = \llbracket q \rrbracket \uplus \llbracket p \rrbracket & \text{(commutativity)} \\ \llbracket p \rrbracket \uplus \llbracket \mathsf{emp} \rrbracket = \llbracket q \rrbracket & \text{(neutrality)} \end{array}$$

where $\llbracket \mathsf{emp} \rrbracket$ denotes the set of all states with an empty heap component.

Now we present our basic definition for an algebraic treatment of assertions in separation logic.

Definition 3.2.1 (Set-based view on assertions) Let e and e' be $\langle\text{exp}\rangle$-expressions and p, q be arbitrary assertions and v a variable then the set-based view for assertion is defined as follows

$$\begin{array}{lll} \llbracket p \rrbracket & =_{df} & \{(s,h) : s,h \models p,\ p \in \langle\text{bexp}\rangle\} \\ \llbracket \neg p \rrbracket & =_{df} & \{(s,h) : s,h \not\models p\} = \overline{\llbracket p \rrbracket} \\ \llbracket p \lor p \rrbracket & =_{df} & \llbracket p \rrbracket \cup \llbracket q \rrbracket \\ \llbracket p \land q \rrbracket & =_{df} & \llbracket p \rrbracket \cap \llbracket q \rrbracket = \overline{\overline{\llbracket p \rrbracket} \cup \overline{\llbracket q \rrbracket}} \\ \llbracket p \rightarrow q \rrbracket & =_{df} & \llbracket \neg p \lor q \rrbracket = \overline{\llbracket p \rrbracket} \cup \llbracket q \rrbracket \\ \llbracket p \leftarrow q \rrbracket & =_{df} & \llbracket q \rightarrow p \rrbracket = \overline{\llbracket q \rrbracket} \cup \llbracket p \rrbracket \\ \llbracket p \leftrightarrow q \rrbracket & =_{df} & \llbracket (p \rightarrow q) \land (q \rightarrow p) \rrbracket = (\overline{\llbracket p \rrbracket} \cup \llbracket q \rrbracket) \cap (\overline{\llbracket q \rrbracket} \cup \llbracket p \rrbracket) \\ \llbracket \exists v.\ p \rrbracket & =_{df} & \{(s,h) : \exists x \in \mathbb{Z} .\ (v,x) \mid s,h \models p\} \\ \llbracket \forall v.\ p \rrbracket & =_{df} & \{(s,h) : \forall x \in \mathbb{Z} .\ (v,x) \mid s,h \models p\} \\ \hline \llbracket \mathsf{emp} \rrbracket & =_{df} & \{(s,h) : h = \emptyset\} \\ \llbracket e \mapsto e' \rrbracket & =_{df} & \{(s,h) : h = \{(\langle e \rangle_{\text{exp}}(s), \langle e' \rangle_{\text{exp}}(s))\}\ \} \\ \llbracket p * q \rrbracket & =_{df} & \{(s,h) \uplus (s',h') : (s,h) \in \llbracket p \rrbracket, (s',h') \in \llbracket q \rrbracket\} \\ & = & \llbracket p \rrbracket \uplus \llbracket q \rrbracket \\ \llbracket p -\!\!* q \rrbracket & =_{df} & \{(s,h) : \forall h' \in \mathit{Heaps} :\ (\mathsf{dom}(h) \cap \mathsf{dom}(h') = \emptyset \land (s,h') \in \llbracket p \rrbracket) \\ & & \qquad \Rightarrow (s, h \cup h') \in \llbracket q \rrbracket\} \end{array}$$

As already mentioned, most of the usual Boolean connectives can be modeled by union and intersection on sets of states. An exception are the quantification in assertions. Later we will discuss how these situations can be treated.

Furthermore the separating conjunction is now modeled with the mechanism described above. Only the definition for the separating implication still looks difficult to get under control. Later we give some laws ([Rey08a]) for this operation and finally explain how this operator can be axiomatized using our algebraic approach.

3.3 The Algebraic Structure

To keep the proofs for the later given laws simple, we introduce some definitions for the used algebraic structure and give some proof principles. First of all we give definitions for a *semiring* and a relation $\leq$ for an ordering in this structure.

Definition 3.3.1 (Semiring) A *semiring* is a structure $(S, +, 0, \cdot, 1)$ such that $(S, +, 0)$ is a commutative monoid and $(S, \cdot, 1)$ is a monoid. Furthermore multiplication distributes over addition in both arguments, i.e., for arbitrary x, y, $z \in S$

$$(x + y) \cdot z = x \cdot z + y \cdot z \qquad z \cdot (x + y) = z \cdot x + z \cdot y$$

and 0 is a left and right annihilator with respect to multiplication, i.e.,

$$x \cdot 0 = 0 = 0 \cdot x$$

We call a semiring *idempotent* if the addition satisfies

$$x + x = x$$

and *commutative* if multiplication satisfies

$$x \cdot y = y \cdot x$$

Definition 3.3.2 (Natural order) In an *idempotent semiring* S the relation $\leq$ is defined by $x \leq y \Leftrightarrow_{df} x + y = y$. This relation is a partial order and has 0 as its least element. Multiplication and addition are isotone in both arguments. Therefore it is called the natural ordering on S. This makes S into a semilattice with addition as join and 0 as its least element.

With these definitions we can now go on and define our mainly used algebraic structure, a *quantale* or *standard Kleene Algebra* as it is also called ([Con71, Mul86]).

Definition 3.3.3 (Quantale) A quantale is an idempotent semiring that is a complete lattice under the natural order and in which multiplication distributes over arbitrary suprema. The supremum of a subset S is denoted by $\sqcup S$ and the infimum by $\sqcap S$. In particular, the greatest element is denoted by $\top$, i.e., $p \leq \top$ for arbitrary elements p. Furthermore if the underlying lattice is a Boolean algebra we call it a *Boolean quantale.*

Next we introduce an important tool which we will use in proofs. For proving inequalities we use the principle of *indirect inequality*

$$\begin{gathered} x \leq y \Leftrightarrow (\forall\, z : y \leq z \Rightarrow x \leq z) \\ x \leq y \Leftrightarrow (\forall\, z : z \leq x \Rightarrow z \leq y) \end{gathered}$$

and the proof principle of *indirect equality* for proving equalities

$$x = y \Leftrightarrow (\forall\, z : x \leq z \Leftrightarrow y \leq z) \Leftrightarrow (\forall\, z : z \leq x \Leftrightarrow z \leq y)$$

The ($\Rightarrow$) directions for the inequality proof principle follows immediately from transitivity of $\leq$. For the reverse direction set $z = y$ or $z = x$. The proof principle for equality then holds immediately by antisymmetry of the order $\leq$.

Finally, we can explain how the assertions or the sets of states can be embedded into such an algebraic structure. The union on sets coincides with the addition in an idempotent semiring and so does the relation $\leq$ with the relation $\subseteq$. Hence $(S, \cup, \emptyset)$ is a commutative monoid. To define the intersection $\cap$ on sets we will use the following equivalence

$$z \leq x \sqcap y \Leftrightarrow_{df} z \leq x \wedge z \leq y$$

It is a natural definition and allows us to use intersection in absence of negation, i.e., if we are not in a Boolean quantale. Furthermore, since the separating conjunction is only a union on disjoint heaps, it is clear that it is commutative, associative and has the empty heap as its neutral element. We sum this result up in the next lemma.

Lemma 3.3.4 *The separating conjunction $*$ induces a commutative monoid with* emp *as its neutral element, i.e., for arbitrary assertions p, q and r the following laws hold :*

$$\begin{array}{rcl} p * q & \Leftrightarrow & q * p \\ (p * q) * r & \Leftrightarrow & p * (q * r) \\ p * \mathsf{emp} & \Leftrightarrow & p \end{array}$$

Of course, the same holds for our new operator $\uplus$, since it is only a union on heaps. By applying the mechanism of pointwise extension to it, we can view $\uplus$ as the multiplication in a quantale with $[\![\mathsf{emp}]\!]$ as its neutral element. As already mentioned, we know that pointwise extension lifts up laws like associativity, commutativity and neutrality. Furthermore it is strict in both arguments with respect to $\emptyset$. Therefore we have $\emptyset$ as a left and right annihilator. To see that $\uplus$ distributes over arbitrary suprema we give a short proof for it in separation logic. But as mentioned above, we have $+$ as the join denoting the supremum $\sqcup$. And since we are again only calculating on union of sets, the only problem that could arise is that we are calculating on both, states and heaps of states. The following proof in separation logic shows that the distributivity law in fact holds. We will use the abbreviation $h - h_0 =_{df} h \cap \overline{h_0}$ as the usual known difference on sets.

$$\begin{array}{ll} & s, h \models (p \vee q) * r \\ \Leftrightarrow & \quad \{\![\text{ definition of } *]\!\} \\ & \exists\, h_0 : h_0 \subseteq h \quad \text{and} \quad s, h_0 \models p \vee q \quad \text{and} \quad s, h - h_0 \models r \\ \Leftrightarrow & \quad \{\![\text{ definition of } \vee]\!\} \\ & \exists\, h_0 : h_0 \subseteq h \quad \text{and} \quad (s, h_0 \models p \quad \text{or} \quad s, h_0 \models q) \quad \text{and} \quad s, h - h_0 \models r \\ \Leftrightarrow & \quad \{\![\text{ logic }]\!\} \\ & \exists\, h_0 : h_0 \subseteq h \quad \text{and} \quad ((s, h_0 \models p \quad \text{and} \quad s, h - h_0 \models r) \quad \text{or} \\ & \qquad\qquad\quad (s, h_0 \models q \quad \text{and} \quad s, h - h_0 \models r)) \\ \Leftrightarrow & \quad \{\![\text{ logic }]\!\} \\ & (\exists\, h_0 : h_0 \subseteq h \quad \text{and} \quad (s, h_0 \models p \quad \text{and} \quad s, h - h_0 \models r)) \quad \text{or} \\ & \qquad\qquad (\exists\, h_0 : h_0 \subseteq h \quad \text{and} \quad (s, h_0 \models q \quad \text{and} \quad s, h - h_0 \models r)) \\ \Leftrightarrow & \quad \{\![\text{ definition of } *]\!\} \\ & s, h \models (p * r) \quad \text{or} \quad s, h \models (q * r) \\ \Leftrightarrow & \quad \{\![\text{ definition of } \vee]\!\} \\ & s, h \models (p * r) \vee (q * r) \end{array}$$

Therefore by pointwise extension also $(\llbracket p \rrbracket \cup \llbracket q \rrbracket) \mathbin{\dot{\cup}} \llbracket r \rrbracket = (\llbracket p \rrbracket \mathbin{\dot{\cup}} \llbracket r \rrbracket) \cup (\llbracket q \rrbracket \mathbin{\dot{\cup}} \llbracket r \rrbracket)$ holds. A similiar argument then shows that $\dot{\cup}$ in fact distributes over arbitrary suprema. Moreover since every state satisfies the assertion true, we have $\llbracket \mathsf{true} \rrbracket$ as the greatest element in the algebraic setting. For this reason $\llbracket \mathsf{true} \rrbracket = \top$ holds. One should note that $\llbracket \mathsf{true} \rrbracket$ is also the neutral element for $\sqcap$.

As an important conclusion we sum up the just discussed results in the next lemma.

Lemma 3.3.5 *The structure* $(\mathcal{P}(\mathit{States}), \cup, \emptyset, \dot{\cup}, \llbracket \mathsf{emp} \rrbracket, \neg)$ *forms a Boolean quantale.*

In the following we will assume this algebraic structure when calculating algebraically. We will also assume where needed a Boolean quantale and write for negation $\overline{\cdot}$. It can be axiomatised by $p = \overline{\overline{p} + \overline{q}} + \overline{\overline{p} + q}$ for arbitrary elements p and q ([Hun33]). We continue in next chapter by explaining some useful laws for the new operations and show how the separating implication $-\!\!*$ can be axiomatized in our algebra.

3.4 Laws for the New Heap Operations

We start by some subdistributive and distributive laws which we will axiomate in our algebra.

Theorem 3.4.1 *The separating conjunction $*$ satisfies the following distributivity and subdistributivity laws for arbitrary assertions p, q and r. Furthermore we assume that the variable v is not free in q:*

$$\begin{array}{rcl}
(p \vee q) * r & \Leftrightarrow & (p * r) \vee (q * r) \\
(p \wedge q) * r & \Rightarrow & (p * r) \wedge (q * r) \\
(\exists\, v.\ p) * q & \Leftrightarrow & \exists\, v.\ (p * q) \\
(\forall\, v.\ p) * q & \Rightarrow & \forall\, v.\ (p * q)
\end{array}$$

A proof for the distributivity law $(p \vee q) * r \Leftrightarrow (p * r) \vee (q * r)$ has already been given before in separation logic. Next we will give an algebraic proof for $(p \wedge q) * r \Rightarrow (p * r) \wedge (q * r)$. This law can be encoded in a quantale by $(p \sqcap q) \cdot r \leq (p \cdot r) \sqcap (q \cdot r)$, whereas $\Rightarrow$ is algebraically encoded as $\leq$. We also write in the following p, q and r for elements of our quantale. They denote as mentioned above, sets of states on which we calculate.

$$\begin{array}{ll}
 & (p \sqcap q) \cdot r \\
= & \quad \{\!\!\{ \text{ idempotence of } \sqcap \text{ } \}\!\!\} \\
 & ((p \sqcap q) \cdot r) \sqcap ((p \sqcap q) \cdot r) \\
\leq & \quad \{\!\!\{ \text{ isotony of } \sqcap \text{ } \}\!\!\} \\
 & (p \cdot r) \sqcap (q \cdot r)
\end{array}$$

The other direction does not hold in general. We will give here a short counterexample according to [Rey08a] and explain the situation. Consider a state (s, h) with $s = \{(x, 5), (y, 6)\}$ and $h = \{(5, 10), (6, 20)\}$. Then the right hand side with the assertions $p = x \mapsto 10$, $q = y \mapsto 20$ resolves to

$$(x \mapsto 10 * (x \mapsto 10 \vee y \mapsto 20)) \wedge (y \mapsto 20 * (x \mapsto 10 \vee y \mapsto 20))$$

which is satisfiable on the given state, but the left hand side

$$(x \mapsto 10 \ \wedge \ y \mapsto 20) \ * \ (x \mapsto 10 \ \vee \ y \mapsto 20)$$

is not satisfied by any state. The reason therefore can be seen by writing the assertion as

$$((x \mapsto 10 \ \wedge \ y \mapsto 20) \ * \ x \mapsto 10) \ \vee \ ((x \mapsto 10 \ \wedge \ y \mapsto 20) \ * \ y \mapsto 20)$$

using the fact that $*$ distributes over $\vee$. Then if the heap can be split into a part which satisfies $x \mapsto 10 \ \wedge \ y \mapsto 20$, the rest of the heap cannot contain cells where $x \mapsto 10$ or $y \mapsto 20$ are satisfied since by the separating conjunction the portions have to be disjoint and both assertions have to agree on the same store. For a certain subclass of assertions this law becomes a full distributivity law. We will discuss this subclass later and explain why the full law holds in this class.

The next laws denote import and export rules over variable quantifications in assertions. We will continue by explaining these laws and their algebraic background. First, looking at the semantics of these assertions, they describe that under a certain condition an assertion can be imported into or or exported out of the binding range of a variable quantification. The first law is a full distributive law and therefore denotes a portation of q in both directions if v is not a free variable in q, i.e., if it has no occurrence in q or is bound by another quantifier.

To see that this law holds we will give again a short proof in separation logic.

$$\begin{array}{ll}
 & s, h \models (\exists\, v.\ p) \ * \ q \\
\Leftrightarrow & \quad \{\!\![\text{ definition of } * \ ]\!\!\} \\
 & \exists\, h_0 : h_0 \subseteq h \quad \text{and} \quad s, h_0 \models \exists\, v.\ p \quad \text{and} \quad s, h - h_0 \models q \\
\Leftrightarrow & \quad \{\!\![\text{ definition of } \exists\, v.\ ]\!\!\} \\
 & \exists\, h_0 : h_0 \subseteq h \quad \text{and} \quad (\exists\, x \in \mathbb{Z} : \quad (v, x) \,|\, s, h_0 \models p) \quad \text{and} \quad s, h - h_0 \models q \\
\Leftrightarrow & \quad \{\!\![\ v \text{ is not free in } q \ ]\!\!\} \\
 & \exists\, h_0 : h_0 \subseteq h \quad \text{and} \quad (\exists\, x \in \mathbb{Z} : \quad (v, x) \,|\, s, h_0 \models p \quad \text{and} \quad (v, x) \,|\, s, h - h_0 \models q) \\
\Leftrightarrow & \quad \{\!\![\text{ logic }]\!\!\} \\
 & \exists\, x \in \mathbb{Z} : \quad (\exists\, h_0 : h_0 \subseteq h \quad \text{and} \quad ((v, x) \,|\, s, h_0 \models p \quad \text{and} \\
 & \qquad\qquad\qquad\qquad\qquad\qquad (v, x) \,|\, s, h - h_0 \models q)) \\
\Leftrightarrow & \quad \{\!\![\text{ definition of } * \ ]\!\!\} \\
 & \exists\, x \in \mathbb{Z} : \quad (v, x) \,|\, s, h \models p \ * \ q \\
\Leftrightarrow & \quad \{\!\![\text{ definition of } \exists\, v.\ ]\!\!\} \\
 & s, h \models \exists\, v.\ (p \ * \ q)
\end{array}$$

One can see that the main issue of this proof lies in the modification of the store s such that q still holds with the store $(v, x) \,|\, s$. This is of course possible, since q does not use v as a free variable. In particular, quantification in separation logic always only makes assumptions over the store s of a state.

The last of the laws in Theorem 3.4.1 is again a subdistributive law. So it only denotes an import of an assertion q into the binding range of a $\forall$-quantifier. It says that if the variable v is not free in q then the store component of a state satisfying q can be modified according to the definition of $\forall$ such that q still holds for the modified state. We will give here no proof for this law, since the proof is very similar to the last one given for the $\exists$-quantification. Instead of this, we only give another counterexample for the reverse direction of this law. Consider now a state (s, h) with e.g.[2] $s = \{(y, 5)\}$ and $h = \{(x, 10) : x \in \mathit{Addresses}\}$, i.e., a heap where every cell is initialized with the value

[2] The store is arbitrarily chosen in this example, since we do not need store variables here.

10. We set in the law $\forall x.\ (p * q) \Rightarrow (\forall x.\ p) * q$, where x is free in q, $p = x \mapsto 10$ and $q = \exists y.\ y \mapsto 10$. Then again

$$\forall x.\ (x \mapsto 10 * (\exists y.\ y \mapsto 10))$$

holds for our given state, since we can find for every heap cell containing the value 10 another heap cell with the same value. But up on a closer look at the assertion

$$(\forall x.\ x \mapsto 10) * (\exists y.\ y \mapsto 10)$$

one can see that it does not hold. The reason for this is that already $\forall x.\ x \mapsto 10$ characterizes our whole heap h. Therefore there cannot exist a disjoint heap with a cell containing the value 10. As discussed above this lack can be fixed in a subclass of assertions which we discuss later.

Another useful law, which is very important and ubiquitous in our algebraic proofs is isotony with respect to $*$. This law will facilitate a lot of calculations. With the pointwise extension mechanism we are able to lift up this law to sets of states on which we calculate.

Lemma 3.4.2 *The separating conjunction $*$ is isotone, i.e., for arbitrary assertions p, q, r and s:*

$$\frac{p \Rightarrow r \quad q \Rightarrow s}{p * q \Rightarrow r * s}$$

To understand this law, one can imagine that by the assertion $p * q$ we characterize two disjoint parts of a heap. If the part satisfying p also satisfies r, then again the union of the disjoint parts will also satisfy $r * q$, since by $*$ we only unite both parts and the union on sets is also a isotone operator as discussed above.

A general meaning of this law is that we can weaken the assertion on the left hand or right hand side of $*$. This is especially important for proving correctness of programs, since like in Hoare logic we will often have to strengthen or weaken the pre- and postconditions of inference rules, so they can be applied.

In our algebraic setting this law is encoded as

$$p \leq r \wedge q \leq s \Rightarrow p \cdot q \leq r \cdot s$$

By calculating on sets algebraically we can interpret p, q, r and s as sets of states. Then the meaning is: if p is a subset of r and q a subset of s then the set $p \cdot q$ is still a subset of the set $r \cdot s$.

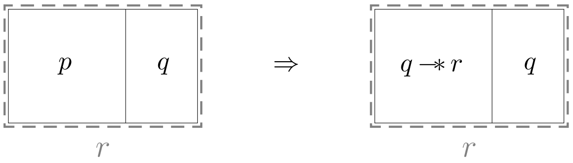
The last laws we discuss in this section are laws for a characterization of the separating implication $-\!\!*$ using the interplay of $-\!\!*$ with the separating conjunction $*$ according to [Rey08a]. In the following we will first explain these laws and then conclude by giving an algebraic characterization for this operator.

Theorem 3.4.3 *The separating implication $-\!\!*$ satisfies the following inference rules for arbitrary assertions p, q and r according to [Rey08a]:*

$$\frac{p * q \Rightarrow r}{p \Rightarrow (q -\!\!* r)} \quad \text{(currying)}$$

$$\frac{p \Rightarrow (q -\!\!* r)}{p * q \Rightarrow r} \quad \text{(decurrying)}$$

The first law (named currying law) says that if a heap can be split into two disjoint parts such that p holds for one part, q holds for the rest while the whole heap also satisfies r then that part of the whole heap which satisfies p will just as well satisfy $q -\!\!* r$. Again, to get an intuition for $q -\!\!* r$, consider a heap h that satisfies r. Then $q -\!\!* r$ characterizes the part of h where the portion that satisfies q is cut out. Thus $q -\!\!* r$ describes a residual part of the heap.



The decurrying law is the other direction of the just discussed law. That means: if a heap that satisfies p is also a residual heap where $q -\!\!* r$ holds, then r holds for the union of this heap and a heap that satisfies q.

This characterization of $-\!\!*$ via $*$ reminds us of the definition of algebraic residuals (see e.g. [Möl05]). In a quantale, residuals represents the largest solution p for the inequation $p \cdot q \le r$ and are defined by a Galois connection as for $-\!\!*$ above. We sum this up in the next definition.

Definition 3.4.4 (Left Residuals) For arbitrary elements p, q and r of a quantale, the left residual $q\backslash r$ is defined by the following Galois connection

$$p \le q\backslash r \ \Leftrightarrow_{df}\ q \cdot p \le r$$

In a quantale there also exists right residuals which are defined symmetrically. But since the multiplication in our quantale is commutative, both residuals coincide and so we stay with left residuals. Furthermore using left residuals will save in later proofs one step for changing $p \cdot q$ into $q \cdot p$ using commutativity of $\cdot$. The interpretation of $p -\!\!* q$ as $p\backslash q$ is also more intuitive. By using the pointwise extension we can again use this algebraic definition for $-\!\!*$ while calculating on our sets of states.

For residuals there already exist a lot of laws (e.g. [Möl05]). We list some of them without proofs to use them afterwards while proving given rules for $-\!\!*$ of [Rey08a]. For proofs of the following theorems the reader is referred to [Möl05].

Theorem 3.4.5 *For arbitrary p, q and r the inequation $(r\backslash q) \cdot (q\backslash p) \le r\backslash p$ holds.*

Theorem 3.4.6 *Left residuals are antitone in their first and isotone in their second argument, i.e.,*

$$p \le q \ \Rightarrow\ r\backslash p \le r\backslash q$$
$$p \le q \ \Rightarrow\ q\backslash r \le p\backslash r$$

Now we list the derived rules for $-\!\!*$ in [Rey08a], prove them algebraically and explain their meaning. To avoid confusions, we mainly list most of the separation logic laws algebraically encoded. The according laws in separation logic can be found by using the translation table in the appendix.

Lemma 3.4.7 *For arbitrary elements p and q the law $q \cdot (q \backslash p) \le p$ holds.*

Proof.

$$\begin{array}{ll} & q \cdot (q \backslash p) \leq p \\ \Leftrightarrow & \quad \{\!\![\text{ definition of } \backslash \,]\!\!\} \\ & q \backslash p \leq q \backslash p \\ \Leftrightarrow & \quad \{\!\![\leq \text{ is a partial order }]\!\!\} \\ & \mathsf{true} \end{array}$$

□

According to the translation table this law is encoded in separation logic as

$$\frac{}{q \ast (q \mathrel{-\!\!\ast} p) \ \Rightarrow\ p}$$

To explain the intuitive meaning of this inference rule, we assume a state (s, h) where h can be split into two disjoint parts h' and h''. Furthermore let (s, h') satisfy q and (s, h'') satisfy $q \mathrel{-\!\!\ast} p$, then (s, h) satisfies the assertion p which is a natural property of this operator.

Lemma 3.4.8 *For arbitrary elements p, q the law $q \cdot (q \backslash p) \leq (q \cdot \top) \sqcap p$ holds.*

Proof.

$$\begin{array}{ll} & q \cdot (q \backslash p) \\ = & \quad \{\!\![\text{ idempotence of } \sqcap \,]\!\!\} \\ & (q \cdot (q \backslash p)) \sqcap (q \cdot (q \backslash p)) \\ \leq & \quad \{\!\![\text{ Lemma 3.4.7, isotony of } \sqcap \,]\!\!\} \\ & (q \cdot (q \backslash p)) \sqcap p \\ \leq & \quad \{\!\![\ x \leq \top, \text{ isotony of } \cdot \text{ and } \sqcap \,]\!\!\} \\ & (q \cdot \top) \sqcap p \end{array}$$

□

Again this rule can be encoded as the following inference rule in separation logic

$$\frac{}{q \ast (q \mathrel{-\!\!\ast} p) \ \Rightarrow\ ((q \ast \mathsf{true}) \wedge p)}$$

This rule in contrast to the previous one, says that a state (s, h) that satisfies $q \ast (q \mathrel{-\!\!\ast} p)$ also satisfies p and somewhere in h there exists a part that still satisfies q. This behavior is modeled by the assertion $q \ast \mathsf{true}$. Assertions of this form are instances of *imprecise* assertions which we discuss later more detailed. The reader should understand up to now that they describe the extent of the heap imprecisely, i.e., it is not clear if the extent is empty or may contain some additional heap cells.

Lemma 3.4.9 *For arbitrary elements p, q the rule $p \leq q \backslash (q \cdot p)$ holds.*

Proof.

$$\begin{array}{ll} & p \leq q \backslash (q \cdot p) \\ \Leftrightarrow & \quad \{\!\![\text{ definition of } \backslash \,]\!\!\} \\ & q \cdot p \leq q \cdot p \\ \Leftrightarrow & \quad \{\!\![\leq \text{ is a partial order }]\!\!\} \\ & \mathsf{true} \end{array}$$

□

This rule is equal to the inference rule

$$\frac{}{p \;\Rightarrow\; (q \mathrel{-\!\!*} (q * p))}$$

To get an idea for this rule consider a state (s,h) that satisfies p. The heap h can also be seen as a residual heap of a state that satisfies $q * p$. That means by adding a disjoint heap h' of a state (s,h') that satisfies q to h, then the resulting state satisfies $q * p$.

Lemma 3.4.10 *For arbitrary assertions p, q, r the law $p \cdot r \leq p \cdot (q \backslash (q \cdot r))$ holds.*

Proof. This lemma follows immediately by setting $p = r$ in Lemma 3.4.9 and using isotony of $\cdot$. □

This rule says nothing new, since it just follows from isotony of $\cdot$. It can be encoded as

$$\frac{}{(p * r) \;\Rightarrow\; (p * (q \mathrel{-\!\!*} (q * r)))}$$

and is only listed here to have all inference rules of [Rey08a] algebraically proved in this report.

Lemma 3.4.11 *For arbitrary assertions p_0, p_1, q, r, s the law*

$$p_0 \leq q \backslash r \;\wedge\; p_1 \leq r \backslash s \;\Rightarrow\; p_1 \cdot p_0 \leq q \backslash s$$

holds.

Proof. Using the assumptions $p_0 \leq q \backslash r$ and $p_1 \leq r \backslash s$, we calculate

$$\begin{array}{ll}
 & p_1 \cdot p_0 \\
= & \quad \{\!\![\text{ commutativity of } \cdot \ ]\!\!\} \\
 & p_0 \cdot p_1 \\
\leq & \quad \{\!\![\text{ isotony of } \cdot\text{, premises }]\!\!\} \\
 & (q \backslash r) \cdot (r \backslash s) \\
\leq & \quad \{\!\![\text{ Lemma 3.4.5 }]\!\!\} \\
 & q \backslash s
\end{array}$$

□

Looking at the translation table we see that this law is equal to the inference rule

$$\frac{p_0 \;\Rightarrow\; (q \mathrel{-\!\!*} r) \quad p_1 \;\Rightarrow\; (r \mathrel{-\!\!*} s)}{(p_1 \,*\, p_0) \;\Rightarrow\; (q \mathrel{-\!\!*} s)}$$

This rule denotes a kind of transitivity law for the separating implication and can be viewed as the chaining law in usual predicate logic. In particular, consider two states (s,h), (s,h') with disjoint heaps that satisfies p_0 and p_1. Then $h \cup h'$ is a residuated heap of a heap h'', assuming (s,h'') satisfies s.

Lemma 3.4.12 *For arbitrary assertions p_0, p_1, q_0, q_1 the law*

$$p_0 \leq p_1 \;\wedge\; q_0 \leq q_1 \;\Rightarrow\; p_1 \,\backslash\, q_0 \leq p_0 \,\backslash\, q_1$$

holds.

Proof. This law follows immediately from Lemma 3.4.6 and the assumptions. □

Finally, we have found an algebraic characterization of the separating implication and are now able to algebraically calculate with this axiomatization. Residuals have already been analyzed earlier and a lot of results have been summarized in [Möl05]. Therefore we are able to show many of the laws of [Rey08a] by already proved algebraic properties of residuals. Comparing these algebraic proofs with the derivations given in [Rey08a], one can see that the algebraic proofs are, on the one hand, shorter and, on the other hand, more concise and easier to follow than the given derivation steps listed in [Rey08a].

In the next section we introduce a classification of assertions (due to [Rey08a]) and show how they can be modeled algebraically. In each class some special laws hold and for most of them we will be able to give short algebraic proofs.

3.5 Special Classes of Assertions

In [Rey08a] Reynolds categorized the assertions into special classes which we now want to introduce and discuss. The first class of assertions are *pure assertions*. These are assertions that are independent of the heap of a state. Therefore these assertions only give conditions on store variables. Examples of such assertions are e.g. $x = 1$, $\exists\, y.\ y < x$ or $k = x + y$. Next we give a definition for pure assertions and list some laws for them which we will prove using our algebraic characterization

Definition 3.5.1 (Pure assertions) An assertion p is called *pure* iff it is independent of every heap h. Or more precisely

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

In [Rey08a] Reynolds gives a few special laws which hold for this class of assertions:

$$\begin{array}{rc}
p \wedge q \Rightarrow p * q & \text{when } p \text{ or } q \text{ is pure} \\
p * q \Rightarrow p \wedge q & \text{when } p \text{ and } q \text{ are pure} \\
(p \wedge q) * r \Leftrightarrow (p * r) \wedge q & \text{when } q \text{ is pure} \\
(p \mathrel{-\!\!*} q) \Rightarrow (p \rightarrow q) & \text{when } p \text{ is pure} \\
(p \rightarrow q) \Rightarrow (p \mathrel{-\!\!*} q) & \text{when } p \text{ and } q \text{ are pure}
\end{array}$$

As we can see from the given laws, under certain conditions, the distinction between the separating conjunction and the usual Boolean conjunction collapses. The same situation can be seen for the separating implication and the Boolean implication[3] →. The central idea behind these assertions is that they are not heap dependent. That means the heap is only carried with in every state without making any changes to it.

The third of the laws above is an important and often used one in proofs of programs Its meaning is: if q is a pure assertion, we can modify the stores of p or r before combining the disjoint heaps satisfying p or q. This behavior will be explained when discussing the algebraic characterization for these assertions.

In [Rey08a] Reynolds only defined pure assertions as in Definition 3.5.1 and gives the five laws above for pure assertions without proving them. We introduce in the following an algebraic characterisation for this class of assertions and show how the five rules given above can be simply proved. One advantage we get is that the proof of the laws above can

[3] → denotes the implication in the logic itself while ⇒ is the implication at the meta-logical level

be fully automated using Prover9. Furthermore we also found from this characterization some useful features for pure assertions.

We now present an algebraic characterization for pure assertions and the idea behind it. Thereafter we list some features of pure assertions and prove the given laws above in separation logic algebraically.

Definition 3.5.2 (Algebraic characterization of pure assertions) An element p of a quantale is called *pure* iff for arbitrary elements q and r

$$p \cdot \top \leq p \qquad \text{(upward closure)}$$

$$(q \cdot r) \sqcap p \leq (q \sqcap p) \cdot (r \sqcap p) \qquad \text{(meet-distributivity)}$$

Remark 3.5.3 Of course, both axioms can be strenghtend to equations, since for the upward closure axiom $p \leq p \cdot \top$ always holds by isotony. For the meet-distributivity axiom we calculate

$$\begin{array}{ll} & (q \sqcap p) \cdot (r \sqcap p) \\ \leq & \quad \{\!\![\text{ isotony of } \sqcap, \cdot \ ]\!\!\} \\ & p \cdot \top \\ \leq & \quad \{\!\![\text{ upward closure }]\!\!\} \\ & p \end{array}$$

and

$$\begin{array}{ll} & (q \sqcap p) \cdot (r \sqcap p) \\ \leq & \quad \{\!\![\text{ isotony of } \sqcap, \cdot \ ]\!\!\} \\ & q \cdot r \end{array}$$

Thus, by definition the claim $(p \sqcap q) \cdot (p \sqcap r) \leq p \sqcap (q \cdot r)$ follows. We use $=$ where needed in the later proofs. □

To get an idea for the upward closure axiom, consider an arbitrary state (s, h) in p. Then every state (s, h') with $h \subseteq h'$ is an element of $p \cdot \top$. That means the set $p \cdot \top$ consists of all states that we have in p including those where the heap is extended with arbitrary disjoint parts and therefore p is upward closed.

But since pure assertions holds for every heap, we have only modeled one direction for pureness by this axiom. We will later see that by the meet-distributivity axiom we also capture the downward closureness.

Note that the upward closure axiom implies

$$p \cdot q \leq p \qquad (*)$$

for all q.

The meet-distributivity axiom characterizes the interplay between the separating conjunction and the Boolean conjunction with pure assertions. In detail, it says algebraically that some states of the sets q and r can be discarded before joining disjoint heaps of the states. This is clear since, by closure of p, e.g. in $q \sqcap p$ only such states will be discarded from q where the store does not equal the store in some state in p. Remember the heap will not be changed by this intersection since in p every possible heap combination is included for arbitrary states.

To see that this law in fact holds, we give a correctness proof in separation logic. We assume that p is a pure assertion.

$$\begin{array}{ll}
 & s, h \models p \wedge (q * r) \\
\Leftrightarrow & \quad \{\![\ \text{definition of } \wedge\ ]\!\} \\
 & s, h \models p \quad \text{and} \quad s, h \models q * r \\
\Leftrightarrow & \quad \{\![\ \text{definition of } *\ ]\!\} \\
 & s, h \models p \quad \text{and} \quad (\exists\, h_0 : h_0 \subseteq h \quad \text{and} \quad s, h_0 \models q \quad \text{and} \quad s, h - h_0 \models r) \\
\Leftrightarrow & \quad \{\![\ \text{logic}\ ]\!\} \\
 & \exists\, h_0 : h_0 \subseteq h \quad \text{and} \quad s, h \models p \quad \text{and} \quad s, h_0 \models q \quad \text{and} \quad s, h - h_0 \models r \\
\Leftrightarrow & \quad \{\![\ p \text{ is pure}\ ]\!\} \\
 & \exists\, h_0 : h_0 \subseteq h \quad \text{and} \quad (s, h_0 \models p \quad \text{and} \quad s, h_0 \models q) \quad \text{and} \\
 & \qquad\qquad\qquad (s, h - h_0 \models p \quad \text{and} \quad s, h - h_0 \models r) \\
\Leftrightarrow & \quad \{\![\ \text{definition of } \wedge\ ]\!\} \\
 & \exists\, h_0 : h_0 \subseteq h \quad \text{and} \quad s, h_0 \models p \wedge q \quad \text{and} \quad s, h - h_0 \models p \wedge r \\
\Leftrightarrow & \quad \{\![\ \text{definition of } *\ ]\!\} \\
 & s, h \models (p \wedge q) * (p \wedge r)
\end{array}$$

Next we give a consequence of these two axioms to see that we have exactly modeled the behavior in Definition 3.5.1.

Corollary 3.5.4 *For an arbitrary pure element p,*

$$p = (p \sqcap 1) \cdot \top \qquad \textit{(downward closure)}$$

Proof.
($\leq$):

$$\begin{array}{ll}
 & p \\
= & \quad \{\![\ \top \text{ neutral for } \sqcap,\ 1 \text{ neutral for } \cdot\ ]\!\} \\
 & p \sqcap (1 \cdot \top) \\
= & \quad \{\![\ \text{meet-distributivity}\ ]\!\} \\
 & (p \sqcap 1) \cdot (p \sqcap \top) \\
\leq & \quad \{\![\ \text{isotony w.r.t. } \cdot\ ]\!\} \\
 & (p \sqcap 1) \cdot \top
\end{array}$$

($\geq$):

$$\begin{array}{ll}
 & (p \sqcap 1) \cdot \top \\
\leq & \quad \{\![\ \text{isotony w.r.t. } \cdot, \sqcap\ ]\!\} \\
 & p \cdot \top \\
\leq & \quad \{\![\ p \text{ is upward closed}\ ]\!\} \\
 & p
\end{array}$$

□

The element 1 is the neutral element for $\cdot$ in our quantale. Therefore it denotes the set $\llbracket\, \mathsf{emp}\, \rrbracket$. This corollary says that we can empty the heap of all states in p and then build the heap up with arbitrary cells. This means that p holds for states with an arbitrary heap $h \in \mathit{Heaps}$ and therefore is independent of the heap. And since the two axioms in Definition 3.5.2 follow from Definition 3.5.1, we have exactly modeled pure assertions algebraically.

Before we give algebraic proofs for the above listed consequences of pure assertions, we give a major result which immediately follow from Definition 3.5.2.

Corollary 3.5.5 *For a pure element p and an arbitrary element q the following law holds:*

$$p \cdot q = p \sqcap (q \cdot \top)$$

Proof. We split the first law into two inequations, proving each separately.
$(\geq)$:

$$\begin{array}{ll}
 & p \sqcap (q \cdot \top) \\
\leq & \quad \{\![\ p \text{ is pure }]\!\} \\
 & (p \sqcap q) \cdot (p \sqcap \top) \\
= & \quad \{\![\ \top \text{ neutral w.r.t. } \sqcap\ ]\!\} \\
 & (p \sqcap q) \cdot p \\
\leq & \quad \{\![\ \text{isotony of } \cdot,\ p \sqcap r \leq p\ ]\!\} \\
 & q \cdot p \\
= & \quad \{\![\ \text{commutativity of } \cdot\ ]\!\} \\
 & p \cdot q
\end{array}$$

$(\leq)$: From $(*)$ we know $p \cdot q \leq p$. Furthermore $p \cdot q = q \cdot p \leq q \cdot \top$ and by definition we get $p \cdot q \leq p \sqcap (q \cdot \top)$. □

This law says that since p is a pure element in our quantale it holds for every heap. Therefore we can first calculate $q \cdot \top$ which have on the heap component of our states the same effect as $q \cdot p$ and afterwards discard all states that do not occur in p, i.e., where the store component still differs from every store of the states in p.

In the following we use this result to prove the five laws given at the beginning of this section for pure assertions. Furthermore we give some consequences which follow by the algebraic axiomatization. Since we give algebraic proofs, we encode all laws algebraically. They can be translated in separation logic using the given translation table in the appendix.

Lemma 3.5.6 *The law $(p \sqcap q) \cdot r = (p \cdot r) \sqcap q$ holds if q is pure.*

Proof. We will split the proof of $(p \sqcap q) \cdot r = (p \cdot r) \sqcap q$ into two parts, proving each inequation separately.
$(\leq)$:

$$\begin{array}{ll}
 & (p \sqcap q) \cdot r \\
\leq & \quad \{\![\ \text{subdistributivity of } \cdot \text{ w.r.t } \sqcap\ ]\!\} \\
 & (p \cdot r) \sqcap (q \cdot r) \\
\leq & \quad \{\![\ \text{by } (*)\text{, isotony of } \sqcap\ ]\!\} \\
 & (p \cdot r) \sqcap q
\end{array}$$

$(\geq)$:

$$\begin{array}{ll}
 & (p \cdot r) \sqcap q \\
= & \quad \{\![\ q \text{ is meet-distributive }]\!\} \\
 & (p \sqcap q) \cdot (r \sqcap q) \\
\leq & \quad \{\![\ \text{isotony of } \cdot, \sqcap\ ]\!\} \\
 & (p \sqcap q) \cdot r
\end{array}$$

□

Lemma 3.5.7 *The law $p \sqcap q \leq p \cdot q$ holds if p or q is pure.*

Proof. We assume q is a pure element. The proof for p is symmetric since $\cdot$ and $\sqcap$ are commutative.

$$\begin{aligned}
& p \sqcap q \\
\leq \quad & \{\![\ \text{isotony of } \cdot, \sqcap, 1 \leq \top\ ]\!\} \\
& (p \cdot \top) \sqcap q \\
= \quad & \{\![\ \text{Corollary 3.5.5}\ ]\!\} \\
& p \cdot q
\end{aligned}$$

□

Lemma 3.5.8 *The law $p \cdot q \leq p \sqcap q$ holds if p and q are pure.*

Proof. This law follows immediately using Corollary 3.5.5:

$$p \cdot q = p \sqcap (q \cdot \top) \leq p \sqcap q$$

□

Now we immediately conclude by using the previous two results that $\cdot$ and $\sqcap$ coincides when we only calculate on pure elements.

Corollary 3.5.9 *When p and q are pure then $p \cdot q = p \sqcap q$.*

Corollary 3.5.10 *If p is pure then $p \cdot p = p$ holds.*

Proof. Using Corollary 3.5.9 we calculate $p \cdot p = p \sqcap p = p$. □

This law says that $\cdot$ becomes an idempotent operator when calculating on pure elements. This is a natural behaviour since we know that on pure elements $\cdot$ and $\sqcap$ coincide and $\sqcap$ is clearly an idempotent operator.

Next we give a closure property for pure assertions.

Lemma 3.5.11 *If p and q are pure elements then*

$$p + q \qquad\qquad p \sqcap q \qquad\qquad p \cdot q$$

are also pure elements. Hence pure assertions themselves form a semiring.

Proof. We start by proving that $p + q$ is upward closed and calculate

$$(p + q) \cdot \top = p \cdot \top + q \cdot \top \leq p + q$$

For meet-distributivity we calculate for arbitrary elements r and s

$$\begin{aligned}
& (p + q) \sqcap (r \cdot s) \\
= \quad & \{\![\ \text{distributivity}\ ]\!\} \\
& (p \sqcap (r \cdot s)) + (q \sqcap (r \cdot s)) \\
\leq \quad & \{\![\ p,\ q \text{ are meet-distributive}\ ]\!\} \\
& (p \sqcap r) \cdot (p \sqcap s) + (q \sqcap r) \cdot (q \sqcap s) \\
\leq \quad & \{\![\ \text{isotony}\ ]\!\} \\
& ((p + q) \sqcap r) \cdot ((p + q) \sqcap s) + ((p + q) \sqcap r) \cdot ((p + q) \sqcap s) \\
= \quad & \{\![\ \text{idempotence of } +\ ]\!\} \\
& ((p + q) \sqcap r) \cdot ((p + q) \sqcap s)
\end{aligned}$$

That $p \sqcap q$ is upward closed, can be shown by

$$(p \sqcap q) \cdot \top \leq p \cdot \top \sqcap q \cdot \top \leq p \sqcap q$$

Meet-distributivity can be seen from

$$\begin{array}{ll}
 & (p \sqcap q) \sqcap (r \cdot s) \\
= & \quad \{\![\ \text{associativity}\ ]\!\} \\
 & p \sqcap (q \sqcap (r \cdot s)) \\
\leq & \quad \{\![\ q \text{ is meet-distributive}\ ]\!\} \\
 & p \sqcap ((q \sqcap r) \cdot (q \sqcap s)) \\
\leq & \quad \{\![\ p \text{ is meet-distributive, associativity}\ ]\!\} \\
 & ((p \sqcap q) \sqcap r) \cdot ((p \sqcap q) \sqcap s))
\end{array}$$

From this and $p + q = p \sqcap q$ by Corollary 3.5.9 it follows that $p \cdot q$ is also pure. □

In the next two laws we use instead of the definition of → by negation the following definition

$$\forall\, r \ : \ r \leq p \mathrel{-\!\triangleright} q \ \Leftrightarrow_{df} \ r \sqcap p \leq q$$

where $\mathrel{-\!\triangleright}$ denotes the algebraic version of →. Well-definedness is assured since $p \mathrel{-\!\triangleright} q = \overline{p} + q$ and the equivalence above follows by shunting. With this definition we are also able to prove the laws without the assumption that we are in a Boolean quantale.

Lemma 3.5.12 *The law $p \setminus q \leq p \mathrel{-\!\triangleright} q$ holds if p is pure.*

Proof. We prove $p \setminus q \leq p \mathrel{-\!\triangleright} q$ using the proof principle of indirect inequality.

$$\begin{array}{ll}
 & p \setminus q \leq p \mathrel{-\!\triangleright} q \\
\Leftrightarrow & \quad \{\![\ \text{indirect inequality}\ ]\!\} \\
 & \forall\, r : r \leq p \setminus q \ \Rightarrow \ r \leq p \mathrel{-\!\triangleright} q \\
\Leftrightarrow & \quad \{\![\ \text{definition of } \setminus\,,\ \mathrel{-\!\triangleright}\ ]\!\} \\
 & \forall\, r : p \cdot r \leq q \ \Rightarrow \ r \sqcap p \leq q \\
\Leftarrow & \quad \{\![\ p \text{ is pure, Lemma 3.5.7}\ ]\!\} \\
 & \forall\, r : r \sqcap p \leq r \cdot p \ \wedge \ r \cdot p = p \cdot r \leq q \ \Rightarrow \ r \sqcap p \leq q
\end{array}$$

□

Lemma 3.5.13 *The law $p \mathrel{-\!\triangleright} q \leq p \setminus q$ holds if p and q are pure.*

Proof.

$$\begin{array}{ll}
 & p \mathrel{-\!\triangleright} q \leq p \setminus q \\
\Leftrightarrow & \quad \{\![\ \text{indirect inequality}\ ]\!\} \\
 & \forall\, r : r \leq p \mathrel{-\!\triangleright} q \ \Rightarrow \ r \leq p \setminus q \\
\Leftrightarrow & \quad \{\![\ \text{definition of } \setminus\,,\ \mathrel{-\!\triangleright}\ ]\!\} \\
 & \forall\, r : r \sqcap p \leq q \ \Rightarrow \ p \cdot r \leq q
\end{array}$$

Now we assume $p \sqcap r = r \sqcap p \leq q$. By Corollary 3.5.5, meet-distributivity of p, neutrality of $\top$ w.r.t. $\sqcap$, the assumption, isotony of $\cdot$, upward closure of q,

$$p \cdot r = p \sqcap (r \cdot \top) = (p \sqcap r) \cdot (p \sqcap \top) = (p \sqcap r) \cdot p \leq q \cdot p \leq q \cdot \top \leq q$$

□

Now, using the assumption that we are in a Boolean quantale we sum up some further results in the following.

Corollary 3.5.14 *If p is a pure element then $\overline{p}$ is also a pure element. Hence in a Boolean quantale pure assertions themselves form a Boolean semiring.*

Proof. Since p is pure we can instantiate the meet-distributivity axiom for pure assertions using $q = \overline{p}$ and $r = \top$, i.e.,

$$(\overline{p} \cdot \top) \sqcap p \leq (\overline{p} \sqcap p) \cdot (\top \sqcap p) = 0.$$

From this we obtain by shunting $\overline{p} \cdot \top \leq \overline{p}$. Next we calculate for arbitrary elements q and r

$$\begin{array}{ll}
 & (q \cdot r) \sqcap \overline{p} \\
= & \quad \{\!\![\ \top \text{ is neutral w.r.t. } \sqcap\ ]\!\!\} \\
 & ((q \sqcap \top) \cdot (r \sqcap \top)) \sqcap \overline{p} \\
= & \quad \{\!\![\ \top = p + \overline{p}\ ]\!\!\} \\
 & ((q \sqcap (p + \overline{p})) \cdot (r \sqcap (p + \overline{p}))) \sqcap \overline{p} \\
= & \quad \{\!\![\ \text{distributivity of } \sqcap \text{ over } +\ ]\!\!\} \\
 & ((q \sqcap p + q \sqcap \overline{p}) \cdot (r \sqcap p + r \sqcap \overline{p})) \sqcap \overline{p} \\
= & \quad \{\!\![\ \text{distributivity of } \cdot \text{ over } +\ ]\!\!\} \\
 & ((q \sqcap p) \cdot (r \sqcap p) + (q \sqcap \overline{p}) \cdot (r \sqcap p) + (q \sqcap p) \cdot (r \sqcap \overline{p}) + (q \sqcap \overline{p}) \cdot (r \sqcap \overline{p})) \sqcap \overline{p} \\
= & \quad \{\!\![\ p \text{ is pure, Lemma 3.5.6}\ ]\!\!\} \\
 & ((q \cdot r) \sqcap p + ((q \sqcap \overline{p}) \cdot r) \sqcap p + ((r \sqcap \overline{p}) \cdot q) \sqcap p + (q \sqcap \overline{p}) \cdot (r \sqcap \overline{p})) \sqcap \overline{p} \\
= & \quad \{\!\![\ \text{distributivity of } \sqcap \text{ over } +,\ p \sqcap \overline{p} = 0\ ]\!\!\} \\
 & ((q \sqcap \overline{p}) \cdot (r \sqcap \overline{p})) \sqcap \overline{p} \\
\leq & \quad \{\!\![\ \text{isotony of } \sqcap\ ]\!\!\} \\
 & (q \sqcap \overline{p}) \cdot (r \sqcap \overline{p})
\end{array}$$

And together we get $\overline{p}$ is also pure. □

Corollary 3.5.15 *If p is pure then $p \cdot \overline{p} = 0$.*

Proof. By Lemma 3.5.11 we know $\overline{p}$ is pure and using Corollary 3.5.9 we calculate $p \cdot \overline{p} = p \sqcap \overline{p} = 0$. □

Finally, all laws given in [Rey08a] can be easily proved algebraically from two axioms. Notice that in [Rey08a] no proofs for these laws in separation logic are given. Using the algebraic approach one gets short and easy proofs. They are, compared to separation logic proofs given in this report, much easier to understand since they are precise and emphasize the basic ideas of the proofs.

In the following we introduce some other classes of assertions. It is to mention here that we only define and explain them very shortly. An algebraic treatment for them is subject of future work.

The next special class we discuss is the class of *precise* assertions [Rey08a]. Assertions of this class describe the domain of a heap exactly, in contrast to intuitionistic assertions which we discuss afterwards. This means for every heap that we know from these assertions exactly which heap cells are allocated. Examples are

$$x \mapsto 10 \qquad \text{or} \quad \exists\, v.\ x \mapsto v \qquad \text{or} \quad \mathsf{emp}$$

while imprecise assertions are, as already mentioned, e.g.

$$\mathsf{true} \qquad \text{or} \quad \exists\, x.\ x \mapsto v \qquad \text{or} \quad \mathsf{emp}\ \lor\ x \mapsto 10$$

Notice the distinction between $\exists\, v.\; x \mapsto v$ and $\exists\, x.\; x \mapsto v$. In the first case the quantified variable is a placeholder for the value of a heap cell and in the second case it denotes a placeholder for the address of a heap cell. Hence, the second assertion is an instance of an imprecise assertion since the domain of the heap is not described exactly. Precise assertions are defined as follows.

Definition 3.5.16 (Precise assertions) Given a heap, if precise assertion holds for any subheap, then it holds for a unique subheap. In detail, an assertion p is called *precise* iff for all heaps h and stores s, there exists a unique subheap $h' \subseteq h$ such that $s, h' \models p$ holds.

Next we come back to the subdistributivity law $(p * r) \wedge (q * r) \Rightarrow (p \wedge q) * r$ which does not hold for arbitrary assertions p, q and r. The problem in our counterexamples we encountered before can be avoided if we assume that the assertion r is precise [Rey08a]. To see this, let (s, h) be a state assuming $s, h \models (p * r) \wedge (q * r)$ holds. Then by definition there exists a heap $h_0 \subseteq h$ such that $s, h - h_0 \models p$ and $s, h_0 \models r$ holds. Equally there exists a heap h_1 such that $s, h - h_1 \models q$ and $s, h_1 \models r$ holds. Now, since r is precise it follows $h_0 = h_1$ and therefore $s, h - h_0 \models p \wedge q$. Finally by definition of $*$ the claims follows.

Thus, for these assertions the distributive law in fact holds. Almost the same argumentation can be used for the $\forall$-quantified law mentioned above.

The next lemma gives a simple closure characterization w.r.t. to the separation conjunction and the usual conjunction for precise assertions. We do not go more into detail here as already mentioned.

Lemma 3.5.17 *The following assertions are precise under the given conditions.*

$$\begin{array}{cc} p * q & \textit{when } p \textit{ and } q \textit{ are precise} \\ p \wedge q & \textit{when } p \textit{ or } q \textit{ is precise} \\ p & \textit{when } p \Rightarrow q \textit{ holds and } q \textit{ is precise} \end{array}$$

We conclude this chapter by defining *intuitionistic assertions.* These assertions are instances of imprecise assertions, i.e., they do not describe the domain of a heap exactly. Hence, using these assertions one might do not know if the heap does not contain additional anonymous cells. We define them first and then give an example.

Definition 3.5.18 (Intuitionistic assertions) An assertion p is called intuitionistic iff for all stores s and heaps h, h':

$$(h \subseteq h' \text{ and } s, h \models p) \text{ implies } s, h' \models p$$

This means for a heap that satisfies an intuitionistic assertion p that it can be extended with arbitrary cells and still satisfies p. In particular, these assertions are instances of imprecise assertions since the domain of a heap is not determined exactly. As a simple example consider the assertion

$$(x \mapsto 1) * \mathsf{true}$$

It describes that the heap of a state that satisfies this assertion can be split into a part in which we only got one cell containing the value 1 and into another disjoint part for which true is satisfiable. That means we can make any assumption for the rest of the heap since it may contain arbitrary cells or is empty. This brings us to the idea of characterizing this class of assertions as follows.

Definition 3.5.19 (Intuitionistic elements) Consider an element i of our algebraic setting. Then it is called *intuitionistic* iff

$$i \cdot \top \quad \leq \quad i.$$

Of course, an assertion is intutionistic iff it is upward closed. All of the laws for pure assertions we have given in the previous section are also applicable for this class of assertions if they can be proved without using the meet-distributivity axiom for pure assertions. In particular, assuming i is intuitionistic,

- $(i \sqcap 1) \cdot \top \leq i$
- $i \cdot q \leq i \sqcap (q \cdot \top)$
- $(p \sqcap i) \cdot q \leq (p \cdot q) \sqcap i$
- $i \cdot i \leq i$
- $i + i'$ and $i \sqcap i'$ are intuitionistic if i, i' are intuitionistic

holds. We give a proof for one law of [Rey08a] which we have not proved for pure assertions by using only the upward closure axiom.

Lemma 3.5.20 *Let i and i' be arbitrary intuitionistic elements then*

$$i \cdot i' \leq i \sqcap i'$$

is valid.

Proof.

$$\begin{array}{ll}
 & i \cdot i' \\
= & \quad \{\![\ \sqcap \text{ is idempotent }]\!\} \\
 & (i \cdot i') \sqcap (i \cdot i') \\
\leq & \quad \{\![\ \text{isotony of } \cdot \text{ and } \sqcap,\ \top \text{ greatest element }]\!\} \\
 & (i \cdot \top) \sqcap (\top \cdot i') \\
\leq & \quad \{\![\ i \text{ and } i' \text{ are intuitionistic, isotony }]\!\} \\
 & i \sqcap i'
\end{array}$$

□

All the laws given in [Rey08a] are simple consequences of the axiomatization and knowledge given so far. We will not list and prove them in the following since they should be straightforward.

In the next chapter we want to have a closer look at the command language of separation logic and try to handle it algebraically by using a relational approach.

Chapter 4

Commands

In this chapter we introduce the command constructs of separation logic. We start by explaining the semantics of the commands. Next we present an algebraic approach and give a relational view for the commands. Furthermore we explain how the assertion quantale, presented before, can be extended and used in this relational view. After that we go into more detail and give some inference rules for the commands.

We begin by syntactically defining the constructs for the command language in separation logic.

Definition 4.0.21 (Syntax for commands)

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

As in Hoare logic we have a lot of well-known constructs like variable assignments and definitions in local scopes. For example the if-then-else, while-do constructs and command composition are carried over.

The cons command is an allocation command: if $e_1 \ldots e_n$ are n arbitrary expressions then $\mathsf{cons}\,(e_1, \ldots, e_n)$ allocates n arbitrary consecutive cells with e_1 as the content of the first cell, e_2 as the content of the second cell, and so forth. The cells have to be unused and lie somewhere on the heap, since this allocation process is defined to be non-deterministic. The command only returns the address of the first cell while the rest of the cells can be indirectly addressed using the start address.

The dereferencing command $[e]$ assumes that e is an $\langle\text{exp}\rangle$-expression and the evaluated expression corresponds to an allocated address on the heap, i.e., $\langle e\rangle_{\text{exp}}(s) \in \mathsf{dom}(h)$ for the given heap h. It has the same meaning as e.g. the expression `*e` in the programming language `C`. In particular, this means for commands of the form $\langle\text{var}\rangle := [\langle\text{exp}\rangle]$ that after execution, the variable on the left hand side of := saves the contents of a dereferenced heap cell. Conversely an execution of a command $[\langle \text{exp} \rangle] := \langle \text{exp} \rangle$ assigns the value of the expression on the right hand side to the cell being the value of the left hand side.

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

In the following we define a relational view for the commands to be able to treat them algebraically.

4.1 Relational Semantics

As already mentioned, separation logic is an extension of Hoare logic with special operators to reason about separate heap regions. For Hoare logic there already exists a relational approach which we reuse for a relational treatment of separation logic. First we summarize the main concepts for the relational approach of Hoare logic. Next we extend the semantics so that we are able to define the new heap-related commands of separation logic in this relational view.

In Hoare logic triples $\{p\}\, c\, \{q\}$, where p and q are assertions and c is a command, are named *specifications* ([Rey08a, Mor98]). As usual, p denotes the precondition or initial state while q denotes the postcondition or final state after the execution of c. The triples we are considering in this report are partial correctness triples, i.e., we do not consider the termination of a program execution. This is sufficient, since for the examples in this report termination of the programs is assumed.

The relational approach for Hoare logic considers assertions p, q and commands c to be relations. The relation c denotes transitions from states of the precondition p to states of the postcondition q. We denote the relation for the command skip, which denotes the command that does nothing, by $[\![\, \mathsf{skip} \,]\!]$. Furthermore we know that the states in Hoare logic only consist of stores. The algebraic structure for Hoare logic is the quantale $(\mathcal{P}(\mathit{REL}(\mathit{Stores})), \cup, \emptyset, ;, [\![\, \mathsf{skip} \,]\!])$ where $\mathit{REL}(S) =_{\mathit{df}} S \times S$ for arbitrary set S and ; denotes the relational composition.

The conditions p and q are represented as special relations called *test elements*.

Definition 4.1.1 (Test elements) An element $p \leq 1$ in a semiring $(S, +, 0, \cdot, 1)$ is called a *test element* iff it has a complement $\neg p$ which satisfies

$$p + \neg p = 1 \qquad p \cdot \neg p = 0 = \neg p \cdot p$$

Hence the test elements in the quantale form the assertions in Hoare logic and the triples above can be algebraically encoded in the quantale by

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

The quantale can also be extended to an iteration algebra using the Kleene star operator ([Koz94, Koz97]). It is well-known that we are able to algebraically encode while loops using this operator.

Definition 4.1.2 (Kleene algebra) A *Kleene algebra* is an idempotent semiring S extended by an operation $^* : S \to S$ that satisfies the *star unfold* axioms

$$1 + x \cdot x^* \leq x^* \qquad 1 + x^* \cdot x \leq x^*$$

and the *star induction* axioms

$$y + x \cdot z \leq z \;\Rightarrow\; x \cdot y^* \leq z \qquad y + z \cdot x \leq z \;\Rightarrow\; y^* \cdot x \leq z$$

With this definition the structure $(\mathcal{P}(REL(Stores)), \cup, \emptyset, ;, [\![\mathsf{skip}]\!], {}^*)$ forms a Kleene algebra. According to [MS06, Koz00], we can algebraically encode the partial correctness inference rule

$$\frac{\{p \wedge i\}\, c\, \{i\}}{\{i\}\ \mathsf{while}\ p\ \mathsf{do}\ c\, \{\neg p \wedge i\}}$$

for the while loop where i denotes a loop invariant, e.g. by

$$p \cdot i \cdot c \leq c \cdot i \ \Rightarrow\ i \cdot (p \cdot c)^* \cdot \neg p \leq (p \cdot c)^* \cdot \neg p \cdot i.$$

In the concrete case of the relation quantale $\cdot$ is relational composition ; .

Using these fundamentals we first extend the semantics so that we are also able to work with heaps. Therefore we interpret the states as tuples consisting of a store and a heap component and extend the relations now to pairs of these states. By defining the set $[\![\mathsf{skip}]\!] =_{df} \{((s,h),(s,h)) : (s,h) \in States\}$, which is the neutral element for ; , we get the quantale $(\mathcal{P}(REL(States)), \cup, \emptyset, ;, [\![\mathsf{skip}]\!])$ as our basic algebraic structure for the commands. We call this structure the *command quantale*. Finally we define a relational view for new commands. For all following commands we assume that $FV(e), FV(e') \subseteq s$ always holds for the stores s involved in the definitions.

For a relational view of mutation commands $[\![[e] := e]\!]$ the transition from an arbitrary state (s,h) can be defined using the update operator $\mid$:

$$[\![[e] := e']\!] =_{df}$$
$$\{((s,h),(s,(a,v) \mid h)) : (s,h) \in States, a = \langle e \rangle_{\exp}(s) \in \mathsf{dom}(h), v = \langle e' \rangle_{\exp}(s)\}.$$

This definition coincides exactly with the described semantics for mutation commands at the beginning of this chapter. The condition $\langle e \rangle_{\exp}(s) \in \mathsf{dom}(h)$ is needed to ensure that the dereferenced heap cell is already allocated on the heap, since otherwise we would immediately get a fault in the program execution.

Next we define a relational semantics for the $\mathsf{dispose}$ command. As already mentioned, it is used to deallocate heap cells.

$$[\![\mathsf{dispose}\, e]\!] =_{df} \{((s,h),(s,h - \{(a,h(a))\})) : (s,h) \in States, a = \langle e \rangle_{\exp}(s) \in \mathsf{dom}(h)\}.$$

Of course, after the execution of $\mathsf{dispose}$, the address $\langle e \rangle_{\exp}(s)$ is not allocated anymore in the resulting state. Again if $\langle e \rangle_{\exp}(s) \notin \mathsf{dom}(h)$, we would get a fault when executing $\mathsf{dispose}$.

Conversely to the command $\mathsf{dispose}$, we now define a relational semantics for the command cons which is used to allocate cells on the heap.

$$[\![v := \mathsf{cons}\,(e_0, ..., e_{n-1})]\!] =_{df}$$
$$\{((s,h),((v,a) \mid s,\ \{(a,v_0)...(a+n-1,v_{n-1})\} \mid h)) : (s,h) \in States\},$$

where $a,..., a+n-1 \notin \mathsf{dom}(h)$, $\{v\} \cup FV(e_0, \ldots, e_{n-1}) \subseteq \mathsf{dom}(s)$ and $v_0 = \langle e_0 \rangle_{\exp}(s)$,..., $v_{n-1} = \langle e_{n-1} \rangle_{\exp}(s)$. For an allocation of n consequtive cells, the addresses $a, \ldots, a+n-1$ have to be unallocated.

The last heap-dependent command we add to our relational view are lookup commands, i.e., commands where values of heap cells are read and assigned to a store variable. They are defined as follows:

$$[\![v := [e]]\!] =_{df} \{((s,h),((v,h(a)) \mid s,h)) : (s,h) \in States, a = \langle e \rangle_{\exp}(s) \in \mathsf{dom}(h)\}.$$

By these commands only the store gets updated and the heap remains unaltered.

Usual variable assignments, as they are known from Hoare logic, are carried over and semantically extended to work on states with an additional heap component. Therefore we assume that v is a variable and e an $\langle\text{exp}\rangle$-expression. Then variable assignment is defined as follows:

$$[\![v := e]\!] =_{df} \{((s,h),((v,a)\,|\,s,h)) : (s,h) \in States, a = \langle e\rangle_{\text{exp}}(s)\}.$$

With these definitions we are able to work with the new commands of separation logic. But the algebraic structure still lacks expressiveness for the separating conjunction. In the previous chapter we presented an algebraic approach to assertions which we will embed in the next section into the relational view. This will enrich test elements with the possiblity to work with the heap operators.

4.2 Embedding the Assertion Quantale

As known from the previous section, the test elements of our relational view coincide with the assertions in Hoare logic. In Chapter 3 we presented an algebraic approach for working with assertions in separation logic. Now we explain how the assertion quantale of Chapter 3 can be extended to express separation on parts on the heap, so that it can be integrated into the relational view. Furthermore we continue to use all the laws for assertions we have proved before.

The test elements in the defined relation structure are subidentities, i.e. $p \in test(S) \Leftrightarrow p \leq 1$, where 1 denotes the set $[\![\,\mathsf{skip}\,]\!]$. This set consists of pairs of the form $((s,h),(s,h))$, i.e., the states in each component are equal. Therefore we have to extend the elements of the assertion quantale to sets of such pairs so that we can work with them using the relational view. By extending the elements to relations, defining $\hat{P} =_{df} \{(t,t) : t \in P\}$ for $P \in \mathcal{P}(States)$, we have to extend the corresponding operators of the quantale. The union and empty set can be adopted from the command quantale. Only the disjoint union on heaps which denotes the multiplication in the assertion quantale needs to be redefined. We write $\uplus$ for this operator and define $\uplus : test(REL(States)) \times test(REL(States)) \to test(REL(States))$ by

$$\hat{P} \uplus \hat{Q} =_{df} \widehat{P \cup\!\!\!\!\cdot\;\, Q}\ .$$

with $P, Q \in \mathcal{P}(States)$. This operator makes sense only when working on test elements. Hence we define the result to be $\emptyset$ when the relation on the left hand side or on the right hand side is not a test element, i.e., both states in each relation must be equal. Furthermore, as in the definition of $\cup\!\!\!\!\cdot\;$ the stores also have to be equal. Moreover, the disjoint union on heaps is only defined when the domains of the considered heaps are disjoint. For all other cases the operator returns $\emptyset$ which is the result of the error case. Thus, this operator denotes a natural extension of $\cup\!\!\!\!\cdot\;$ to sets of tuples and has the same semantics as $\cup\!\!\!\!\cdot\;$ in the assertion quantale.

The neutral element for this operator can also easily be integrated into the relational view by defining

$$[\![\,\mathsf{emp}\,]\!] =_{df} \{((s,\emptyset),(s,\emptyset)) : s \in Stores\}.$$

Since test elements need to have complements, i.e., the set of test elements is Boolean, we have to assume a Boolean assertion quantale. In Chapter 3 we have written $\bar{\cdot}$ to denote the complement of an element, in the Boolean quantale. To avoid confusions in the following, we will use the notation from the definition of test elements and denote the complement by $\neg$. We sum up the results in the following lemma.

Lemma 4.2.1 *The structure* $(\mathsf{test}(()REL(States)), \cup, \emptyset, \uplus, [\![\,\mathsf{emp}\,]\!], \neg)$ *forms a Boolean quantale.*

Since all test elements p satisfy $p \leq 1$ in our relational view, we know that the neutral element in the command quantale coincides with the greatest element $\top$ in the assertion quantale, i.e., $[\![\mathsf{true}]\!] = [\![\,\mathsf{skip}\,]\!]$ holds.

Up to now we have defined for our relational view for commands in separation logic a quantale for test elements and one for commands so that we are able to treat both algebraically. But there is still a connection between both quantales needed to be able to use the separating conjunction in algebraic proofs of programs. We will use for this purpose an inference rule, called the *frame rule* [ORY01]. It describes an interaction between the separating conjunction in pre- and postconditions using arbitrary commands c. We will define and explain this rule in the following and afterwards use it to define the whole algebraic structure which we will use in the next chapter for proving correctness of a program example.

4.2.1 The Frame Rule

In this section we present the frame rule which is an important key for local reasoning in separation logic. By local reasoning it is meant to have a modular view of the program proofs. One can build up a correctness proof for the whole program by proving certain program parts, each of them independently and afterwards putting them together by applying the frame rule. It is defined as follows:

Definition 4.2.2 (Frame Rule) Let p, q, r be arbitrary assertions and c an arbitrary command. Then the *Frame Rule* in separation logic is defined by

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

assuming that no free variable of r is modified by c.

The premise $\{p\}\, c\, \{q\}$ of this rule ensures that starting the execution of c in a state satisfying p ends in a state satisfying q. Furthermore the conclusion says that extending the initial heap of the state with disjoint parts will not affect the execution of the triple in the premise. The additional heap cells remain unchanged, as long as no free variable of r is changed by c.

Example 4.2.3 As a simple example consider $p = x \mapsto 10$, $q = x \mapsto 20$, $r = y \mapsto 25$ and $c = ([x] := 20)$. Then the frame rule can be instantiated to

$$\frac{\{x \mapsto 10\}\, [x] := 20\, \{x \mapsto 20\}}{\{x \mapsto 10\, *\, y \mapsto 25\}\, [x] := 20\, \{x \mapsto 20\, *\, y \mapsto 25\}}$$

The command $[x] := 20$ in the premise does not change the values of the variables x and y. Hence only the heap cell located at address x lies in its frame. It is clear that the postcondition of the conclusion is satisfied by the final state, since c does not change y or the heap cell located at y according to the premise.

The reader should keep in mind that replacing the separating conjunction by the Boolean conjunction in the frame rule makes it invalid. Looking at such a triple where p is replaced by the imprecise assertion $x \mapsto - =_{df} \exists\, v.\ x \mapsto v$,

$$\frac{\{x \mapsto -\}\,[x] := 20\,\{x \mapsto 20\}}{\{x \mapsto -\ \wedge\ y \mapsto 25\}\,[x] := 20\,\{x \mapsto 20\ \wedge\ y \mapsto 25\}}$$

is not valid anymore, since it is not clear whether x and y contain the same address and whether the command $[x] := 20$ does not change the heap cell located at address y so that a satisfied postcondition might look like $x \mapsto 20\ \wedge\ y \mapsto 20$. □

As mentioned before we now use the frame rule to define the interplay between relational composition and the $\uplus$ operator in our relational view. We can encode this rule algebraically as follows:

$$[\![p]\!]; [\![c]\!] \subseteq [\![c]\!]; [\![q]\!] \Rightarrow ([\![p]\!] \uplus [\![r]\!]); [\![c]\!] \subseteq [\![c]\!]; ([\![q]\!] \uplus [\![r]\!])$$

assuming that $[\![p]\!]$, $[\![q]\!]$, $[\![r]\!]$ are arbitrary test elements and $[\![c]\!]$ is a command element. Furthermore we have to ensure that c does not modify any free variables of the assertion r. The whole algebraic structure is summarized in the following lemma.

Lemma 4.2.4 *Let* $(test(REL(States)), \cup, \emptyset, \uplus, [\![\,\mathsf{emp}\,]\!], \neg)$ *be the assertion quantale and* $(\mathcal{P}(REL(States)), \cup, \emptyset, ;, [\![\,\mathsf{skip}\,]\!], {}^*)$ *be the command quantale extended by the* * *operator. Then the frame rule*

$$[\![p]\!]; [\![c]\!] \subseteq [\![c]\!]; [\![q]\!] \Rightarrow ([\![p]\!] \uplus [\![r]\!]); [\![c]\!] \subseteq [\![c]\!]; ([\![q]\!] \uplus [\![r]\!])$$

holds for arbitrary test elements $[\![p]\!]$, $[\![q]\!]$, $[\![r]\!]$ *and command element* $[\![c]\!]$, *assuming that* c *does not modify any free variables of the assertion* r, *defining the set* $[\![r]\!]$. *We call this structure a Separation Algebra.*

Next we present the inference rules for commands Reynolds has given in [Rey08a]. We give some explanation for these rules and will see that the rules are interderivable. In particular, these rules become consequences of the relational view and the definitions of the commands we have given so far.

4.3 Inference Rules

We start by giving inference rules for the mutation and disposal commands. Using the relational view of these inference rules they become simple consequences. Furthermore we explain some rules for allocating heap cells. These rules need some more explanation since the allocation command cons is non-deterministic as already mentioned. Finally we give two inference rules for lookup commands. For these commands there is a big variety of rules which we do not list in this report. They can be found in [Rey08a].

4.3.1 Mutation and Disposal

In the previous section we defined relational semantics for commands used to change values of heap cells or to delete them. In this section we have a closer look at some inference rules and explain why this axiomatization makes sense.

The following laws denote valid triples for changing values of heap cells.

Theorem 4.3.1 *Let p, r be an arbitrary assertion and e, e' be $\langle exp \rangle$-expressions. Then valid triples for mutation commands are*

$$\frac{}{\{e \mapsto -\}\,[e] := e'\,\{e \mapsto e'\}} \quad (\mathit{local})$$

$$\frac{}{\{(e \mapsto -) * r\}\,[e] := e'\,\{(e \mapsto e') * r\}} \quad (\mathit{global})$$

$$\frac{}{\{(e \mapsto -) * ((e \mapsto e') -\!\!* \, p)\}\,[e] := e'\,\{p\}} \quad (\mathit{backward\text{-}reasoning})$$

Proof. To prove the local rule, let $(s, h) \in [\![e \mapsto -]\!]$. Therefore the condition $\langle e \rangle_{\text{exp}}(s) \in \mathsf{dom}(h)$ in the definition of mutation commands is satisfied. Finally the updated state $(s, (a, v) \mid h)$, whereas $a = \langle e \rangle_{\text{exp}}(s) \in \mathsf{dom}(h)$ and $v = \langle e' \rangle_{\text{exp}}(s)$, satisfies the postcondition $e \mapsto e'$.

The global mutation rule can be proved by instantiating the frame rule, using the local law as its premise.

For a proof of the backward-reasoning rule, we instantiate the global rule by setting $r = (e \mapsto e') -\!\!* \, p$. Therefore we get the postcondition $(e \mapsto e') * ((e \mapsto e') -\!\!* \, p)$. Finally, by setting $q = e \mapsto e'$, we only have to prove $q * (q -\!\!* \, p) \Rightarrow p$ which was shown in Chapter 3 and thus the claim follows. □

The local mutation law is very simple to understand, since it only says: if there exists an allocated heap cell at address e then it will have the value e' after the execution of the assignment in $[e] := e'$. Notice that the precondition was assured in the relational semantics by $\langle e \rangle_{\text{exp}}(s) \in \mathsf{dom}(h)$. The global mutation law is just an instantiation of the frame rule with the use of the local mutation rule. Hence, it says that such a process can be modularized. The local rule can be derived from the global one by setting $r = \mathsf{emp}$.

The backward-reasoning law is often used in a situation where we just have the postcondition of a mutation and need to find a precondition so that p holds after the execution. The precondition $(e \mapsto -) * ((e \mapsto e') -\!\!* \, p)$ says that if the postcondition holds in a state described after the execution there must exist an allocated heap cell at address e which is disjoint from the rest of the heap. The assertion $(e \mapsto e') -\!\!* \, p$ describes that part of the whole heap from which the just-mentioned heap cell is excluded.

The global and the backward-reasoning rules are also interderivable. One direction was already shown in the proof. For the reverse direction, we set $p = (e \mapsto e') * r$ in the backward-reasoning rule and get the precondition $(e \mapsto -) * (e \mapsto e' -\!\!*((e \mapsto e') * r))$. Hence, by setting $q = (e \mapsto e')$, we only need to show $r \Rightarrow q -\!\!*(q * r)$ which was already proved in the previous chapter.

The proofs for next commands is very similar to the proof just given and therefore will be omitted. For the command $\mathsf{dispose}$ we have the following rules.

Theorem 4.3.2 *Let e be an $\langle exp \rangle$-expression and r be an arbitrary assertion. Then the following triples for disposal are*

$$\frac{}{\{e \mapsto -\}\ \mathsf{dispose}\ \ e\,\{\,\mathsf{emp}\,\}} \quad (local)$$

$$\frac{}{\{(e \mapsto -) * r\}\ \mathsf{dispose}\ \ e\,\{r\}} \quad (global)$$

This means that for deleting a cell at address e there must exist an allocated heap cell at address e and afterwards the heap is either empty, in case of the local disposal law, or only the disjoint rest of the heap remains.

4.3.2 Allocation

In separation logic, the process of allocation of cells on the heap is important. This process is, as already mentioned, non-deterministic, i.e., arbitrary not yet allocated n consecutive cells on the heap are used by the command $\mathsf{cons}\,(e_1, \ldots, e_n)$. We present two kinds of laws for this command. The difference between these laws lies in the fact that the variable to which the address of the first cell is assigned, sometimes is used in the expressions $e_1, \ldots, e_n$ of cons. Therefore we distinguish between non-overwriting and overwriting allocation laws. The expressions in the postconditions of the overwriting allocation laws are always evaluated on a store s of a state that satisfies the precondition. Therefore it will be needed to use quantified variables denoting the value before and after the assignment to v. We write v'' to denote the value of v after the assignment and v' to denote the value before the assignment to v.

In the remainder we first list the rules and then explain their meaning.

Theorem 4.3.3 (Non-overwriting allocation)

$$\frac{}{\{\,\mathsf{emp}\,\}\,v := \ \mathsf{cons}\,(e_1, \ldots, e_n)\,\{v \mapsto e_1, \ldots, e_n\}} \quad (local)$$

$$\frac{}{\{r\}\,v := \ \mathsf{cons}\,(e_1, \ldots, e_n)\,\{(v \mapsto e_1, \ldots, e_n) * r\}} \quad (global)$$

assuming v has no free occurrence in the expressions $e_1, \ldots, e_n$ and in the assertion r.

Non-overwriting allocation assumes that the variable v has no free occurrence in the expressions of cons and in the assertion r. That means these expressions and the assertion are not affected by an assignment of the output address of cons to v. Therefore they remain unchanged by the execution of the command. Furthermore the n consecutive cells are new allocated cells and therefore disjoint from the heap portion satisfying r. This is expressed using the $*$ operator.

Theorem 4.3.4 (Overwriting allocation) *We write e'_i for the expression e_i where every occurence of the variable v is replaced by v'. Likewise we write r' for the assertion r where v is replaced by v' and r'' where v is replaced by v''. Then the overwriting allocation rules are*

$$\frac{}{\{v' = v \ \wedge \ \mathsf{emp}\,\}\,v := \ \mathsf{cons}\,(e_1, \ldots, e_n)\,\{v \mapsto e'_1, \ldots, e'_n\}} \quad (local)$$

$$\frac{}{\{r\}\,v := \ \mathsf{cons}\,(e_1, \ldots, e_n)\,\{\exists\, v'.\ (v \mapsto e'_1, \ldots, e'_n) * r'\}} \quad (global)$$

$$\frac{}{\{\forall\, v''.\ (v'' \mapsto e_1, \ldots, e_n) -\!\!* p''\}\,v := \ \mathsf{cons}\,(e_1, \ldots, e_n)\,\{p\}} \quad (backward\text{-}reasoning)$$

assuming v', v'' have no free occurrences in the expressions $e_1, \ldots, e_n$ and in the assertion r.

Let us explain these laws. The variable v' in the local law is used to keep the old value of v before it is overwritten by cons. In the postcondition every occurrence of the variable v is replaced by the variable v', since v now contains the address of the first cell allocated by cons. Therefore the postcondition $v \mapsto e_1, \ldots, e_n$ would be definitely false.

In the global allocation law the variable v' is an existentially quantified variable, in contrast to the local one. That means the old value of v is not necessaritly saved in v' as in the local law. Hence the postcondition is only satisfied when v' contains the old value of v.

The third of these laws is again for finding a satisfied precondition if the postcondition p is given. In this situation one has to be careful with cons, since it allocates non-deterministically n consecutive non-allocated cells on the heap. That means the heap of the state that satisfies the postcondition p must have these n allocated consecutive cells. For the precondition we have to consider this fact and therefore we need to quantify over all possible initial addresses of the n just allocated cells. In summary, the precondition characterizes all states, for which it is possible to insert n consecutive cells with contents $e_1, \ldots, e_n$ at an arbitrary address v'' into the heap such that afterwards the condition p'' is satisfied.

The reader should realize that the variable v in a state of the postcondition is already overwritten in contrast to the variable v of a state of the precondition. In such a state v still holds its original value and v'' denotes the new value or the assigned address to v after the execution of cons.

4.3.3 Lookup

For the lookup commands there is a large variety of inference rules in [Rey08a]. We only give the non-overwriting ones, since the others remain uninteresting here.

Theorem 4.3.5 (Non-overwriting lookup)

$$\frac{}{\{e \mapsto v'\}\, v := [e]\, \{v = v' \land (e \mapsto v)\}} \quad (local)$$

$$\frac{}{\{\exists\, v'.\ (e \mapsto v') * p'\}\, v := [e]\, \{(e \mapsto v) * p\}} \quad (global)$$

where $v \notin FV(e)$, $v' \notin FV(e) \cup (FV(p) - \{v\})$, and p' stands for p where every free occurrence of v is replaced by v'.

These laws should be clear in their meaning with the explanation given in the previous section since the usage of v' is the same as in the allocation rules.

In the next chapter we start with some basic definitions and notations for the list reversal algorithm and then prove it algebraically using our relational view on separation logic.

Chapter 5

Proving the List Reversal Algorithm

The in-place list-reversal algorithm is a standard example for proving correctness of programs. It has been studied in a lot of works, e.g. [Möl93, Rey08a] and will be proved correct here using the presented algebraic approach. First we give the algorithm and its specification in separation logic according to [Rey08a]. Furthermore we add some major definitions and a small example to understand the pointer modifications of the algorithm. Afterwards we identify some problems in the specification and show how we can eliminate them. Finally we give an algebraic proof of the algorithm.

5.1 Definitions and an Example

In this section we present the algorithm with its specification in separation logic. The task is to reverse a given list by simple pointer modifications. Hence no additional memory is needed. In separation logic lists are defined inductively along the structure of a sequence representing the contents of the whole list. Sequences over a given alphabet Σ should be well known. We write greek letters α, β, … to denote sequences while the elements of these sequences are values. Furthermore, we write ε for the empty sequence, $\cdot$ for the composition of sequences and $\alpha^\dagger$ for the reverse of a sequence α. We denote the set of all sequences by Seq.

The predicate for list structures in separation logic is defined as follows:

Definition 5.1.1 Consider an address $a \in \mathbb{Z}$, arbitrary sequences α, α' and a store variable i. Then the predicate $\mathsf{list}\ \alpha\ \mathsf{i}$ expresses that the sequence α is represented in the current state by a list whose head cell is at the address saved in i. It is inductively defined by

$$\begin{array}{rcl} \mathsf{list}\ \varepsilon\ \mathsf{i} & \Leftrightarrow_{df} & \mathsf{emp}\ \wedge\ \mathsf{i} = \mathsf{nil} \\ \mathsf{list}\ (a \cdot \alpha')\ \mathsf{i} & \Leftrightarrow_{df} & \exists\, \mathsf{j}.\ \mathsf{i} \mapsto a, \mathsf{j} * \mathsf{list}\ \alpha'\ \mathsf{j} \end{array}$$

where $\alpha = \varepsilon$ in the first and $\alpha = a \cdot \alpha'$ in the second case.

Remark 5.1.2 Since the list predicate is inductively defined, the sets $[\![\mathsf{list}\ \alpha\ \mathsf{i}]\!]$ in our relational view are also inductively defined. □

In case of an empty list, the variable i has to contain a nil pointer and the heap portion of this list has to be empty. If the list consists of at least one record then the sequence α has at least one element. By this definition we put two adjacent heap cells together to form one list record. The first heap cell contains the first sequence element while the second heap cell contains the starting address of the next list cell. The quantified variable j denotes an *anonymous* pointer.

From this inductive definition of lists, one can see an advantage we get by using the separating conjunction. Under the assumption that the represented sequence has n elements, we automatically get by the list predicate an acyclic list which will be proved in the following.

Lemma 5.1.3 *Given an arbitrary sequence α and a variable i. A list characterized by the predicate $\mathsf{list}\ \alpha\ \mathsf{i}$ does not contain any cycles.*

Proof. We show this by induction over the length n of α.
$\mathbf{n = 0}$: Trivial.
$\mathbf{n = 1}$: Let $\alpha = \mathsf{a}$ for an arbitrary $\mathsf{a} \in \mathbb{Z}$.

$$\begin{array}{ll} & \mathsf{list\ a\ i} \\ \Leftrightarrow & \quad \{\!\![\text{ definition }]\!\!\} \\ & \exists \mathsf{j}.\, \mathsf{i} \mapsto \mathsf{a}, \mathsf{j} * \mathsf{list}\ \varepsilon\ \mathsf{j} \\ \Leftrightarrow & \quad \{\!\![\text{ definition }]\!\!\} \\ & \exists \mathsf{j}.\, \mathsf{i} \mapsto \mathsf{a}, \mathsf{j} * (\mathsf{emp} \wedge \mathsf{j} = \mathsf{nil}) \\ \Leftrightarrow & \quad \{\!\![\text{ Lemma 3.5.6, neutrality }]\!\!\} \\ & \exists \mathsf{j}.\, \mathsf{i} \mapsto \mathsf{a}, \mathsf{j} \wedge \mathsf{j} = \mathsf{nil} \\ \Leftrightarrow & \quad \{\!\![\text{ trivial }]\!\!\} \\ & \mathsf{i} \mapsto \mathsf{a}, \mathsf{nil} \end{array}$$

⇒ Hence there can not be any cycles in a single record.

$\mathbf{n \to n + 1}$: Let $\alpha = \mathsf{a} \cdot \alpha'$ for arbitrary $\mathsf{a} \in \mathbb{Z}, \alpha' \in Seq$ and $\alpha' \neq \varepsilon$.

We know by definition that $\mathsf{list}\ (a \cdot \alpha')\ \mathsf{i} \Leftrightarrow_{df} \exists \mathsf{j}.\, \mathsf{i} \mapsto \mathsf{a}, \mathsf{j} * \mathsf{list}\ \alpha'\ \mathsf{j}$. Since $\alpha \neq \varepsilon$ furthermore $\mathsf{j} \neq \mathsf{nil}$ has to hold and contains a valid address. From the definition of $*$ we finally get that $\mathsf{i} \neq \mathsf{j}$ has to hold. Moreover by induction hypothesis we know that the list starting at j is cycle-free for a quantified variable j. By linking a distinct single record $\mathsf{i} \mapsto \mathsf{a}, \mathsf{j}$ to the beginning of the existing list can not produce any cycles. Thus the whole list remains cycle-free.

□

Notice, for a precise proof to show that sharing is prohibited, we would have to prove that no cycle is reachable by the pointer i. Therefore it would be necessary to include properties for reachability. But this is left for further work. For the list predicate a few further properties are summarized in the next corollary.

Corollary 5.1.4 *Let $\alpha \in Seq$ and i be a variable. Then*

$$\begin{array}{rcl} \mathsf{i} = \mathsf{nil} \wedge \mathsf{list}\ \alpha\ \mathsf{i} & \Rightarrow & \mathsf{emp} \wedge \alpha = \varepsilon \\ \alpha = \varepsilon \wedge \mathsf{list}\ \alpha\ \mathsf{i} & \Rightarrow & \mathsf{emp} \wedge \mathsf{i} = \mathsf{nil} \end{array}$$

and furthermore

$$\mathsf{i} \neq \mathsf{nil} \wedge \mathsf{list}\ \alpha\ \mathsf{i} \Leftrightarrow \alpha \neq \varepsilon \wedge \mathsf{list}\ \alpha\ \mathsf{i} \Leftrightarrow \exists a, \alpha'.\ \mathsf{list}\ (a \cdot \alpha')\ \mathsf{i} \wedge \alpha = a \cdot \alpha'$$

Now we are able to give the in-place list-reversal algorithm with its annotated specification in separation logic, according to [Rey08a]. The list at the address i, representing the sequence α_0, will be reversed.

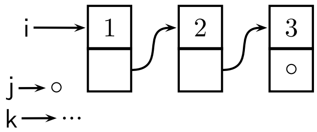
```
$\{\mathsf{list}\ \alpha_0\ \mathsf{i}\}$
$\{\mathsf{list}\ \alpha_0\ \mathsf{i} * (\mathsf{emp} \wedge \mathsf{nil} = \mathsf{nil})\}$
$\mathsf{j} := \mathsf{nil};$
$\{\mathsf{list}\ \alpha_0\ \mathsf{i} * (\mathsf{emp} \wedge \mathsf{j} = \mathsf{nil})\}$
$\{\mathsf{list}\ \alpha_0\ \mathsf{i} * \mathsf{list}\ \varepsilon\ \mathsf{j}\}$
$\{\exists\, \alpha, \beta.\ (\mathsf{list}\ \alpha\ \mathsf{i} * \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\}$
$\mathsf{while\ i} \neq \mathsf{nil\ do}$
  $\big(\{\exists\, \mathsf{a}, \alpha, \beta.\ (\mathsf{list}\ (\mathsf{a}{\cdot}\alpha)\ \mathsf{i} * \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger{\cdot}\beta\}$
  $\{\exists\, \mathsf{a}, \mathsf{k}, \alpha, \beta.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{k} * \mathsf{list}\ \alpha\ \mathsf{k} * \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger{\cdot}\beta\}$
  $\mathsf{k} := [\mathsf{i} + 1];$
  $\{\exists\, \mathsf{a}, \alpha, \beta.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{k} * \mathsf{list}\ \alpha\ \mathsf{k} * \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger{\cdot}\beta\}$
  $[\mathsf{i} + 1] := \mathsf{j};$
  $\{\exists\, \mathsf{a}, \alpha, \beta.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{j} * \mathsf{list}\ \alpha\ \mathsf{k} * \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger{\cdot}\beta\}$
  $\{\exists\, \mathsf{a}, \alpha, \beta.\ (\mathsf{list}\ \alpha\ \mathsf{k} * \mathsf{list}\ \mathsf{a}{\cdot}\beta\ \mathsf{i}) \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}(\mathsf{a}{\cdot}\beta)\}$
  $\{\exists\, \alpha, \beta.\ (\mathsf{list}\ \alpha\ \mathsf{k} * \mathsf{list}\ \beta\ \mathsf{i}) \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\}$
  $\mathsf{j} := \mathsf{i};$
  $\mathsf{i} := \mathsf{k};$
  $\{\exists\, \alpha, \beta.\ (\mathsf{list}\ \alpha\ \mathsf{i} * \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\}\big)$
$\{\exists\, \alpha, \beta.\ \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta \wedge \alpha = \varepsilon\}$
$\{\mathsf{list}\ \alpha_0^\dagger\ \mathsf{j}\}$
```

First we start by some explanation of the semantics behind the pre- and postconditions of this specification. The formulas in braces denote the pre- or postconditions of commands or of weakening rules. Here the sequences α and β are existentially quantified and used as *ghost variables*. That means that these variables do not occur in the program itself as store variables. They are used to denote placeholders for sequences that satisfy the pre- and postconditions, i.e., they are meta-variables and work at a different level of abstraction. Moreover the reader should notice that the variable α in the assertion after entering the while-loop differs from α in the subsequent assertion, since it only represents the tail sequence of the original α value (cf. Corollary 5.1.4).

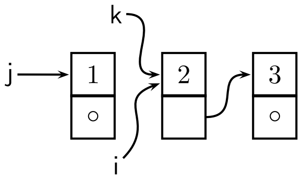
The situation is different for the quantified variables a and k. It is not clear if the semantics of a has to be handled like that of sequence variables, i.e., store-independent, or as it is given in Definition 3.1.3. But in both cases it remains a logical variable. The variable k is confusingly used in two ways. After entering the while-loop the variable k is existentially quantified and not used by the program, hence it is a logical variable. Contrarily the next command assigns a value to k and hence it becomes a program variable, i.e., physical content of the store. We will later present a way how to overcome this difficulty with ghost variables and treat all occurring variables in the same way which will make the program more readable.

To see what the pointer modifications in this algorithm do, we give a small example.

Example 5.1.5 We consider the three-element sequence $\alpha_0 = 1\,2\,3$ and i pointing to the list representing this sequence on the heap. Before entering the while loop, we have the following situation, where j is a nil pointer and k has some arbitrary contents. At the beginning the ghost variables in the invariant have the contents $\alpha = 1\,2\,3$ and $\beta = \varepsilon$. The symbol ○ in a cell denotes that is has no reference, i.e. the value nil.

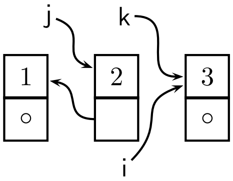


After the first iteration of the while loop the situation looks as follows.

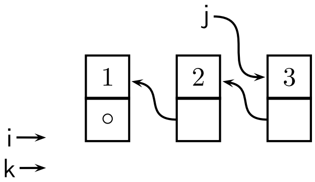


The content of pointer i is now the address of the second list cell and j saves the address of the first list cell. The variable k is always used to save temporary addresses. Furthermore $\alpha = 2\,3$ and $\beta = 1$ are the values of the ghost variables for this iteration.

After the second iteration the address cell of the second list cell points to the first list cell. Every pointer from the previous iteration has now moved over to the next unchanged list cell. For this iteration we have $\alpha = 3$ and $\beta = 2\,1$.



Finally the algorithm terminates with the reversed list starting at an address saved in j. The variables i and k contain nil-pointers and $\alpha = \varepsilon$, $\beta = 3\,2\,1$ are the values for the ghost variables after loop termination.



□

The reader may have noticed that the variable k is not initialized with nil at the beginning of the algorithm. It is possible that we overwrite a pointer when executing this algorithm and therefore may get side effects. Hence, to apply the frame rule it is essential to claim that we must not modify any free variable of the added assertion, i.e., the assertion r in Definition 4.2.2.

In the next section we give an opportunity to get the before mentioned difficulties with ghost variables under control. Afterwards we are ready to give a correctness proof of the list-reversal algorithm using the algebraic approach presented so far.

5.2 Getting Ghost Variables Under Control

As mentioned before the ghost variables for the sequences α, β in the list-reversal algorithm work at a different meta-level and are not part of the program state. To simplify this and to get a consistent treatment for all variables we redefine the stores to let them also handle sequences. Therefore we first introduce a new kind of store variables.

Definition 5.2.1 (Sequence variables) Let V be the set of all variables. Then V_{Seq} denotes the set of all sequence variables. We use greek letters α, β, ... to denote sequence variables. Furthermore $V =_{df} V_s \cup V_{Seq}$ where $V_s \cap V_{Seq} = \emptyset$ and V_s is the usual set of all program store variables, denoted by the letters x, y, z.

In particular, we require sequence variables not to occur in the program itself but still to be a physical part of the store. We use them in the following for an algebraic treatment and verification purposes only. With this definition, we define our stores as follows:

$$Stores \quad =_{df} \quad V_s \rightsquigarrow Values \; \cup \; V_{Seq} \rightsquigarrow Seq$$

Now we are able to handle the sequence variables with our algebraic operators by calculating on states using the new store definition. The next step we take is an extension of the syntax and semantics of expressions to evaluate sequences on a given store. Therefore we expand Definition 2.1.1 by the entries

$$\begin{array}{rcl} \langle\text{seq}\rangle & ::= & \varepsilon \,|\, \alpha \,|\, \beta \,|\, \ldots \\ & & \langle\text{var}\rangle \cdot \langle\text{seq}\rangle \,|\, \langle\text{seq}\rangle \cdot \langle\text{var}\rangle \,|\, \langle\text{seq}\rangle \cdot \langle\text{seq}\rangle \,|\, \langle\text{seq}\rangle^{\dagger} \\ \langle\text{bexp}\rangle & ::= & \ldots \,|\, \langle\text{seq}\rangle = \langle\text{seq}\rangle \,|\, ... \end{array}$$

The semantics given in Definition 2.1.2 can be extended in a straightforward way.

Looking at the pre- and postconditions in the verification of the algorithm, one can see that the quantifiers over the ghost variables range over the whole formulas. This will make an algebraic treatment of the assertions more difficult. The next step we take is an alternative to avoid the quantifications while preserving the semantics.

For this purpose we let ghost variable definitions be saved in the store with the restriction that these will never be used as program variables. Therefore we have to keep these variables disjoint from the usual store variables which must be a physical part of the store and can be used by the program. We write $V_{pro} \subseteq V$ to denote the variables which can be used by the program and $V_{ghost} \subseteq V$ for ghost variables and require

$$V \quad =_{df} \quad V_{pro} \cup V_{ghost}$$

assuming $V_{pro} \cap V_{ghost} = \emptyset$. In particular, no variable used by a program can be a ghost variable and vice versa. To distinguish ghost variables in assertions from program variables we use $k, l, ...$ for them. Since sequences are never used by programs and therefore are always ghosts, there is no need to mark them. Furthermore we have to avoid situations in which a ghost variable is used at the same time as a free and a bound variable in a formula. Therefore we assume that a variable is either free or bound by a quantifier. Otherwise we would get problems with the semantics. Hence, we have to resolve such conflicts by renaming the variables. The same has to be done when a variable is bound by an $\exists$- and an $\forall$-quantifier at the same time.

By this we reproduce the original behavior of the stores for programs and are able to calculate with ghost variables algebraically in the usual way. The new stores are illustrated in the following figure for an arbitrary store s:

V_s
V_{Seq}
s
V_{pro}
V_{ghost}
s

Example 5.2.2 As a small example we consider the loop invariant of the list-reversal algorithm, viz.

$$\{\exists\, \alpha, \beta.\, (\mathsf{list}\ \alpha\ \mathsf{i} * \mathsf{list}\ \beta\ \mathsf{j})\ \wedge\ \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\}.$$

The variables i and j are usual store variables which can be used by a program. Furthermore the variable α_0 is a ghost variable which by the definitions above, now becomes a physical content of the store. The ghost variables α and β are existentially quantified but also are part of the store now. We can transform this formula using our new semantics into the shorter form

$$\{(\mathsf{list}\ \alpha\ \mathsf{i} * \mathsf{list}\ \beta\ \mathsf{j})\ \wedge\ \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\}.$$

The ghost variables α and β are now evaluated on the store. Therefore, this formula holds in a particular store iff the ghosts α and β have suitable values, which is precisely what the original form with the quantification asserts. □

As discussed in the problems for the annotated specification given for the list-reveral algorithm, ghost variables change their contents from one pre- or postcondition to the subsequent one. This behaviour comes implicitly with their existential quantification. To describe these changes we now introduce the concept of *ghost commands* which we use for changing contents of ghost variables only. The basic idea is to extend the usual specification by ghost commands in such a way that it still remains valid after removing the $\exists$-quantifiers. Notice that an extension by ghost commands will not change the original program itself since these commands only manipulate ghost variables. We first extend the syntax of the program commands by

$$\begin{aligned}\langle\text{comm}\rangle\ \ ::=\ \ & ...\,|\,\langle\text{seq}\rangle := \langle\text{seq}\rangle\,|\,\langle\text{ghostvar}\rangle := \mathsf{head}(\langle\text{seq}\rangle)\,|\,\langle\text{seq}\rangle := \mathsf{tail}(\langle\text{seq}\rangle)\\ & |\,\langle\text{ghostvar}\rangle := [\langle\text{exp}\rangle]\,|\,\langle\text{ghostvar}\rangle := \langle\text{exp}\rangle\,|\,...\end{aligned}$$

where $\langle\text{ghostvar}\rangle$ denotes logical variables of V_{ghost} which do not save sequences. The functions head and tail are the usual known functions on sequences which are used later for a rewritten specification. Furthermore we allow ghost commands to dereferences heap cells. But we have to ensure every time when using these commands that such lookups do not use unallocated addresses which will immediately cause a fault in their execution. Next we give some intuitive inference rules for these commands and afterwards give a specification of the list-reversal algorithm using ghost commands.

Assignments to ghost variables can then be handled in the usual way with the assignment rule of Hoare logic resulting from a unique treatment.

Corollary 5.2.3 *The following inference rules for assignments to ghost variables hold. For an assertion p, arbitrary sequences α, β and $a \in V_{ghost}$, $v \in \mathbb{Z}$:*

$$\frac{}{\{p\}\,\alpha := \beta\,\{\alpha = \beta\ \wedge\ p'\}} \qquad \frac{}{\{p\}\ a := v\,\{a = v\ \wedge\ p'\}}$$

assuming p' is p where every occurence of β and v is replaced by α and a, resp.

With this knowledge we now give a rewritten version of the in-place list-reversal algorithm which is, albeit longer, much more perpicuous and easier to use for an algebraic treatment:

```
$\{\mathsf{list}\ \alpha_0\ \mathsf{i}\}$
$\{\mathsf{list}\ \alpha_0\ \mathsf{i} * (\mathsf{emp}\ \wedge\ \mathsf{nil} = \mathsf{nil})\}$
$\mathsf{j} := \mathsf{nil};$
$\{\mathsf{list}\ \alpha_0\ \mathsf{i} * (\mathsf{emp}\ \wedge\ \mathsf{j} = \mathsf{nil})\}$
$\{\mathsf{list}\ \alpha_0\ \mathsf{i} * \mathsf{list}\ \varepsilon\ \mathsf{j}\}$
$\alpha := \alpha_0;$
$\{(\mathsf{list}\ \alpha\ \mathsf{i} * \mathsf{list}\ \varepsilon\ \mathsf{j})\ \wedge\ \alpha = \alpha_0\}$
$\beta := \varepsilon;$
$\{(\mathsf{list}\ \alpha\ \mathsf{i} * \mathsf{list}\ \beta\ \mathsf{j})\ \wedge\ \alpha = \alpha_0\ \wedge\ \beta = \varepsilon\}$
$\{(\mathsf{list}\ \alpha\ \mathsf{i} * \mathsf{list}\ \beta\ \mathsf{j})\ \wedge\ \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\}$
$\mathsf{while}\ (\mathsf{i} \neq \mathsf{nil})\ \mathsf{do}\ \big($
  $a := \mathsf{head}(\alpha);$
  $\{(\mathsf{list}\ \alpha\ \mathsf{i} * \mathsf{list}\ \beta\ \mathsf{j})\ \wedge\ a = \mathsf{head}(\alpha)\ \wedge\ \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\}$
  $\alpha' := \mathsf{tail}(\alpha);$
  $\{(\mathsf{list}\ \alpha\ \mathsf{i} * \mathsf{list}\ \beta\ \mathsf{j})\ \wedge\ a = \mathsf{head}(\alpha)\ \wedge\ \alpha' = \mathsf{tail}(\alpha)\ \wedge\ \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\}$
  $\{(\mathsf{list}\ (a{\cdot}\alpha')\ \mathsf{i} * \mathsf{list}\ \beta\ \mathsf{j})\ \wedge\ \alpha = a{\cdot}\alpha'\ \wedge\ \alpha_0^\dagger = (a{\cdot}\alpha')^\dagger{\cdot}\beta\}$
  $\{(\mathsf{list}\ (a{\cdot}\alpha')\ \mathsf{i} * \mathsf{list}\ \beta\ \mathsf{j})\ \wedge\ \alpha_0^\dagger = (a{\cdot}\alpha')^\dagger{\cdot}\beta\}$
  $l := [\mathsf{i} + 1];$
  $\{(\mathsf{i} \mapsto a, l * \mathsf{list}\ \alpha'\ l * \mathsf{list}\ \beta\ \mathsf{j})\ \wedge\ \alpha_0^\dagger = (a{\cdot}\alpha')^\dagger{\cdot}\beta\}$
  $\mathsf{k} := [\mathsf{i} + 1];$
  $\{(\mathsf{i} \mapsto a, \mathsf{k} * \mathsf{list}\ \alpha'\ \mathsf{k} * \mathsf{list}\ \beta\ \mathsf{j})\ \wedge\ \alpha_0^\dagger = (a{\cdot}\alpha')^\dagger{\cdot}\beta\}$
  $[\mathsf{i} + 1] := \mathsf{j};$
  $\{(\mathsf{i} \mapsto a, \mathsf{j} * \mathsf{list}\ \alpha'\ \mathsf{k} * \mathsf{list}\ \beta\ \mathsf{j})\ \wedge\ \alpha_0^\dagger = (a{\cdot}\alpha')^\dagger{\cdot}\beta\}$
  $\{(\mathsf{list}\ \alpha'\ \mathsf{k} * \mathsf{list}\ a{\cdot}\beta\ \mathsf{i})\ \wedge\ \alpha_0^\dagger = (\alpha')^\dagger{\cdot}(a{\cdot}\beta)\}$
  $\beta := a{\cdot}\beta;$
  $\{(\mathsf{list}\ \alpha'\ \mathsf{k} * \mathsf{list}\ \beta\ \mathsf{i})\ \wedge\ \alpha_0^\dagger = (\alpha')^\dagger{\cdot}\beta\}$
  $\alpha := \alpha';$
  $\{(\mathsf{list}\ \alpha\ \mathsf{k} * \mathsf{list}\ \beta\ \mathsf{i})\ \wedge\ \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\}$
  $\mathsf{j} := \mathsf{i};$
  $\mathsf{i} := \mathsf{k};$
  $\{(\mathsf{list}\ \alpha\ \mathsf{i} * \mathsf{list}\ \beta\ \mathsf{j})\ \wedge\ \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\}\big)$
$\{\mathsf{list}\ \beta\ \mathsf{j}\ \wedge\ \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\ \wedge\ \alpha = \varepsilon\}$
$\{\mathsf{list}\ \alpha_0^\dagger\ \mathsf{j}\}$
```

Notice, that the variable l is the renamed ghost variable k of the old specification. We have done this to distinguish it from the program variable k. Moreover ghost commands with their additional pre- and postconditions are printed in gray. All greek and italic printed variables are ghost variables while the sans-serif letters are program variables. The reader should also keep in mind that the line $\mathsf{k} := [\mathsf{i}+1]$ cannot be replaced by the assignment $\mathsf{k} := l$. Again the reason therefore is that l is a ghost variable and works on a different abstraction layer, i.e. it is not part of the current store.

5.3 An Algebraic Proof

Before giving an algebraic proof of the in-place list-reversal algorithm from the new specification, we first sum up some simple consequences for the list predicate using our algebraic setting. Furthermore we present a law that characterizes an interplay between ghost variables and assignments to store variables and add a consequence for heap-independent assignments.

Corollary 5.3.1 *Let* i, k *and* l *be variables where* l *is a ghost variable. Then the following law holds*

$$(\llbracket \mathsf{i} \mapsto l\,\rrbracket \uplus \llbracket p \rrbracket) ; \llbracket \mathsf{k} := [\mathsf{i}] \rrbracket \ \subseteq\ \llbracket \mathsf{k} := [\mathsf{i}] \rrbracket ; (\llbracket \mathsf{i} \mapsto \mathsf{k} \rrbracket \uplus \llbracket p' \rrbracket),$$

assuming p' *is the assertion* p *where every existentially quantified ghost variable* l *is replaced by* k.

Proof. The ghost variable l is used to save the value $h(s(\mathsf{i}))$. Therefore the assignment $\mathsf{k} := [\mathsf{i}]$ can be treated like the command $\mathsf{k} := v$ for a value $v = h(s(\mathsf{i})) \in \mathbb{Z}$ using the usual assignment rule. Hence for a postcondition p, we have to replace every occurrence of l by the store variable k in the precondition. □

We give a special case of alternative assignment rule (mentioned before) in the next corollary and encode it in the relational view as:

Corollary 5.3.2 *Let* x *be a variable. Furthermore, let* e *be an* $\langle exp \rangle$*-expression and* p *an assertion. We assume that both have no free occurences of* x*. Then we conclude*

$$\llbracket p \rrbracket ; \llbracket x := e \rrbracket \ \subseteq\ \llbracket x := e \rrbracket ; (\llbracket p \rrbracket \ \cap\ \llbracket x = e \rrbracket).$$

The next corollary is an encoding of the laws in Corollary 5.1.4 into our relation setting.

Corollary 5.3.3 *The following laws holds for the* list *predicate*

$$\begin{gathered}
\neg\llbracket \alpha = \varepsilon \rrbracket \ \cap\ \llbracket \mathsf{list}\ \alpha\ \mathsf{i} \rrbracket \ =\ \neg\llbracket \mathsf{i} = \mathsf{nil} \rrbracket \ \cap\ \llbracket \mathsf{list}\ \alpha\ \mathsf{i} \rrbracket \\
\llbracket \mathsf{list}\ (a{\cdot}\alpha)\ \mathsf{i} \rrbracket ; \llbracket l := [\mathsf{i}+1] \rrbracket \ \subseteq\ \llbracket l := [\mathsf{i}+1] \rrbracket ; (\llbracket \mathsf{i} \mapsto a \rrbracket \uplus \llbracket \mathsf{i}+1 \mapsto l \rrbracket \uplus \llbracket \mathsf{list}\ \alpha'\ l \rrbracket) \\
\llbracket \mathsf{i} = \mathsf{nil} \rrbracket \ \cap\ \llbracket \mathsf{list}\ \alpha\ \mathsf{i} \rrbracket \ \subseteq\ \llbracket\, \mathsf{emp}\, \rrbracket \ \cap\ \llbracket \alpha = \varepsilon \rrbracket
\end{gathered}$$

Now we are able to prove the algorithm with our algebraic approach. We split the proof into three parts. First we prove the part up to the while loop. Afterwards, we prove the while loop and finally give a simple conclusion to see that in fact we get our reversed list back at the end of the algorithm. To calculate algebraically, we abbreviate the concrete relations in the following by:

- $p_1 = \llbracket \mathsf{list}\ \alpha_0\ \mathsf{i} \rrbracket$
- $p_2 = \llbracket \mathsf{list}\ \varepsilon\ \mathsf{j} \rrbracket$
- $p_3 = \llbracket \mathsf{list}\ \alpha\ \mathsf{i} \rrbracket$
- $p_4 = \llbracket \mathsf{list}\ \beta\ \mathsf{j} \rrbracket$
- $p_5 = \llbracket \mathsf{list}\ \alpha'\ l \rrbracket$
- $p_6 = \llbracket \mathsf{list}\ \alpha'\ \mathsf{k} \rrbracket$
- $p_7 = \llbracket \mathsf{list}\ (a{\cdot}\beta)\ \mathsf{i} \rrbracket$
- $p_8 = \llbracket \mathsf{list}\ \beta\ \mathsf{i} \rrbracket$
- $p_9 = \llbracket \mathsf{list}\ \alpha'\ \mathsf{i} \rrbracket$
- $p_{10} = \llbracket \mathsf{list}\ {\alpha_0}^\dagger\ \mathsf{j} \rrbracket$
- $p_{11} = \llbracket \mathsf{list}\ (a{\cdot}\alpha')\ \mathsf{i} \rrbracket$
- $p_{12} = \llbracket \mathsf{list}\ \alpha\ \mathsf{k} \rrbracket$
- $q_1 = \llbracket {\alpha_0}^\dagger = \alpha^\dagger{\cdot}\beta \rrbracket$
- $q_2 = \llbracket \alpha = a{\cdot}\alpha' \rrbracket$
- $q_3 = \llbracket {\alpha_0}^\dagger = (\alpha')^\dagger{\cdot}(a{\cdot}\beta) \rrbracket$
- $q_4 = \llbracket \alpha = \varepsilon \rrbracket$
- $q_5 = \llbracket {\alpha_0}^\dagger = \beta \rrbracket$
- $q_6 = \llbracket \alpha = \alpha_0 \rrbracket$
- $q_7 = \llbracket \beta = \varepsilon \rrbracket$
- $q_8 = \llbracket a = \mathsf{head}(\alpha) \rrbracket$
- $q_9 = \llbracket \alpha' = \mathsf{tail}(\alpha) \rrbracket$
- $q_{10} = \llbracket {\alpha_0}^\dagger = (\alpha')^\dagger{\cdot}\beta \rrbracket$
- $r_1 = \llbracket \mathsf{i} \mapsto a \rrbracket$
- $r_2 = \llbracket \mathsf{i}+1 \mapsto l \rrbracket$
- $r_3 = \llbracket \mathsf{i}+1 \mapsto \mathsf{j} \rrbracket$
- $r_4 = \llbracket \mathsf{i}+1 \mapsto \mathsf{k} \rrbracket$
- $s_1 = \llbracket \mathsf{j} = \mathsf{nil} \rrbracket$
- $s_2 = \llbracket \mathsf{i} = \mathsf{nil} \rrbracket$

- $c_1 = \llbracket \mathsf{j} := \mathsf{nil} \rrbracket$
- $c_2 = \llbracket \mathsf{k} := [\mathsf{i}+1] \rrbracket$
- $c_3 = \llbracket [\mathsf{i}+1] := \mathsf{j} \rrbracket$
- $c_4 = \llbracket \mathsf{j} := \mathsf{i} \rrbracket$
- $c_5 = \llbracket \mathsf{i} := \mathsf{k} \rrbracket$
- $c_6 = \llbracket \alpha := \alpha_0 \rrbracket$
- $c_7 = \llbracket \beta := \varepsilon \rrbracket$
- $c_8 = \llbracket a := \mathsf{head}(\alpha) \rrbracket$
- $c_9 = \llbracket \alpha' := \mathsf{tail}(\alpha) \rrbracket$
- $c_{10} = \llbracket l := [\mathsf{i}+1] \rrbracket$
- $c_{11} = \llbracket \beta := a{\cdot}\beta \rrbracket$
- $c_{12} = \llbracket \alpha := \alpha' \rrbracket$

For the algebraic proofs, we also abstract from the concrete model and write $\cdot$ to denote the relation composition ; since it is used as the multiplication in the command quantale. Furthermore we use, for the assertion quantale, the notation $*$ for $\uplus$ and emp for $\llbracket\,\mathsf{emp}\,\rrbracket$ to avoid confusions. We sum up results which hold for the abbreviations above:

(1) According to Corollary 5.3.2 we get

$$\llbracket \mathsf{list}\ \alpha_0\ \mathsf{i} \rrbracket ; \llbracket \mathsf{j} := \mathsf{nil} \rrbracket \subseteq \llbracket \mathsf{j} := \mathsf{nil} \rrbracket ; (\llbracket \mathsf{list}\ \alpha_0\ \mathsf{i} \rrbracket \cap \llbracket \mathsf{j} = \mathsf{nil} \rrbracket)$$

which is algebraically encoded by $p_1 \cdot c_1 \le c_1 \cdot (p_1 \sqcap s_1)$.

(2) Using the alternative assignment rule we have

$$(p_1 * p_2) \cdot c_6 \le c_6 \cdot ((p_1 * p_2) \sqcap q_6)$$
$$((p_1 * p_2) \sqcap q_6) \cdot c_7 \le c_7 \cdot ((p_1 * p_2) \sqcap q_6 \sqcap q_7)$$

(3) A further law is $q_6 \cdot q_7 \le q_1$.

(4) The first law in 5.3.3 is encoded as $\neg s_2 \sqcap p_3 = \neg q_4 \sqcap p_3$

(5) Also we have $q_8 \sqcap q_9 = q_2$, $q_2 \sqcap q_1 \le q_3$ and $p_3 \sqcap q_2 \le p_{11}$

(6) The second law in 5.3.3 is encoded as $p_{11} \cdot c_{10} \le c_{10} \cdot (r_1 * r_2 * p_5)$ holds

(7) Using Lemma 5.3.1 we conclude

$$(\llbracket \mathsf{i}+1 \mapsto l \rrbracket \uplus \llbracket p \rrbracket) ; \llbracket \mathsf{k} := [\mathsf{i}+1] \rrbracket \subseteq \llbracket \mathsf{k} := [\mathsf{i}+1] \rrbracket ; (\llbracket \mathsf{i}+1 \mapsto \mathsf{k} \rrbracket \uplus \llbracket p' \rrbracket),$$

which encodes into $(r_2 * p) \cdot c_2 \le c_2 \cdot (r_4 * p')$ for suitable p, p'.

(8) By the global mutation inference rule $(r_4 * p) \cdot c_3 \leq c_3 \cdot (r_3 * p')$ holds for suitable p, p'

(9) Moreover $r_1 * r_3 * p_4 \leq p_8$ holds

(10) By Corollary 5.3.3 we get

$$[\![\mathsf{i} = \mathsf{nil}]\!] \cap [\![\mathsf{list}\ \alpha\ \mathsf{i}]\!] \subseteq [\![\mathsf{emp}]\!] \cap [\![\alpha = \varepsilon]\!]$$

which encodes into $s_2 \sqcap p_3 \leq \mathsf{emp} \sqcap q_4$.

(11) Furthermore we have

$$\alpha = \varepsilon \wedge {\alpha_0}^\dagger = \alpha^\dagger {\cdot} \beta \Rightarrow {\alpha_0}^\dagger = \beta$$

which encodes to $q_4 \sqcap q_1 \leq q_5$ and

$$\mathsf{list}\ \beta\ \mathsf{j} \wedge {\alpha_0}^\dagger = \beta \Rightarrow \mathsf{list}\ {\alpha_0}^\dagger\ \mathsf{j}$$

which encodes to $p_4 \sqcap q_5 \leq p_{10}$.

Finally we algebraically calculate:

$$\begin{array}{ll} & p_1 \cdot c_1 \cdot c_6 \cdot c_7 \\ \leq & \quad \{\![\text{ assumption (1) }]\!\} \\ & c_1 \cdot (p_1 \sqcap s_1) \cdot c_6 \cdot c_7 \\ = & \quad \{\![\ \mathsf{emp} \text{ neutral w.r.t. } * \ ]\!\} \\ & c_1 \cdot ((p_1 * \mathsf{emp}) \sqcap s_1) \cdot c_6 \cdot c_7 \\ = & \quad \{\![\ s_1 \text{ is pure, Lemma 3.5.6 }]\!\} \\ & c_1 \cdot (p_1 * (\mathsf{emp} \sqcap s_1)) \cdot c_6 \cdot c_7 \\ = & \quad \{\![\text{ Definition 5.1.1 }]\!\} \\ & c_1 \cdot (p_1 * p_2) \cdot c_6 \cdot c_7 \\ \leq & \quad \{\![\text{ Assumption (2) }]\!\} \\ & c_1 \cdot c_6 \cdot ((p_1 * p_2) \sqcap q_6) \cdot c_7 \\ \leq & \quad \{\![\text{ Assumption (2) }]\!\} \\ & c_1 \cdot c_6 \cdot c_7 \cdot ((p_1 * p_2) \sqcap q_6 \sqcap q_7) \\ \leq & \quad \{\![\text{ Assumption (3) }]\!\} \\ & c_1 \cdot c_6 \cdot c_7 \cdot ((p_3 * p_4) \sqcap q_1) \end{array}$$

Next we prove correctness of the while loop. This is done by using the algebraically encoded inference rule for the while loop:

$$p \cdot i \cdot c \leq c \cdot i \Rightarrow i \cdot (p \cdot c)^* \cdot \neg p \leq (p \cdot c)^* \cdot \neg p \cdot i$$

where i denotes the loop invariant. By instantiating this law we have to prove

$$\begin{array}{ll} & \neg s_2 \cdot ((p_3 * p_4) \sqcap q_1) \cdot c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \\ \leq & c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \cdot ((p_3 * p_4) \sqcap q_1). \end{array}$$

Again we calculate

$$
\begin{aligned}
& \neg s_2 \cdot ((p_3 \,*\, p_4) \sqcap q_1) \cdot c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \\
= \quad & \{\![\sqcap \text{ coincides with } \cdot \text{ on tests }]\!\} \\
& (\neg s_2 \sqcap ((p_3 \,*\, p_4) \sqcap q_1)) \cdot c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \\
= \quad & \{\![\neg s_2 \text{ is pure }]\!\} \\
& (((\neg s_2 \sqcap p_3) \,*\, p_4) \sqcap q_1) \cdot c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \\
= \quad & \{\![\text{Assumption (4)}]\!\} \\
& (((\neg q_4 \sqcap p_3) \,*\, p_4) \sqcap q_1) \cdot c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \\
\leq \quad & \{\![\text{Assignment, frame rule}]\!\} \\
& c_8 \cdot (((\neg q_4 \sqcap p_3) \,*\, p_4) \sqcap q_1 \sqcap q_8) \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \\
\leq \quad & \{\![\text{Assignment, frame rule}]\!\} \\
& c_8 \cdot c_9 \cdot (((\neg q_4 \sqcap p_3) \,*\, p_4) \sqcap q_1 \sqcap q_8 \sqcap q_9) \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \\
\leq \quad & \{\![\text{Assumption (5), idempotency of } \sqcap\text{, isotony, } q_2 \text{ is pure}]\!\} \\
& c_8 \cdot c_9 \cdot ((p_{11} \,*\, p_4) \sqcap q_3 \sqcap q_2) \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \\
\leq \quad & \{\![\text{isotony}]\!\} \\
& c_8 \cdot c_9 \cdot ((p_{11} \,*\, p_4) \sqcap q_3) \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \\
\leq \quad & \{\![q_3 \text{ is pure}]\!\} \\
& c_8 \cdot c_9 \cdot (p_{11} \,*\, (p_4 \sqcap q_3)) \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \\
\leq \quad & \{\![\text{Assumption (6), frame rule}]\!\} \\
& c_8 \cdot c_9 \cdot c_{10} \cdot (r_1 \,*\, r_2 \,*\, p_5 \,*\, (p_4 \sqcap q_3)) \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \\
\leq \quad & \{\![\text{Assumption (7) with } p = r_1 \,*\, p_5 \,*\, (p_4 \sqcap q_3),\ p' = r_1 \,*\, p_6 \,*\, (p_4 \sqcap q_3)]\!\} \\
& c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot (r_1 \,*\, r_4 \,*\, p_6 \,*\, (p_4 \sqcap q_3)) \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \\
\leq \quad & \{\![\text{Assumption (8) with } p = p' = r_1 \,*\, p_6 \,*\, (p_4 \sqcap q_3)]\!\} \\
& c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot (r_1 \,*\, r_3 \,*\, p_6 \,*\, (p_4 \sqcap q_3)) \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \\
\leq \quad & \{\![q_3 \text{ is pure}]\!\} \\
& c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot ((r_1 \,*\, r_3 \,*\, p_6 \,*\, p_4) \sqcap q_3) \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \\
\leq \quad & \{\![\text{Assumption (9), commuativity of } *]\!\} \\
& c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot ((p_6 \,*\, p_7) \sqcap q_3) \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \\
\leq \quad & \{\![\text{Assignment rule}]\!\} \\
& c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot ((p_6 \,*\, p_8) \sqcap q_{10}) \cdot c_{12} \cdot c_4 \cdot c_5 \\
\leq \quad & \{\![\text{Assignment rule}]\!\} \\
& c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot ((p_{12} \,*\, p_8 \sqcap q_1) \cdot c_4 \cdot c_5 \\
\leq \quad & \{\![\text{Assignment rule}]\!\} \\
& c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot ((p_{12} \,*\, p_4) \sqcap q_1) \cdot c_5 \\
\leq \quad & \{\![\text{Assignment rule}]\!\} \\
& c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5 \cdot ((p_3 \,*\, p_4) \sqcap q_1)
\end{aligned}
$$

The last part of the correctness proof shows that the algorithm in fact returns the reverted list at the end.

$$
\begin{aligned}
& s_2 \cdot ((p_3 \,*\, p_4) \sqcap q_1) \\
= \quad & \{\![\cdot \text{ coincides with } \sqcap \text{ on tests}]\!\} \\
& s_2 \sqcap ((p_3 \,*\, p_4) \sqcap q_1) \\
= \quad & \{\![s_2,\ q_1 \text{ are pure}]\!\} \\
& (s_2 \sqcap p_3 \sqcap q_1) \,*\, p_4 \\
\leq \quad & \{\![\text{Assumption (10)}]\!\} \\
& (\mathsf{emp} \sqcap q_4 \sqcap q_1) \,*\, p_4 \\
\leq \quad & \{\![\text{Assumption (11)}]\!\} \\
& (\mathsf{emp} \sqcap q_5) \,*\, p_4 \\
= \quad & \{\![q_5 \text{ is pure, } \mathsf{emp} \text{ is neutral w.r.t. } *]\!\}
\end{aligned}
$$

$$\begin{aligned} & p_4 \sqcap q_5 \\ \leq \quad & \{\!\![\text{ Assumption (11) }]\!\!\} \\ & p_{10} \end{aligned}$$

Finally we can use all three proved parts above and easily show

$$\begin{aligned} & p_1 \cdot c_1 \cdot c_6 \cdot c_7 \cdot (\neg s_2 \cdot c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5)^* \cdot s_2 \\ \leq \; & c_1 \cdot c_6 \cdot c_7 \cdot (\neg s_2 \cdot c_8 \cdot c_9 \cdot c_{10} \cdot c_2 \cdot c_3 \cdot c_{11} \cdot c_{12} \cdot c_4 \cdot c_5)^* \cdot p_{10}. \end{aligned}$$

Hence, at the end of this algorithm, we get the reversed list on the heap with the pointer j saving the initial address of the list. One can see that in the verification in separation logic above a lot of steps have been left out. Also the quantification over the four ghost variables even makes the verification more difficult to read. In our algebraic approach we treated the ghost variables as normal store variables but kept them disjoint from the usual program store variables to ensure that they are not used by the program itself. This gives us the freedom to calculate with them in states as we had done with the usual store variables and also makes the semantics of the verification proofs easier to understand. Compared with the separation logic proof, the algebraic approach is more precise and exactly points out what is needed for every step in the verification. By this our proof also becomes more lengthy than the original, not fully formal verification in separation logic. However since this algebraic approach is completely first-order based fully automated theorem provers can be applied for the presented verification task. Only the given assumptions use domain-specific knowledge which has to be considered. One approach to handle this could be an integration of SMT-solvers for verifications at this level since these systems support arithmetics. But this is left for future research.

Chapter 6

Conclusion and Outlook

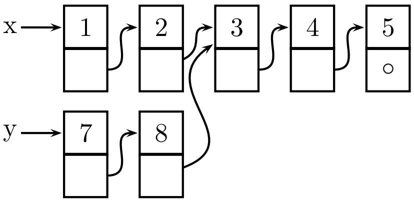
We have presented an algebraic treatment of separation logic. In particular, we introduced a model for the assertions which is based on sets of states like it has been done for temporal logics ([DMS04, MHS06]). Furthermore we defined an algebraic version of the separating conjunction and were able to embed the assertion part of the logic into a Boolean quantale. Using this approach we identified the separating implication with algebraic residuals. A lot of work ([Möl05]) has already been done on composing laws for these operators. We figured out that all of the derived inference rules of [Rey08a] can be proved algebraically very easily and are just simple consequences of most of the laws in [Möl05].

We got another result from this algebraic approach on a closer look to the mentioned classes of assertions. For pure assertions we found an axiomatization which made it possible for us to prove all laws in [Rey08a]. Furthermore we have found a lot of useful consequences for pure assertions. Another class of assertions we have focused on was the class of precise assertions. But for this class and most other given in [Rey08b] we have not found a characterization so far and will have to investigate further in future work.

The next step we took was an embedding of the command language of this logic into an algebraic structure. We used a relational view for the commands as it has already been done for Hoare logic. We defined this relational view for the new heap-dependent commands and explained a lot of the inference rules for them given in [Rey08a]. To use the assertion quantale with commands, we lifted the set-based semantics of our assertions up to sets of tuples of states and linked the assertion and command quantales by the frame rule. As related work we could try to find another approach which might be better than the relational view on the command language.

Finally we algebraically verified the standard example of an in-place list reversal algorithm with the algebraic structure we presented so far. The name ”in-place” means that there is no copying of whole structures, i.e., the reversal is done by simple pointer modifications. As a result of a unique treatment of store and ghost variables our proof was more precise than the one given in [Rey08a]. For this proof we needed to sum up a few laws for the list predicate to get some proof steps under control. But this approach is still quite promising and we might get more general inference rules for our new commands by analyzing another examples.

In this report we have not analyzed situations where data structures may share some parts of their cells. Consider a situation where we got two lists like in the following figure:



The cells with contents 3, 4 and 5 are list cells of both lists. Of course, if one dereferences one of the two lists and changes these cells, it will affect both lists. There are already some works published for this topic with an algebraic approach ([Möl99, Ehm04]). These have to be analyzed in a more detailed way using our algebraic approach for separation logic.

Appendix A - Prover9 Input File

As mentioned at the beginning of this report we give in the following an input file for the *Prover9* proofs which we used for a verification of our results.

```
op(500, infix, "+").   %addition
op(490, infix, ";").   %meet
op(490, infix, "*").   %multiplication
op(490, infix, "\").   %left residual
op(480, infix, "-->"). %implication

formulas(assumptions).
  %commutative additive monoid
    x+y = y+x.
    x+0 = x.
    x+(y+z) = (x+y)+z.
  %commutative multiplicative monoid
    x*1 = x.
    x*y = y*x.
    x*(y*z) = (x*y)*z.
  %annihilation
    0*x = 0.
  %addition is idempotent
    x+x = x.
  %distributivty of * w.r.t. +
    (x+y)*z = x*z + y*z.
  %greatest element T
    x <= T.
  %definition of residuals
    x <= y\z <-> y*x <= z.
  %natural order
    x <= y <-> x+y = y.
  %definition of ; (meet)
    x <= y;z <-> (x <= y & x <= z).
  %distributivity of ; w.r.t. +
    (x+y);z = x;z + y;z.
  %definition of implication
    all z (z <= x --> y <-> z;x <= y).
  %definition of pure elements
    pure(x) <-> (x*T <= x & all y all z (y*z ; x <= (y;x)*(z;x))).
end_of_list.
```

Some of the goals need some additional lemmas which we give in the following.

```
formulas(assumptions).
 %isotony of multiplication
    x <= y -> z*x <= z*y.
    x <= y -> x*z <= y*z.
 %isotony of meet
    x <= y -> x;z <= y;z.
    x <= y -> z;x <= z;y.
 %definition of meet with negation (Boolean quantale)
    %x = c(c(x) + c(y))+c(c(x)+y).
    %x;y = c(c(x) + c(y)). %only used for Corollary 3.5.15
end_of_list.
```

In the following we list some of the laws which we have succesfully verified.

```
formulas(goals).
 %Lemma 3.4.7
    %y*(y\x) <= x.
 %Lemma 3.4.8
    %y*(y\x) <= y*T ; x.
 %Lemma 3.4.9
    %x <= y\(y*x).
 %Lemma 3.4.11
    %all x1 all x2 ((x1 <= (x\y) & x2 <= (y\z)) -> x1*x2 <= x\z).
 %Lemma 3.4.12
    %all x1 all y1 ((x <= x1 & y <= y1) -> x1\y <= x\y1).

 %Corollary 3.5.4
    %pure(x) -> x = (x;1)*T.
 %Corollary 3.5.5
    %pure(x) -> (all y x*y = x;(y*T)).
 %Lemma 3.5.6
    %pure(x) -> (y;x)*z = (y*z);x.
 %Lemma 3.5.7
    %pure(x) -> x;y <= x*y.
 %Lemma 3.5.8
    %pure(x) & pure(y) -> x*y <= x;y.
 %Corollary 3.5.10
    %pure(x) -> x*x = x.
 %Lemma 3.5.12
    %pure(x) -> x\y <= x --> y.
 %Lemma 3.5.13
    pure(x) & pure(y) -> (x --> y <= x\y).
 %Corollary 3.5.15
    %pure(x) & (all y x*y = x;(y*T)) -> x*c(x) = 0.

end_of_list.
```

Appendix B - Translation Table

encoded	algebraically	in separation logic
Lemma 3.4.7	$q \cdot (q \backslash p) \leq p$	$\dfrac{}{q * (q \mathrel{-\!\!*} p) \Rightarrow p}$
Lemma 3.4.8	$q \cdot (q \backslash p) \leq (q \cdot \top) \sqcap p$	$\dfrac{}{q * (q \mathrel{-\!\!*} p) \Rightarrow ((q * \mathsf{true}) \wedge p)}$
Lemma 3.4.9	$p \leq q \backslash (q \cdot p)$	$\dfrac{}{p \Rightarrow (q \mathrel{-\!\!*} (q * p))}$
Lemma 3.4.10	$p \cdot r \leq p \cdot (q \backslash (q \cdot r))$	$\dfrac{}{(p * r) \Rightarrow (p * (q \mathrel{-\!\!*} (q * r)))}$
Lemma 3.4.11	$p_0 \leq q \backslash r \wedge p_1 \leq r \backslash s$ $\Rightarrow p_1 \cdot p_0 \leq q \backslash s$	$\dfrac{p_0 \Rightarrow (q \mathrel{-\!\!*} r) \quad p_1 \Rightarrow (r \mathrel{-\!\!*} s)}{(p_1 * p_0) \Rightarrow (q \mathrel{-\!\!*} s)}$
Lemma 3.4.12	$p_0 \leq p_1 \wedge q_0 \leq q_1$ $\Rightarrow p_1 \backslash q_0 \leq p_0 \backslash q_1$	$\dfrac{p_0 \Rightarrow p_1 \quad q_0 \Rightarrow q_1}{(p_1 \mathrel{-\!\!*} q_0) \Rightarrow (p_0 \mathrel{-\!\!*} q_1)}$
Definition 3.5.2	$p \cdot \top \leq p$	$p * \mathsf{true} \Rightarrow p$
	$(q \cdot r) \sqcap p \leq (q \sqcap p) \cdot (r \sqcap p)$	$(q * r) \wedge p \Rightarrow (q \wedge p) * (r \wedge p)$
Corollary 3.5.4	$p = (p \sqcap 1) \cdot \top$	$p \Leftrightarrow (p \wedge \mathsf{emp}) * \mathsf{true}$
Corollary 3.5.5	$p \cdot q = p \sqcap (q \cdot \top)$	$p * q \Leftrightarrow p \wedge (q * \mathsf{true})$
Lemma 3.5.6	$(p \sqcap q) \cdot r = (p \cdot r) \wedge q$	$(p \wedge q) * r \Leftrightarrow (p * r) \wedge q$
Lemma 3.5.7	$p \sqcap q \leq p \cdot q$	$p \wedge q \Rightarrow p * q$
Lemma 3.5.8	$p \cdot q \leq p \sqcap q$	$p * q \Rightarrow p \wedge q$
Corollary 3.5.9	$p \cdot q = p \sqcap q$	$p * q \Leftrightarrow p \wedge q$
Corollary 3.5.10	$p \cdot p = p$	$p * p \Leftrightarrow p$
Lemma 3.5.11	$p + q$	$p \vee q$
	$p \sqcap q$	$p \wedge q$
	$p \cdot q$	$p * q$
Lemma 3.5.12	$p \backslash q \leq p \mathrel{-\!\!\triangleright} q$	$(p \mathrel{-\!\!*} q) \Rightarrow (p \rightarrow q)$
Lemma 3.5.13	$p \mathrel{-\!\!\triangleright} q \leq p \backslash q$	$(p \rightarrow q) \Rightarrow (p \mathrel{-\!\!*} q)$
Corollary 3.5.14	$\overline{p}$	$\neg p$
Corollary 3.5.15	$p \cdot \overline{p} = 0$	$p * \neg p \Leftrightarrow \mathsf{false}$
Definition 3.5.19	$i \cdot \top \leq i$	$i * \mathsf{true} \Rightarrow i$
Lemma 3.5.20	$i \cdot i' \leq i \sqcap i'$	$i * i' \Rightarrow i \wedge i'$

Bibliography

[Con71] Conway, J. H.: *Regular Algebra and Finite Machines.* Chapman and Hall, 1971.

[DH08] Dang, H.-H. and P. Höfner: *First-Order Theorem Prover Evaluation w.r.t. Relation- and Kleene algebra.* In Berghammer, R., B. Möller and G. Struth (editors): *Relations and Kleene Algebra in Computer Science — PhD Programme at RelMiCS 10/AKA 05*, number 2008-04 in *Technical Report*, pages 48–52. Institut für Informatik, Universität Augsburg, 2008.

[DMS04] Desharnais, J., B. Möller and G. Struth: *Modal Kleene algebra and Applications — A Survey.* Journal of Relational Methods in Computer Science, 1:93–131, 2004.

[Ehm04] Ehm, T.: *Pointer Kleene Algebra.* In Berghammer, R., B. Möller and G. Struth (editors): *RelMiCS*, volume 3051 of *Lecture Notes in Computer Science*, pages 99–111. Springer, 2004.

[HMSW09] Hoare, C.A.R., B. Möller, G. Struth and I. Wehrman: *Concurrent Kleene Algebra.* (to appear), 2009.

[Hun33] Huntington, E.V.: *Boolean algebra, A correction.* Trans. AMS 35, pages 557–558, 1933.

[HWW09] Hoare, C.A.R., I. Wehrman and O'Hearn P. W.: *Graphical Models of Separation Logic.* (to appear), 2009.

[Koz94] Kozen, D.: *A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events.* Information and Computation, 110(2):366–390, 1994.

[Koz97] Kozen, D.: *Kleene Algebra with Tests.* ACM Transactions on Programming Languages and Systems, 19(3):427–443, 1997.

[Koz00] Kozen, D.: *On Hoare Logic and Kleene Algebra with Tests.* ACM Transactions on Computational Logic, 1(1):60–76, 2000.

[MHS06] Möller, B., P. Höfner and G. Struth: *Quantales and Temporal Logics.* In Johnson, M. and V. Vene (editors): *Algebraic Methodology and Software Technology*, volume 4019 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2006.

[Möl93] Möller, B.: *Towards Pointer Algebra.* Science of Computer Programming, 21(1):57–90, 1993.

[Möl05] Möller, B.: *Residuals and Detachments.* Technical Report, Department for Computer Science, December 2005.

[Mor98] Morgan, C.: *Programming from Specifications.* Prentice-Hall, Inc, Second edition, 1998.

[MS06] Möller, B. and G. Struth: *Algebras of Modal Operators and Partial Correctness.* Theoretical Computer Science, 351(2):221–239, 2006.

[Mul86] Mulvey, C. J.: &. In *Second Topology Conference*, Rendiconti del Circolo Matematico di Palermo 12, pages 99–104, 1986.

[Möl99] Möller, B.: *Calculating with Acyclic and Cyclic Lists.* Information Sciences — An International Journal., 119(3-4):135–154, 1999.

[ORY01] O'Hearn, P. W., J. C. Reynolds and H. Yang: *Local Reasoning about Programs that Alter Data Structures.* In Fribourg, L. (editor): *CSL '01: Proceedings of the 15th International Workshop on Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[Rey08a] Reynolds, J. C.: *An Introduction to Separation Logic*, June 2008.

[Rey08b] Reynolds, J. C.: *An Introduction to Separation Logic (Preliminary Draft)*, October 2008.