Concurrency and local reasoning under reverse exchange

Han Hing Dang, Bernhard Möller

Angaben zur Veröffentlichung / Publication details:

Dang, Han Hing, and Bernhard Möller. 2014. "Concurrency and local reasoning under reverse exchange." *Science of Computer Programming* 85: 204–23. https://doi.org/10.1016/j.scico.2013.07.006.

Concurrency and Local Reasoning Under Reverse Exchange

H.-H. Dang[a], B. Möller[a]

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

Abstract

Quite a number of aspects of concurrency are reflected by the inequational exchange law $(P * Q) ; (R * S) \leq (P ; R) * (Q ; S)$ between sequential composition ; and concurrent composition $*$. In particular, recent research has shown that, under a certain semantic definition, validity of this law is equivalent to that of the familiar concurrency rule for Hoare triples. Unfortunately, while the law holds in the standard model of concurrent Kleene algebra, its is not true in the relationally based setting of algebraic separation logic. However, we show that under mild conditions the reverse inequation $(P ; R) * (Q ; S) \leq (P * Q) ; (R * S)$ still holds there. From this reverse exchange law we derive slightly restricted but still reasonably useful variants of the concurrency rule. Moreover, using a corresponding definition of locality, we obtain also a variant of the frame rule, where $*$ now is interpreted as separating conjunction. These results allow using the relational setting also for modular and concurrency reasoning. Finally, we interpret the results further by discussing several variations of the approach.

Keywords: True concurrency, relational semantics, Hoare logic, concurrent separation logic, locality

1. Introduction

Over the recent years, logical techniques in program semantics have been supplemented by algebraic approaches which frequently allow more concise and perspicuous reasoning. The present paper extends one particular approach in that area, viz. *Algebraic Separation Logic* [2]. That framework was developed to reflect *separation logic (SL)* [3]. Although SL originally was developed to facilitate reasoning about shared mutable data structures, it has proved to be also very effective for modular reasoning about concurrency [4, 5]. For this logic there are already several abstract approaches that capture corresponding calculi, e.g., [6]. A more comprehensive general algebraic structure is provided by *Concurrent Kleene Algebra (CKA)* [7]. A central concept of that algebra is that it allows easy soundness proofs of important rules like the concurrency and frame rules used in logics for concurrency and modular reasoning.

The *concurrency* and *frame* rules have the form

$$\frac{\{P_1\}\, Q_1 \,\{R_1\} \qquad \{P_2\}\, Q_2 \,\{R_2\}}{\{P_1 * P_2\}\, Q_1 * Q_2 \,\{R_1 * R_2\}}\ (\mathit{conc}) \qquad \frac{\{P\}\, Q \,\{R\}}{\{P * S\}\, Q \,\{R * S\}}\ (\mathit{frame})\ .$$

Here Q and Q_i denote programs while all other letters denote assertions. The essential feature of these rules is the *separating conjunction* $*$ which stands for non-interfering concurrency or disjointness of resources. Hence these rules express that one may reason in a modular way about program parts when the context does not interfere with them.

Interestingly, the recent paper [8] shows that validity of the concurrency rule using the triple interpretation $\{P\}\, Q \,\{R\} \Leftrightarrow_{df} P ; Q \subseteq R$ is equivalent to validity of the *exchange law*

$$(P_1 * P_2) ; (Q_1 * Q_2) \leq (P_1 ; Q_1) * (P_2 ; Q_2)\ ,$$

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

for programs P_i and Q_i. Likewise, validity of the frame rule is equivalent to validity of the *small exchange law*

$$(P_1 * P_2)\,;Q_1 \leq (P_1\,;Q_1) * P_2$$

In these laws, semicolon denotes sequential composition, while $\leq$ denotes a partial ordering expressing refinement. The exchange laws abstractly characterise the interplay between sequential and concurrent composition. Each of them expresses that the program on the right-hand side has fewer sequential dependences than the one on the left-hand side.

There are several algebraic models satisfying those laws:

- The standard model is based on sets of traces. This model is defined very abstractly so that it needs to be refined further to model concurrency with concrete programs adequately enough. However, it enables elegant and simple proofs.
- Another model employs predicate transformers to abstractly capture program behaviour of separation logic. It validates a certain part of the CKA laws, in particular the exchange law. But it fails to satisfy other important laws needed for program proofs as, e.g., laws in connection with nondeterministic choice.

More details may be found in [7, 8].

The purpose of the present paper is to investigate relationally based Algebraic Separation Logic mentioned above with respect to exchange laws. As a relational structure it copes well with nondeterminacy; moreover, it allows the re-use of a large and well studied body of algebraic laws in connection with assertion logic. Surprisingly, although the model satisfies neither of the mentioned exchange laws, it validates an exchange law with the reversed refinement order. This entails variants of the concurrency and frame rules with similarly simple soundness proofs as in the original Concurrent Kleene Algebra approach. Additionally, we establish an equivalence between the concurrency rule and the reverse exchange law analogous to the one in [8]. This shows that the relational model can be applied in reasoning about programs that involve true concurrency and modularity. To underpin this further, we also study a number of variations of our main relational model and discuss their adequacy and usefulness.

2. Basic Definitions and Properties

We start by repeating some basic definitions from [2] and some direct consequences. Summarised, the central concept of this paper is a relational structure enriched by an operator that ensures disjointness of program states or executions. Notationally, we follow [2, 8].

Definition 2.1. A *separation algebra* is a partial commutative monoid $(\Sigma, \bullet, u)$; the elements of Σ are called *states* and denoted by $\sigma, \tau, \ldots$. The operator $\bullet$ denotes state combination and the *empty state* u is its unit. A partial commutative monoid is given by a partial binary operation satisfying the unity, commutativity and associativity laws w.r.t. the equality that holds for two terms iff both are defined and equal or both are undefined. The induced *combinability* or *disjointness* relation $\#$ is defined by

$$\sigma_0 \# \sigma_1 \;\Leftrightarrow_{df}\; \sigma_0 \bullet \sigma_1 \text{ is defined .}$$

As a concrete example one can instantiate the states to heaps. For this one has $\Sigma = \mathbb{N} \rightharpoonup \mathbb{N}$, i.e., the set of partial functions from naturals to naturals. Moreover $\bullet$ is the union of domain-disjoint heaps and $u = \emptyset$, the empty heap. The corresponding combinability relation is $h_0 \# h_1 \Leftrightarrow dom(h_0) \cap dom(h_1) = \emptyset$ for heaps h_0, h_1. More concrete examples can be found in [6].

Definition 2.2. We assume a separation algebra $(\Sigma, \bullet, u)$. A *command* is a relation $P \subseteq \Sigma \times \Sigma$ between states. Relational composition is denoted by $;$. The command skip is the identity relation between states. A *test* is a sub-identity, i.e., a command P with $P \subseteq \mathsf{skip}$. In the remainder we will denote tests by lower case letters $p, q, \ldots$. A particular test that characterises the empty state u is provided by $\mathsf{emp} =_{df} \{(u, u)\}$.

Moreover, the domain of a command P, represented as a test, will be denoted by ΔP. It is characterised by the universal property

$$\Delta P \subseteq q \;\Leftrightarrow\; P \subseteq q\,;P\ .$$

In particular, $P \subseteq \Delta P\,;P$ and hence $P = \Delta P\,;P$.

Note that tests form a Boolean algebra with skip as its greatest and $\emptyset$ as its least element. Moreover, on tests $\cup$ coincides with join and ; with meet. In particular, tests are idempotent and commute under composition, i.e., $p\,;p = p$ and $p\,;q = q\,;p$.

With these definitions, faulting and divergence of commands are identified, i.e., they are both modeled by the empty command $\emptyset$. We provide some examples of commands over the above concrete separation algebra on heaps.

$$\begin{array}{lll} [x] := n & =_{df} & \{(h, \{(x,n)\}|h) : x \in dom(h)\}\ , \\ \mathsf{malloc}\,(x,n) & =_{df} & \{(h, \{(x,0),...,(x+n-1,0)\} \cup h) : \{x,...,x+n-1\} \not\subseteq dom(h), n \in \mathbb{N}\}\ , \\ \mathsf{delete}\,(x) & =_{df} & \{(h, h - \{(x,n)\}) : (x,n) \in h\}\ , \end{array} \qquad (1)$$

where $|$ denotes function update. The first command changes the content of the heap cell x to the value n. The second command allocates n contiguous heap cells starting from the address x, while the third one deletes the cell x and its content from the heap. As a further command we define $(x \mapsto y) =_{df} \{(\{(x,y)\}, \{(x,y)\})\}$. It tests whether the current heap consists of a single cell with address x and content y. If so, it leaves the heap unchanged; otherwise it evaluates to the empty command $\emptyset$. Finally we can define a while in the standard relational way by

$$\mathsf{while}\ P\ \mathsf{do}\ C\ =_{df}\ (P\,;C)^*\,;\neg P\ ,$$

where P is a test, $\neg P$ its complement relative to skip and D^* is the reflexive-transitive closure of command D. Then we have the following examples:

$$\begin{array}{lcl} (x \mapsto 1)\,;\mathsf{delete}\,(x) & = & \{(\{(x,1)\}, \emptyset) : x \in \mathbb{N}\}\,, \\ \mathsf{delete}\,(x)\,;\mathsf{delete}\,(x) & = & \emptyset\,, \\ (2 \mapsto 3)\,;[5] := 6 & = & \emptyset\,, \\ \mathsf{while\ true\ do\ skip} & = & (\mathsf{true}\,;\mathsf{skip})^*\,;\neg\mathsf{true} \quad = \quad (\mathsf{skip}\,;\mathsf{skip})^*\,;\neg\mathsf{skip} \quad = \quad \emptyset\,. \end{array}$$

The latter holds since true coincides with skip.

Let us briefly sketch how this denotational semantics could be supplemented by an operational one (e.g. [4, 9]). The transition rules employed have the format $\sigma \xrightarrow{C} \sigma'$, where C is a command and σ, σ' are states in the separation algebra under consideration. The meaning of such a rule is that C may transform σ into σ'.

For those atomic commands that are defined over the concrete separation algebra of heaps the transition rules look as follows:

$$\frac{x \in dom(h)}{h \xrightarrow{[x]:=n} \{(x,n)\}|h} \qquad \frac{\{x,...,x+n-1\} \not\subseteq dom(h), n \in \mathbb{N}}{h \xrightarrow{\mathsf{malloc}\,(x,n)} \{(x,0),...,(x+n-1,0)\} \cup h} \qquad \frac{(x,n) \in h}{h \xrightarrow{\mathsf{delete}\,(x)} h - \{(x,n)\}}$$

For the other commands we have the standard rules which can be formulated for general states. For that we use the abbreviation $W =_{df} \mathsf{while}\ P\ \mathsf{do}\ C$:

$$\frac{}{\sigma \xrightarrow{\mathsf{skip}} \sigma} \qquad \frac{\sigma \xrightarrow{C} \sigma' \quad \sigma' \xrightarrow{C'} \sigma''}{\sigma \xrightarrow{C;C'} \sigma''} \qquad \frac{(\sigma,\sigma) \in P \quad \sigma \xrightarrow{C;W} \sigma'}{\sigma \xrightarrow{W} \sigma'} \qquad \frac{(\sigma,\sigma) \notin P}{\sigma \xrightarrow{W} \sigma}$$

A straightforward induction on the structure of a command C shows the correctness of these rules, viz. that $\sigma \xrightarrow{C} \sigma' \Rightarrow (\sigma,\sigma') \in C$. We forego a discussion of completeness, since this would inflate the paper too much.

Next we give some definitions to introduce separation relationally. Separating conjunction of commands can be interpreted as their parallel execution on disjoint or combinable portions of the state or, in the special case of tests, by asserting disjointness of certain resources.

Conceptually, the idea is to denote and manipulate splits of executions or resources by a lift to pairs of relations. This allows calculational proofs in the usual way, since we stay in a relational setting, but one can treat the parts independently.

Definition 2.3. We will frequently work with pairs of commands. Union, inclusion and composition of such pairs are defined componentwise. The *Cartesian product* $P \times Q$ of commands P, Q is given by

$$(\sigma_1, \sigma_2)\ (P \times Q)\ (\tau_1, \tau_2) \Leftrightarrow_{df} \ \sigma_1\ P\ \tau_1 \wedge \sigma_2\ Q\ \tau_2\ .$$

We assume that ; binds tighter than $\times$ and $\cap$. It is clear that $\mathsf{id} =_{df} \mathsf{skip} \times \mathsf{skip}$ is the identity of composition. Note that $\times$ and ; satisfy an *equational* exchange law:

$$P\,;Q \ \times\ R\,;S \ =\ (P \times R)\ ;\ (Q \times S)\ . \tag{2}$$

Definition 2.4. *Tests* in the set of product relations are again sub-identities ; as before they are idempotent and commute under ;. The Cartesian product of tests is a test again. However, there are other tests, such as the *combinability check* # [2], on pairs of states:

$$(\sigma_1, \sigma_2)\ \#\ (\tau_1, \tau_2) \Leftrightarrow_{df} \ \sigma_1 \# \sigma_2 \wedge \sigma_1 = \tau_1 \wedge \sigma_2 = \tau_2\ .$$

Definition 2.5. We define *split* $\lhd$ and its converse *join* $\rhd$ as in [2] by

$$\sigma \lhd (\sigma_1, \sigma_2) \ \Leftrightarrow_{df} \ (\sigma_1, \sigma_2) \rhd \sigma \ \Leftrightarrow_{df} \ \sigma_1 \# \sigma_2 \wedge \sigma = \sigma_1 \bullet \sigma_2\ .$$

Lemma 2.6. *We have* $\# = \rhd\,; \lhd \cap \mathsf{id}$ *and hence* $\# \subseteq \rhd\,; \lhd$. *Moreover* $\#\,; \rhd = \rhd$ *and symmetrically* $\lhd\,; \# = \lhd$. *Finally,* $\lhd\,; \rhd = \mathsf{skip}$.

The proof can be found in the Appendix.

One might conjecture $\mathsf{id} \subseteq \rhd\,; \lhd$ at first. However, this is not true, since id also considers non-combinable pairs of states which are not related by $\rhd\,; \lhd$ (cf. Lemma 2.6), since # is built into the definitions of $\rhd$ and $\lhd$. We will see in the next section that this fact requires us to impose an additional compatibility condition on commands for proving soundness of the reverse exchange law, i.e., the exchange law with the inequation reversed.

Definition 2.7. Generalising [2], we define the *parallel composition (separating conjunction)* of commands as $P * Q =_{df} \ \lhd\,; (P \times Q)\,; \rhd$.

By this definition, the relation $\sigma\,(P * Q)\,\tau$ holds iff σ can be split as $\sigma = \sigma_1 \bullet \sigma_2$ with disjoint combinable parts σ_1, σ_2 on which P and Q can act and produce results τ_1, τ_2 that are again disjoint and combine to $\tau = \tau_1 \bullet \tau_2$. This reflects angelic behaviour in the sense that, whenever σ_1 and σ_2 are not combinable or disjoint, P and Q are prevented from starting, since these states are eliminated by the definition. The same happens if σ_1 and σ_2 are combinable but P and Q produce non-combinable output states τ_1 and τ_2.

Hence $P * Q$ may be viewed as a program that runs P and Q in a concurrent fashion as indivisible actions, at least conceptually. An actual implementation may still do this in an interleaved or even truly concurrent fashion, as long as non-interference is guaranteed. A corresponding rule for the operational semantics reads as follows:

$$\frac{\sigma \xrightarrow{C} \tau \quad \sigma' \xrightarrow{C'} \tau' \quad \sigma \# \sigma' \quad \tau \# \tau'}{\sigma \bullet \sigma' \xrightarrow{C * C'} \tau \bullet \tau'}$$

We note that for tests p, q the command $p * q$ is a test again. Moreover, $*$ is associative and commutative and emp is its unit. In addition, we have the following result.

Lemma 2.8. skip *is idempotent w.r.t.* $*$*, i.e.,* $\mathsf{skip} * \mathsf{skip} = \mathsf{skip}$.

Proof. We calculate, using the definitions and Lemma 2.6,

$$\mathsf{skip} * \mathsf{skip} \;=\; \lhd \,;\, (\mathsf{skip} \times \mathsf{skip}) \,;\, \rhd \;=\; \lhd \,;\, \mathsf{id} \,;\, \rhd \;=\; \lhd \,;\, \rhd \;=\; \mathsf{skip} \;.$$

□

Finally, there is the following interplay between $*$ and the domain operator.

Lemma 2.9. *For commands* P, Q *we have* $\Delta(P * Q) \subseteq \Delta P * \Delta Q$.

The proof can be found in the Appendix.

3. Compatibility and the Reverse Exchange Law

According to the general results in [8], soundness of the concurrency rule in the relational setting would follow immediately if the exchange law

$$(P * Q) \,;\, (R * S) \;\subseteq\; (P \,;\, R) * (Q \,;\, S)$$

with relational inclusion $\subseteq$ as the refinement order were to hold there.

However, as also shown in [8], we have

Lemma 3.1. *The exchange law implies* $\mathsf{skip} \subseteq \mathsf{emp}$.

On the other hand, by definition $\mathsf{emp} \subseteq \mathsf{skip}$, so that by antisymmetry skip and emp would be equal, a contradiction.

Example 3.2. For a concrete counterexample one can instantiate the exchange rule with

$$\begin{array}{cc} P = (1 \mapsto 2) = \{(\{(1,2)\},\{(1,2)\})\}, & Q = (2 \mapsto 3) = \{(\{(2,3)\},\{(2,3)\})\}, \\ R = ([2] := 4) = \{(h, \{(2,4)\} | h) : 2 \in \mathit{dom}(h)\}, & S = ([1] := 5) = \{(h, \{(1,5)\} | h) : 1 \in \mathit{dom}(h)\} \;. \end{array}$$

Now, we have $P * Q = \{(\{(1,2),(2,3)\},\{(1,2),(2,3)\})\}$ and $(\{(1,2),(2,3)\},\{(1,5),(2,4)\}) \in R * S$. Hence the command on the left side of the exchange rule wrt. $\subseteq$ is non-empty. Unfortunately, the the programs in the right hand side of the rule resolve to programs $P ; R = (1 \mapsto 2) ; [2] := 4$ and $Q ; S = (2 \mapsto 3) ; [1] := 5$ which by definition are equal to the empty command. Therefore the rule is violated, since the whole command on the right side is $\emptyset$. □

We conclude that the exchange law is not valid in the relational setting. Instead, and surprisingly, we were able to show soundness of a restricted variant of the exchange law with the reversed inclusion order. The proof uses a restriction on pairs (P, Q) of commands: when P and Q start from combinable pairs of input states they produce combinable pairs of output states, or the other way around. This is formalised as follows.

Definition 3.3. Commands P and Q are *forward compatible* iff

$$\# \,;\, (P \times Q) \;\subseteq\; (P \times Q) \,;\, \# \;.$$

Symmetrically P and Q are *backward compatible* iff $(P \times Q) \,;\, \# \subseteq \# \,;\, (P \times Q)$. Two commands are called *compatible* iff they are forward and backward compatible, i.e., $\# \,;\, (P \times Q) = (P \times Q) \,;\, \#$.

To get an intuition for the concept of forward compatible commands we recall our instantiation of states to heaps and the commands described in Section 2. Additionally we define

$$\mathsf{arb\_alloc}\,(x) \quad =_{df} \quad \{(h, \{(x,n)\} \cup h) : x \notin dom(h), n \in \mathbb{N}\} \tag{3}$$

which allocates a new heap cell at x with arbitrary contents. For illustration, consider heaps $h_1 = \{(1,1)\}$, $h_2 = \emptyset$ and $h_1' = h_2' = \{(1,2)\}$. Clearly, $h_1 \# h_2$ and $((h_1,h_2),(h_1',h_2')) \in \# \,;((\![1] := 2) \times \mathsf{arb\_alloc}\,(1))$.

But $h_1' \# h_2'$ does not hold and hence $((h_1,h_2),(h_1',h_2')) \notin ((\![1] := 2) \times \mathsf{arb\_alloc}\,(1)) \,;\#$.

By changing $\mathsf{arb\_alloc}\,(1)$ to $\mathsf{arb\_alloc}\,(i)$ with $i \neq 1$, one would end up with compatible commands. In that case, the two compatible commands would work on disjoint portions of the heap and hence ensure disjointness before and after their execution.

Now we are ready for the central result of this section. For forward or backward compatible commands we are able to prove soundness of a variant of the reverse exchange law using the inclusion order. Note that validity of the exchange law in [8] is proved for arbitrary predicate transformers. In the next section we will see that specialising validity of the reversed law to compatible commands does not impair proving concurrency and frame rules in later sections.

Theorem 3.4 (Reverse Exchange). *If P, Q are forward compatible or R, S are backward compatible then*

$$(P \,;R) * (Q \,;S) \;\subseteq\; (P * Q) \,;(R * S) \;.$$

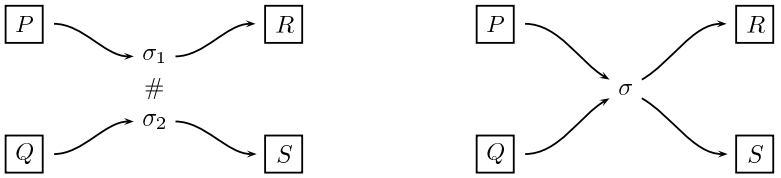
In particular, if P, Q or R, S are tests the inequation holds.

Proof. We assume that P and Q are forward compatible.

$$\begin{array}{ll}
 & (P \,;R) * (Q \,;S) \\
= & \quad \{\![\text{ definition of } * \,]\!\} \\
 & \lhd \,;(P \,;R \times Q \,;S) \,;\rhd \\
= & \quad \{\![\,;/\times \text{ exchange (2) }]\!\} \\
 & \lhd \,;(P \times Q) \,;(R \times S) \,;\rhd \\
= & \quad \{\![\text{ Lemma 2.6 }]\!\} \\
 & \lhd \,;\# \,;(P \times Q) \,;(R \times S) \,;\rhd \\
\subseteq & \quad \{\![\text{ forward compatibility }]\!\} \\
 & \lhd \,;(P \times Q) \,;\# \,;(R \times S) \,;\rhd \\
\subseteq & \quad \{\![\text{ Lemma 2.6 }]\!\} \\
 & \lhd \,;(P \times Q) \,;\rhd \,;\lhd \,;(R \times S) \,;\rhd \\
= & \quad \{\![\text{ definition of } * \,]\!\} \\
 & (P * Q) \,;(R * S) \;.
\end{array}$$

The proof for backward compatibility and R, S is symmetric. □

The reverse exchange law expresses an increase in granularity: while in the left-hand side programs $P \,;R$ and $Q \,;S$ are treated as indivisible, they are split in the right-hand side program, at the expense of a "global" synchronisation point marked by the semicolon.



The possibility of such a synchronisation point is established on the left-hand side by the compatibility requirement. I.e., in the above picture, the output states σ_1 of P and σ_2 of Q could be combined into a global state σ, as in the right-hand side program. In other words, the implicit split in the left hand side is one of the possible splits admitted by $\rhd ; \lhd$ in the right hand side. Still another way of viewing the rule is that the right-hand side "forgets" information about splits and therefore is more liberal.

As an example, we define programs *produce*(i) that produce some resource at address i and *consume*(i) that consume a resource at address i. Using heaps as states, these programs can be realised, e.g., by $\mathsf{arb\_alloc}\,(i)$ from Equation (3) and $\mathsf{delete}\,(i) =_{df} \{(h, h - \{(i,n)\}) : (i,n) \in h, n \in \mathbb{N}\}$. Next, consider for an $i \in \mathbb{N}$ the program

$$(\mathit{produce}(i)\,;\,\mathit{consume}(i)) * (\mathit{produce}(i+1)\,;\,\mathit{consume}(i+1))\ .$$

In this program, each producer and its corresponding consumer are treated together as an indivisible program. Since the producers and consumers work on disjoint resources, they are compatible and we can use the reverse exchange law to reorder the program above into

$$(\mathit{produce}(i) * \mathit{produce}(i+1))\,;\,(\mathit{consume}(i) * \mathit{consume}(i+1))\ .$$

This version represents a program where all resource allocations need to be executed concurrently before any of the resources can be consumed. The synchronisation point ; reflects a state that provides the two produced resources at i and $i+1$.

4. More on Compatibility

In this section we present further properties of the notion of compatibility given in Definition 3.3. For better readability, a few proofs and auxiliary results have been moved to the appendix at the end of the paper.

Lemma 4.1. *Assume P, Q are forward compatible. Then $\Delta P * \Delta Q = \Delta(P * Q)$, i.e., $*$ distributes over domain.*

A proof can be found in the Appendix.

Lemma 4.2. *All test commands are compatible with each other. In particular,* skip *is compatible with itself.*

Proof. For test commands p, q the relation $p \times q$ is a test in the algebra of relations on pairs. Since $\#$ is a test there, too, they commute, which means forward and backward compatibility of p and q. □

Since the combinability check $\#$ is a test on pairs of commands, it induces some useful closure properties.

Corollary 4.3. *If P, Q are forward compatible and $R \subseteq P$ then also R, Q are forward compatible. This result also holds for backward compatibility, hence compatibility is downward closed, too.*

Proof. We show the following more general result: Let C, D, E be relations on pairs of states such that C is a test. If C is an invariant of D, i.e., $C\,;D \subseteq D\,;C$, and $E \subseteq D$ then C is also an invariant of E. For this we calculate

$$\begin{aligned} C\,;E \;&=\; C\,;(D \cap E) \;=\; C\,;D \cap C\,;E \;\subseteq\; D\,;C \cap C\,;E \;= \\ &\quad D \cap C\,;E\,;C \;\subseteq\; C\,;E\,;C \;\subseteq\; E\,;C\ . \end{aligned}$$

The fourth step follows, since C is a test. A proof can, e.g., be found in [10]. Now the main claim follows by setting $C = \#$, $D = P \times Q$ and $E = R \times Q$. □

We note that this proof extends to arbitrary test semirings.

Corollary 4.4. *If P is forward/backward compatible with Q and R then it is also forward/backward compatible with $Q \cup R$.*

Proof. We show the case of forward compatibility.

$$\#\,;(P\times(Q\cup R)) \;=\; \#\,;((P\times Q)\cup(P\times R)) \;=\; \#\,;(P\times Q)\cup\#\,;(P\times R)\;\subseteq$$
$$(P\times Q)\,;\#\cup(P\times R)\,;\#\;=\;((P\times Q)\cup(P\times R))\,;\#\;=\;(P\times(Q\cup R))\,;\#\;.$$

□

Corollary 4.5.

1. *Let P, Q and R, S be forward compatible. Then also $P\,;R$ and $Q\,;S$ are forward compatible. Again the same holds for backward compatibility.*
2. *If P and Q are forward/backward compatible then so are P^n and Q^n for all $n\in\mathbb{N}$, where the n-th power of a command means n-fold sequential composition of the command with itself. Hence also $P^n * Q^n \;\subseteq\; (P*Q)^n$.*

Proof.

1. $\#\,;(P\,;R\times Q\,;S) = \#\,;(P\times Q)\,;(R\times S)\subseteq(P\times Q)\,;\#\,;(R\times S)\subseteq$
   $(P\times Q)\,;(R\times S)\,;\#=(P\,;R\times Q\,;S)\,;\#.$
2. Straightforward induction on n using Part 1. and reverse exchange. □

As a concrete example, Part 2 can be applied to `for` loops: if neither P nor Q change the loop counter `i` then

$$\texttt{(for (int i = 0 ; i < n; i++) }\{P\}) * \texttt{(for (int i = 0 ; i < n; i++) }\{Q\})$$
$$\subseteq\quad \texttt{for (int i = 0 ; i < n; i++) }\{P*Q\}\;.$$

5. Hoare Triples and the Concurrency Rule

To prepare our variant of the concurrency rule we now define Hoare triples in our setting.

Definition 5.1. For general commands P, Q, R, the *general Hoare triple* [8] is defined as

$$P\,\{\!|Q|\!\}\,R \;\Leftrightarrow_{df}\; P\,;Q\subseteq R\;.$$

For tests p, r and arbitrary command Q the *standard* Hoare triple [11] $\{p\}\,Q\,\{r\}$ is defined by

$$\{p\}\,Q\,\{r\} \;\Leftrightarrow_{df}\; p\,;Q\;\subseteq\;Q\,;r\;.$$

General Hoare triples also admit programs as assertions, in contrast to the standard ones that only allow tests to denote pre- and postconditions. As shown in [2], we have the relationship

$$\{p\}\,Q\,\{r\} \;\Leftrightarrow\; (U\,;p)\,\{\!|Q|\!\}\,(U\,;r)$$

where U denotes the universal relation. Hence our results for standard Hoare triples can be immediately translated into ones for general triples. The composition $U\,;p$ maps a test p to a command that makes no assumption about its starting state. Intuitively, starting from an arbitrary state that command will end up in one satisfying p. Trivially, a symmetrical command $p\,;U$ makes no restriction on the ending state or codomain.

In standard SL, the semantics of Hoare triples usually carries an enabledness condition $\Delta Q\subseteq P$ or $P\subseteq\Delta Q$ as additional conjunct. We decided not to follow this approach, since then we stay closer with standard relational semantics and also have more freedom: on different occasions we will need different enabling conditions (see the Concurrency Rules in Theorem 5.5 and Theorem 6.3 below). This would be ruled out using an SL-like definition.

Next we turn to the definition of properties and conditions that will allow us to prove variants of the concurrency rule for standard Hoare triples using the reverse exchange law.

The following observation is trivial, but useful for our first variant of the concurrency rule.

Lemma 5.2. $\{p\,;q\}\,Q\,\{r\} \;\Leftrightarrow\; \{p\}\,q\,;Q\,\{r\}$.

Proof. This is a straightforward calculation: By definition of standard Hoare triples, associativity of sequential composition, definition of standard Hoare triples,

$$\{p\,;q\}\,Q\,\{r\} \;\Leftrightarrow\; (p\,;q)\,;Q \subseteq Q\,;r \;\Leftrightarrow\; p\,;(q\,;Q) \subseteq Q\,;r \;\Leftrightarrow\; \{p\}\,q\,;Q\,\{r\}\ .$$

□

This lemma specialises in a number of ways. First, by the above remark, all tests p are idempotent, i.e., satisfy $p = p\,;p$. Hence, setting $q = p$ we obtain

Corollary 5.3. $\{p\}\,Q\,\{r\} \;\Leftrightarrow\; \{p\}\,p\,;Q\,\{r\}$.

The command $p\,;Q$ can be viewed as asserting the precondition p before executing Q. Next, we may set $p = \mathsf{skip} = \mathsf{true}$ in Lemma 5.2 to get

Corollary 5.4. $\{q\}\,Q\,\{r\} \;\Leftrightarrow\; \{\mathsf{true}\}\,q\,;Q\,\{r\}$.

The condition we need for our first variant of the concurrency rule is that the commands Q_i enforce the preconditions p_i in that all their starting states satisfy the respective p_i. Algebraically this is expressed by the formula $Q_i \subseteq p_i\,;Q_i$, which is equivalent to $Q_i = p_i\,;Q_i$ and to $\Delta Q_i \subseteq p_i$. This restriction is not essential: by Cor. 5.3 and the idempotence of tests we can always replace Q_i by $Q_i' =_{df} p_i\,;Q_i$ to achieve this.

Theorem 5.5 (Concurrency Rule I).

$$\frac{\{p_1\}\,Q_1\,\{r_1\} \quad \{p_2\}\,Q_2\,\{r_2\} \quad \Delta Q_1 \subseteq p_1 \quad \Delta Q_2 \subseteq p_2}{\{\mathsf{true}\}\,Q_1 * Q_2\,\{\,r_1 * r_2\}}$$

Proof.

$$\begin{array}{ll} & \mathsf{true}\,;(Q_1 * Q_2) \\ = & \quad \{\![\ \mathsf{true} \text{ is the identity }]\!\} \\ & Q_1 * Q_2 \\ \subseteq & \quad \{\![\ Q_i \subseteq p_i\,;Q_i\ ]\!\} \\ & (p_1\,;Q_1) * (p_2\,;Q_2) \\ \subseteq & \quad \{\![\ \text{by } \{p_i\}\,Q_i\,\{r_i\}\ ]\!\} \\ & (Q_1\,;r_1) * (Q_2\,;r_2) \\ \subseteq & \quad \{\![\ \text{reverse exchange law (Lemma 3.4), since } r_1 \text{ and } r_2 \text{ are tests} \\ & \quad \text{and hence compatible by Lemma 4.2 }]\!\} \\ & (Q_1 * Q_2)\,;(r_1 * r_2)\ . \end{array}$$

□

Note that compatibility of the commands Q_i is not needed.

We can bring the rule into a form closer to the original version:

Corollary 5.6. *Concurrency Rule I is equivalent to*

$$\frac{\{p_1\}\,Q_1\,\{r_1\} \quad \{p_2\}\,Q_2\,\{r_2\} \quad \Delta Q_1 \subseteq p_1 \quad \Delta Q_2 \subseteq p_2}{\{p_1 * p_2\}\,Q_1 * Q_2\,\{\,r_1 * r_2\}}$$

Proof. By the second proof step above, reverse exchange and $p_1 * p_2 \subseteq \mathsf{skip}$ we have

$$Q_1 * Q_2 \;\subseteq\; (p_1 \,;\, Q_1) * (p_2 \,;\, Q_2) \;\subseteq\; (p_1 * p_2) \,;\, (Q_1 * Q_2) \;\subseteq\; Q_1 * Q_2 \;.$$

Hence $Q_1 * Q_2 \;=\; (p_1 * p_2) \,;\, (Q_1 * Q_2)$, so that Corollary 5.4 shows the claim. □

A discussion of the relevance and use of this rule can be found in Section 8.

Next, we prove the following result which, together with Lemma 5.5, provides the analogue of the equivalence between the full exchange law and the concurrency rule shown in [8].

Lemma 5.7. *Validity of Concurrency Rule I implies a special case of the reverse exchange law: for arbitrary commands P_i and tests r_i,*

$$(P_1 \,;\, r_1) * (P_2 \,;\, r_2) \;\subseteq\; (P_1 * P_2) \,;\, (r_1 * r_2) \;.$$

Proof. In Concurrency Rule I we set $Q_i = P_i \,;\, r_i$ and $p_i = \Delta Q_i$. By this the premise of the rule becomes valid, since

$$\{\Delta Q_i\}\, Q_i \,\{r_i\} \;\;\Leftrightarrow\;\; \Delta Q_i \,;\, Q_i \;\subseteq\; Q_i \,;\, r_i \;\;\Leftrightarrow\;\; Q_i \;\subseteq\; Q_i \,;\, r_i$$

and $Q_i \,;\, r_i = (P_i \,;\, r_i) \,;\, r_i = P_i \,;\, r_i = Q_i$. Hence, by the conclusion of the rule we have

$$(\Delta Q_1 * \Delta Q_2) \,;\, (Q_1 * Q_2) \;\subseteq\; (Q_1 * Q_2) \,;\, (r_1 * r_2) \;. \qquad (\dagger)$$

Now we calculate:

$$\begin{array}{ll}
 & (P_1 \,;\, r_1) * (P_2 \,;\, r_2) \\
= & \quad \{\![\text{ definitions of } Q_i \text{ }]\!\} \\
 & Q_1 * Q_2 \\
= & \quad \{\![\text{ property of domain }]\!\} \\
 & \Delta(Q_1 * Q_2) \,;\, (Q_1 * Q_2) \\
\subseteq & \quad \{\![\text{ by Lemma 2.9 }]\!\} \\
 & (\Delta Q_1 * \Delta Q_2) \,;\, (Q_1 * Q_2) \\
\subseteq & \quad \{\![\text{ by } (\dagger) \text{ }]\!\} \\
 & (Q_1 * Q_2) \,;\, (r_1 * r_2) \\
= & \quad \{\![\text{ definitions of } Q_i \text{ }]\!\} \\
 & ((P_1 \,;\, r_1) * (P_2 \,;\, r_2)) \,;\, (r_1 * r_2) \\
\subseteq & \quad \{\![\text{ by } r_i \subseteq \mathsf{skip} \text{ }]\!\} \\
 & (P_1 * P_2) \,;\, (r_1 * r_2) \;.
\end{array}$$

□

We conclude by showing that the symmetric special case already follows without assuming reverse exchange or Concurrency Rule I or even mentioning the notion of compatibility.

Lemma 5.8. *For arbitrary commands Q_i and tests p_i,*

$$(p_1 \,;\, Q_1) * (p_2 \,;\, Q_2) \;\subseteq\; (p_1 * p_2) \,;\, (Q_1 * Q_2) \;.$$

Proof. We calculate:

$$\begin{array}{ll}
 & (p_1 \,;\, Q_1) * (p_2 \,;\, Q_2) \\
= & \quad \{\![\text{ property of domain }]\!\} \\
 & \Delta((p_1 \,;\, Q_1) * (p_2 \,;\, Q_2)) \,;\, ((p_1 \,;\, Q_1) * (p_2 \,;\, Q_2)) \\
\subseteq & \quad \{\![\text{ by Lemma 2.9 }]\!\} \\
 & (\Delta(p_1 \,;\, Q_1) * \Delta(p_2 \,;\, Q_2)) \,;\, ((p_1 \,;\, Q_1) * (p_2 \,;\, Q_2)) \\
= & \quad \{\![\text{ property of domain }]\!\}
\end{array}$$

$$
\begin{array}{ll}
 & ((p_1 \,;\, \Delta Q_1) * (p_2 \,;\, \Delta Q_2)) \,;\, ((p_1 \,;\, Q_1) * (p_2 \,;\, Q_2)) \\
\subseteq & \quad \{\![\text{ by } \Delta Q_i \subseteq \mathsf{skip} \text{ and } p_i \subseteq \mathsf{skip} \,]\!\} \\
 & (p_1 * p_2) \,;\, (Q_1 * Q_2) \;.
\end{array}
$$

□

Since Lemma 2.9 holds analogously for the codomain operator, this proof could also be adapted to a direct proof of the property in Lemma 5.7.

Finally, the special case of reverse exchange mentioned in Lemma 5.8 in turn implies Lemma 2.9:

$$
\begin{array}{ll}
 & \Delta(P * Q) \;\subseteq\; \Delta P * \Delta Q \\
\Leftrightarrow & \quad \{\![\text{ universal characterisation of domain }]\!\} \\
 & P * Q \;\subseteq\; (\Delta P * \Delta Q) \,;\, (P * Q) \\
\Leftarrow & \quad \{\![\text{ special case of reverse exchange }]\!\} \\
 & P * Q \;\subseteq\; (\Delta P \,;\, P) * (\Delta Q \,;\, Q) \\
\Leftrightarrow & \quad \{\![\text{ property of domain }]\!\} \\
 & P * Q \;\subseteq\; P * Q \\
\Leftrightarrow & \quad \{\![\text{ reflexivity of } \subseteq]\!\} \\
 & \mathsf{TRUE} \;.
\end{array}
$$

6. A Second Concurrency Rule

We now present a second variant of the concurrency rule. Its main idea is inspired by a more special property given in [2], which will also figure again in the next section.

Definition 6.1. Two commands Q_1, Q_2 have the *concurrency property* iff

$$(\Delta Q_1 \times \Delta Q_2) \,;\, \rhd \,;\, Q_1 * Q_2 \;\subseteq\; (Q_1 \times Q_2) \,;\, \rhd \;. \tag{4}$$

Note that by Lemma 2.6 this property is also equivalent to the equational form

$$(\Delta Q_1 \times \Delta Q_2) \,;\, \rhd \,;\, Q_1 * Q_2 \;=\; \# \,;\, (Q_1 \times Q_2) \,;\, \rhd \;.$$

The property reflects angelic behaviour in the following sense: whenever two combinable states σ_1 and σ_2 provide enough resources for the execution of the programs Q_i then each Q_i will be able to acquire its needed resource from the joined state $\sigma_1 \bullet \sigma_2$. Conceptually, this property can be seen as a fault-avoiding interpretation for the process of assigning a command those resources it needs. To see that this behaviour is not always guaranteed cf. Example 3.2. There, although the required resources are available, the program on the right side of the exchange law faults. We will elaborate on the meaning of the concurrency property in a more detailed discussion later.

We are now interested in relating the concurrency property to the exchange law for concurrent processes. It turns out that the property is sufficient for validating a special case of the exchange law which we use to prove soundness of our second concurrency rule.

Lemma 6.2. *Let commands Q_1 and Q_2 have the concurrency property and assume $p_i \subseteq \Delta Q_i$. Then the following weak version of the regular exchange law holds:*

$$(p_1 * p_2) \,;\, (Q_1 * Q_2) \;\subseteq\; (p_1 \,;\, Q_1) * (p_2 \,;\, Q_2) \;.$$

Proof.

$$
\begin{array}{ll}
 & (p_1 * p_2) \,;\, (Q_1 * Q_2) \\
= & \quad \{\![\text{ definition of } *,\; p_i \subseteq \Delta Q_i \text{ for } i = 1, 2 \,]\!\} \\
 & \lhd \,;\, (p_1 \,;\, \Delta Q_1 \times p_2 \,;\, \Delta Q_2) \,;\, \rhd \,;\, (Q_1 * Q_2)
\end{array}
$$

$$
\begin{array}{ll}
= & \quad \{\![\ ;/\times \text{ exchange (2) }]\!\} \\
& \lhd\,;(p_1 \times p_2)\,;(\Delta Q_1 \times \Delta Q_2)\,;\rhd\,;(Q_1 * Q_2) \\
\subseteq & \quad \{\![\ \text{concurrency property (4) }]\!\} \\
& \lhd\,;(p_1 \times p_2)\,;(Q_1 \times Q_2)\,;\rhd \\
= & \quad \{\![\ ;/\times \text{ exchange (2) }]\!\} \\
& \lhd\,;(p_1\,;Q_1 \times p_2\,;Q_2)\,;\rhd \\
= & \quad \{\![\ \text{definition of } *\ ]\!\} \\
& (p_1\,;Q_1) * (p_2\,;Q_2)\ .
\end{array}
$$

□

The premise expresses that resource p_i is accepted by command Q_i.

Interestingly, this special case of the exchange law already suffices to prove our second variant of the concurrency rule although the complete exchange law is needed for the concurrency rule in [8]; note also that the inclusion relations between the preconditions and the domains of the commands are the reverses of the ones in Lemma 5.5.

Theorem 6.3 (Concurrency Rule II). *Let Q_1 and Q_2 have the concurrency property. Then*

$$
\frac{\{p_1\}\,Q_1\,\{r_1\} \quad \{p_2\}\,Q_2\,\{r_2\} \quad p_1 \subseteq \Delta Q_1 \quad p_2 \subseteq \Delta Q_2}{\{p_1 * p_2\}\,Q_1 * Q_2\,\{r_1 * r_2\}}\ .
$$

Proof.

$$
\begin{array}{ll}
& (p_1 * p_2)\,;(Q_1 * Q_2) \\
\subseteq & \quad \{\![\ \text{Lemma 6.2 }]\!\} \\
& (p_1\,;Q_1) * (p_2\,;Q_2) \\
\subseteq & \quad \{\![\ \{p_i\}\,Q_i\,\{r_i\}\ ]\!\} \\
& (Q_1\,;r_1) * (Q_2\,;r_2) \\
\subseteq & \quad \{\![\ \text{reverse exchange law (Lemma 3.4), since } r_1, r_2 \text{ as tests are compatible }]\!\} \\
& (Q_1 * Q_2)\,;(r_1 * r_2)\ .
\end{array}
$$

□

An example for the application of this rule will be given in the next section.

Again we have a weak reverse implication.

Lemma 6.4. *Assume the validity of Concurrency Rule II. Then for arbitrary commands P_i and tests r_i,*

$$
(P_1\,;r_1) * (P_2\,;r_2) \ \subseteq\ (P_1 * P_2)\,;(r_1 * r_2)\ .
$$

Proof. The proof is verbatim the same as for Lemma 5.7. □

7. Concurrency and Preciseness

We now continue by a discussion about particular commands that satisfy the concurrency property, i.e., the premise of Concurrency Rule II.

First, to see that this property is not applicable to arbitrary commands, we present a concrete example, again in the heap model from Section 2, with two commands that do not satisfy the concurrency property. Consider

$$
Q_1 \ =_{df}\ ([1] := 1) \cup ([2] := 1) \quad \text{and} \quad Q_2 \ =_{df}\ ([1] := 2) \cup ([2] := 2)
$$

where $[x] := y$ represents a command that changes the content of the heap cell x to y (as defined in Equation (1)) and $\cup$ denotes non-deterministic choice. Clearly, the commands show interference with each other, since both may access the same heap locations.

To see that Q_1 and Q_2 do not satisfy the concurrency property, first note that $\Delta Q_1 = \Delta Q_2 = \{(h,h) : 1 \in dom(h) \vee 2 \in dom(h)\}$. Next, we consider heaps $h_1 = \{(1,0)\}$ and $h_2 = \{(2,0)\}$ with $(h_i, h_i) \in \Delta Q_i$. Thus, using $h = h_1 \bullet h_2$ and $h_1 \# h_2$, we have $(h,h) \in \Delta(Q_1 * Q_2)$. Moreover, a possible execution of $Q_1 * Q_2$ is $(h, \{(1,2),(2,1)\})$. Hence, $((h_1,h_2), \{(1,2),(2,1)\})$ is included in the left-hand side of the instantiated concurrency property but not in the right-hand side, since we only have $((h_1,h_2), \{(1,1),(2,2)\})$ there.

A closer look at the commands above shows that non-determinism in combination with commands working on different heaps introduces undesired behaviour. We will see in the following that the definition of a condition to rule out such commands coincides with a concept already used in earlier separation logic literature. The concept is called *preciseness* and will enable us to define a subset of commands that satisfy the concurrency property to get a better intuition for it.

First, note that for a test p we have $(\sigma,\tau) \in p \Leftrightarrow (\sigma,\sigma) \in p \wedge \sigma = \tau$. Hence, in the following we abbreviate for a test p the formula $(\sigma,\tau) \in p$ by $\sigma \in p$. Preciseness is used to characterise a subset of predicates of separation algebras [6].

Definition 7.1. A test p is called *precise* iff for all states σ, there exists at most one substate σ' for which $\sigma' \in p$, i.e.,

$$\forall \sigma, \sigma_1, \sigma_2 : (\sigma_1 \in p \wedge \sigma_2 \in p \wedge \sigma_1 \sqsubseteq \sigma \wedge \sigma_2 \sqsubseteq \sigma) \Rightarrow \sigma_1 = \sigma_2$$

where the *substate* relation is defined by $\sigma_0 \sqsubseteq \sigma_2 \Leftrightarrow_{df} \exists \sigma_1 : \sigma_0 \# \sigma_1 \wedge \sigma_0 \bullet \sigma_1 = \sigma_2$.

Intuitively in the heap model, a subheap $h' \subseteq h$ that satisfies a precise test, can be unambiguously identified in h. Examples for precise tests are

$$x \mapsto 1 \quad \text{or} \quad x \mapsto - * y \mapsto -$$

where $x \mapsto - =_{df} \bigcup_{n \in \mathbb{N}} x \mapsto n$. Negative examples are

$$\mathsf{true} \quad \text{or} \quad (x \mapsto -) * \mathsf{true} \quad \text{or} \quad (x \mapsto - * y \mapsto -) \vee y \mapsto - \ .$$

It turns out that precise tests can also be defined in the relational approach by the use of the split and join relations with tests. We start by an intermediate result that facilitates the proof of Lemma 7.3.

Lemma 7.2. $\sigma_1 \in p \wedge \sigma_1 \sqsubseteq \sigma \Leftrightarrow \exists \sigma_2 : ((\sigma_1,\sigma_2),\sigma) \in (p \times \mathsf{skip}) \,;\, \rhd$.

Proof.

$$\begin{array}{ll} & \sigma_1 \in p \wedge \sigma_1 \sqsubseteq \sigma \\ \Leftrightarrow & \quad \{\![\text{ definition of } \sqsubseteq \ ]\!\} \\ & \sigma_1 \in p \wedge \exists \sigma_2 : \ \sigma_1 \# \sigma_2 \wedge \sigma_1 \bullet \sigma_2 = \sigma \\ \Leftrightarrow & \quad \{\![\text{ logic, } \sigma_2 \in \mathsf{skip} \Leftrightarrow \mathsf{true} \ ]\!\} \\ & \exists \sigma_2 : \ \sigma_1 \in p \wedge \sigma_2 \in \mathsf{skip} \wedge \sigma_1 \# \sigma_2 \wedge \sigma_1 \bullet \sigma_2 = \sigma \\ \Leftrightarrow & \quad \{\![\text{ definition of } \rhd \ ]\!\} \\ & \exists \sigma_2 : \ ((\sigma_1,\sigma_2),\sigma) \in (p \times \mathsf{skip}) \,;\, \rhd \end{array}$$

□

Lemma 7.3. *If a test p satisfies*

$$(p \times \mathsf{skip}) \,;\, \rhd \,;\, \lhd \,;\, (p \times \mathsf{skip}) \subseteq p \times \mathsf{skip} \tag{5}$$

then p is precise. If the underlying separation algebra is cancellative then the reverse implication holds as well.

Proof. Using Lemma 7.2 we rewrite Definition 7.1:

$$
\begin{array}{ll}
& \forall\, \sigma, \sigma_1, \sigma_2 : (\sigma_1 \in p \ \wedge\ \sigma_2 \in p \ \wedge\ \sigma_1 \sqsubseteq \sigma \ \wedge\ \sigma_2 \sqsubseteq \sigma) \ \Rightarrow\ \sigma_1 = \sigma_2 \\
\Leftrightarrow & \quad \{\![\text{ logic }]\!\} \\
& \forall\, \sigma_1, \sigma_2 : (\exists\, \sigma :\ \sigma_1 \in p \ \wedge\ \sigma_2 \in p \ \wedge\ \sigma_1 \sqsubseteq \sigma \ \wedge\ \sigma_2 \sqsubseteq \sigma) \ \Rightarrow\ \sigma_1 = \sigma_2 \\
\Leftrightarrow & \quad \{\![\text{ Lemma 7.2, } \lhd \text{ is the converse of } \rhd \text{ }]\!\} \\
& \forall\, \sigma_1, \sigma_2 : (\exists\, \sigma :\ (\exists\, \tau_1 :\ ((\sigma_1, \tau_1), \sigma) \in (p \times \mathsf{skip})\,;\, \rhd) \wedge \\
& (\exists\, \tau_2 :\ (\sigma, (\sigma_2, \tau_2)) \in \lhd\,;\, (p \times \mathsf{skip})\,) \ \Rightarrow\ \sigma_1 = \sigma_2 \\
\Leftrightarrow & \quad \{\![\text{ definition of ; , logic }]\!\} \\
& \forall\, \sigma_1, \sigma_2, \tau_1, \tau_2 :\ (\,((\sigma_1, \tau_1), (\sigma_2, \tau_2)) \in (p \times \mathsf{skip})\,;\, \rhd\,;\, \lhd\,;\, (p \times \mathsf{skip})\,) \ \Rightarrow\ \sigma_1 = \sigma_2 \\
\Leftarrow & \quad \{\![\text{ definition of tests and } \times \text{ }]\!\} \\
& \forall\, \sigma_1, \sigma_2, \tau_1, \tau_2 :\ (\,((\sigma_1, \tau_1), (\sigma_2, \tau_2)) \in (p \times \mathsf{skip})\,;\, \rhd\,;\, \lhd\,;\, (p \times \mathsf{skip})\,) \\
& \ \Rightarrow\ ((\sigma_1, \tau_1), (\sigma_2, \tau_2)) \in p \times \mathsf{skip} \\
\Leftrightarrow & \quad \{\![\text{ logic }]\!\} \\
& (p \times \mathsf{skip})\,;\, \rhd\,;\, \lhd\,;\, (p \times \mathsf{skip}) \ \subseteq\ p \times \mathsf{skip}
\end{array}
$$

By cancellativity, i.e., for arbitrary σ, τ_1, τ_2. $\sigma \bullet \tau_1 = \sigma \bullet \tau_2 \ \Rightarrow\ \tau_1 = \tau_2$, the above implication turns into an equivalence. □

For the rest of this section we assume a cancellative separation algebra and hence use the stronger condition in Equation (5) as a pointfree characterisation of precise tests, since cancellativity is a natural requirement satisfied by most separation algebras anyway. As a sanity check, also a fully pointfree proof of $*$-distributivity over $\cap$ is possible. This property is stated and commonly used in various papers on separation logic [6, 12].

Lemma 7.4. *If p satisfies Equation* (5) *then for arbitrary tests q, r*

$$p * (q \cap r) = p * q \cap p * r \ .$$

A proof can be found in the Appendix.

Now, using this class of predicates, it turns out that commands with a precise domain satisfy the concurrency property. This enables a characterisation of a subset of commands that validate the concurrency rule II.

Definition 7.5. A command Q is called *domain-precise* iff ΔQ is precise.

Lemma 7.6. *Let Q be domain-precise. Then Q and an arbitrary command R have the concurrency property.*

Proof.

$$
\begin{array}{ll}
& (\Delta Q \times \Delta R)\,;\, \rhd\,;\, Q * R \\
= & \quad \{\![\text{ neutrality, } \times/;\text{ - exchange }]\!\} \\
& (\mathsf{skip} \times \Delta R)\,;\, (\Delta Q \times \mathsf{skip})\,;\, \rhd\,;\, Q * R \\
= & \quad \{\![\text{ definition of * }]\!\} \\
& (\mathsf{skip} \times \Delta R)\,;\, (\Delta Q \times \mathsf{skip})\,;\, \rhd\,;\, \lhd\,;\, (Q \times R)\,;\, \rhd \\
= & \quad \{\![\text{ property of } \Delta_{\_}\text{, neutrality, } \times/;\text{ - exchange }]\!\} \\
& (\mathsf{skip} \times \Delta R)\,;\, (\Delta Q \times \mathsf{skip})\,;\, \rhd\,;\, \lhd\,;\, (\Delta Q \times \mathsf{skip})\,;\, (Q \times R)\,;\, \rhd \\
= & \quad \{\![\text{ Lemma 7.3 }]\!\} \\
& (\mathsf{skip} \times \Delta R)\,;\, (\Delta Q \times \mathsf{skip})\,;\, (Q \times R)\,;\, \rhd \\
= & \quad \{\![\text{ neutrality, } \times/;\text{ - exchange }]\!\} \\
& (\Delta Q \times \Delta R)\,;\, (Q \times R)\,;\, \rhd \\
= & \quad \{\![\times/;\text{ - exchange, property of } \Delta_{\_} \text{ }]\!\} \\
& (Q \times R)\,;\, \rhd
\end{array}
$$

□

Corollary 7.7. *Let Q be domain-precise. Then Q and an arbitrary command R validate Concurrency Rule II. In particular, all pairs of domain-precise commands validate Concurrency Rule II.*

Note, that the reverse direction of Lemma 7.6 does not hold. One can instantiate the concurrency property, e.g., with commands in the heap model given in Equation (1). These commands are not domain-precise; e.g., the domain of $[x] := y$ is $(x \mapsto -) * \mathsf{true}$. To get precise versions of allocation or mutation commands, one would require the redefinition

$$[x] := y \;=_{df}\; \{(\{(x,n)\},\{(x,y)\}) \,:\, n \in \mathbb{N}\} \qquad \text{and} \qquad \mathsf{arb\_alloc}\,(x) \;=_{df}\; \{(\emptyset,\{(x,n)\}) \,:\, n \in \mathbb{N}\}.$$

By these definitions, examples for domain-precise commands are ($\cup$ models non-deterministic choice)

$$[x] := y \;, \qquad ([x] := y) \cup ([x] := z) \;, \qquad \mathsf{arb\_alloc}\,(x) \;,$$

since their domains equal $x \mapsto -$ in the first and second command and emp in the last one. Negative examples are

$$([1] := 1) \cup ([2] := 1) \qquad \text{and} \qquad Q * \mathsf{skip} \quad \text{if } Q \text{ and } \mathsf{skip} \text{ are forward compatible .}$$

The domain of the latter commands above is $\Delta Q * \mathsf{skip}$ by Lemma 4.1. Concrete commands of this form are given in Equation (1). Such commands are also called *local* and will be discussed more detailed in Section 9.

8. Discussion

We have now presented two variations of the concurrency rule in our relational calculus. To round off this theme, we exemplify some differences of the proof rules with a program `ms` for parallel mergesort [13]:

$$\frac{\begin{array}{c}\{array(a,i,m)\}\,\mathtt{ms}(a,i,m)\,\{sorted(a,i,m)\} \\ \{array(a,m+1,j)\}\,\mathtt{ms}(a,m+1,j)\,\{sorted(a,m+1,j)\}\end{array}}{\{array(a,i,m) * array(a,m+1,j)\,\}\,\mathtt{ms}(a,i,m) * \mathtt{ms}(a,m+1,j)\,\{\,sorted(a,i,m) * sorted(a,m+1,j)\}}$$

where $array(a,i,j)$, assuming $i < j$, asserts that the store range with addresses $a+i$ to $a+j$ forms an array, i.e., contains elements of equal type, and $sorted(a,i,j)$ ensures that the content in that range is sorted.

Consider first the following definition of the *array* predicate for appropriate i, j, m

$$array(a,i,j) \;=_{df}\; \{(h,h) : \{a+i,\ldots,a+j\} = dom(h), i < j, \{i,j\} \subset \mathbb{N}\} \;.$$

It can be seen that this is a precise predicate, as the domain of each considered heap equals $\{a+i,\ldots,a+j\}$. If we additionally define the command `ms` to satisfy

$$dom(\mathtt{ms}(a,i,j)) \;=_{df}\; \{(h,h) : h \in array(a,i,j)\} \tag{6}$$

then the above instantiation of the concurrency rule II with `ms` is a valid example. Now if we would vary the above definition to

$$array(a,i,j) \;=_{df}\; \{(h,h) : \{a+i,\ldots,a+j\} \subseteq dom(h), i < j, \{i,j\} \subset \mathbb{N}\} \;,$$

the predicate would satisfy the equation $array(a,i,j) * I = array(a,i,j)$ and hence become an imprecise predicate, i.e., it could not be used with the above approach. Nevertheless, since the commands $\mathtt{ms}(a,i,m)$ and $\mathtt{ms}(a,m+1,j)$ satisfy Equation (6), they can be used with the concurrency property and the above instantiation is still valid.

Note that both definitions of the *array* predicate would work with Concurrency Rule I, since by Equation (6) we always have $dom(\mathtt{ms}(a,i,j)) \subseteq array(a,i,j)$. Unfortunately, if we would, e.g., change the first triple in the premise above to

$$\{sorted(a,i,m)\}\,\mathtt{ms}(a,i,m)\,\{sorted(a,i,m)\}$$

Concurrency Rule I could not be used, since $sorted(a,i,m) \subset array(a,i,j) = dom(\mathtt{ms}(a,i,j))$. Again this condition is enough to allow using Concurrency Rule II.

A major difference that can be seen is that the domain restriction in Concurrency Rule I is very liberal. It allows, e.g., triples like

$$\{array(a,i,m) \vee array(a,m,j)\}\,\mathtt{ms}(a,i,m)\,\{sorted(a,i,m)\}\ .$$

Informally, states that do not belong to the domain of `ms` are discarded as long as there exists at least one starting state for each execution of `ms`. In contrast, the condition $p \subseteq dom(Q)$ of Concurrency Rule II ensures that each state of the precondition p has at least one execution in Q.

One advantage of Concurrency Rule I is that it only requires that the domains of the commands Q_i coincide with the respective preconditions, but needs no connection between the Q_i. Contrarily, Concurrency Rule II is more liberal w.r.t. the preconditions but requires the Q_i to have the concurrency property.

9. Locality and the Frame Rule

We now turn to another important proof rule for modular reasoning. Validity of that rule is based on the concept of *locality* which describes the behaviour of programs that only access certain subsets of the available resources. Hence locality allows embedding a program into a larger context so that any resource not accessed by that program remains unchanged. This fact is expressed by the *frame rule*

$$\frac{\{p\}\,Q\,\{q\}}{\{p*r\}\,Q\,\{q*r\}}\ .$$

In [8, 14] the frame rule is obtained as a special instance of the general concurrency rule by setting $Q_1 = Q$ and $Q_2 = \mathsf{skip}$ there. In attempting to do the same in our relational setting we notice that Concurrency Rule I cannot be used because of the premise $\Delta\,\mathsf{skip} \subseteq p_2$ which fails for all $p_2 \neq \mathsf{skip}$.

However, there is a different specialisation which comes close to the original frame rule:

Theorem 9.1 (Frame Rule I).

$$\frac{\{p\}\,Q\,\{q\} \quad \Delta Q \subseteq p}{\{p*r\}\,Q*r\,\{p*r\}}$$

Proof. This arises from Concurrency Rule I by setting $Q_1 = Q, Q_2 = r, p_1 = p, p_2 = r, r_1 = q$ and $r_2 = r$. The premise $\Delta Q_2 \subseteq p_2$ is satisfied automatically, since $Q_2 = p_2$ and every test equals its own domain. Finally, we use Corollary 5.6. □

Although this rule shows clearly that Q may be embedded into the context $Q*r$ that does not modify r, it is different in spirit from the original rule which says that Q does not need more resources than admitted by p and hence can be used unmodified to take precondition $p*r$ to postcondition $q*r$. We will obtain such a rule from Concurrency Rule II by imposing different conditions on Q.

To do that we first remind the reader of a central result of [8]. A predicate transformer F in that model is called *local* iff it satisfies the equation

$$F * \mathsf{skip} = F\,.$$

This equation characterises exactly the above-mentioned modularity concept. Each execution of the program F can be replaced by one that only operates on the necessary and possible smaller part of the state while the rest of it remains unchanged (abstractly denoted by the program skip). In the following we derive the same compact characterisation for commands.

First remember that emp is the unit of $*$ and $\mathsf{emp} \subseteq \mathsf{skip}$.

Lemma 9.2. *For arbitrary commands Q we have $Q \subseteq Q * \mathsf{skip}$.*

Proof. $Q = Q * \mathsf{emp} \subseteq Q * \mathsf{skip}$. □

To get the other inclusion, i.e., $Q * \mathsf{skip} \subseteq Q$, we need an additional assumption about Q. Surprisingly, this inequation can be derived from a property given in [2] which was called test preservation and used there to prove soundness of the frame rule.

Definition 9.3. A command Q *preserves* a test r iff

$$\lhd ; (Q \times r) ; \# \;\subseteq\; Q ; \lhd ; (\mathsf{skip} \times r) \; . \tag{7}$$

We call a command Q *local* iff Q preserves all tests.

Formula (7) means that when running Q on a part of the state such that the remainder of the state satisfies r one might also run Q first on the complete state and will still find an r-part in the result state.

Preservation of r by Q is an abstraction of the property that Q does not modify the free variables of r; a more refined version of this definition was given in [2] and another one was studied in [15]. Locality as preservation of all tests, does not seem very realistic in that domain. Nevertheless, it turns out to be equivalent to the algebraic formulation of [8]:

Theorem 9.4. *A command Q is local iff $Q * \mathsf{skip} \subseteq Q$.*

The proof can be found in the Appendix. In particular, by Lemma 2.8, skip is local.

By Theorem 9.4 we may, as in [8], define local commands as fixpoints of the localising operation $(\,\cdot\,) * \mathsf{skip}$.

With this definition of locality we now prove our variant of the frame rule. We take a similar direction as in Section 5 by defining sufficient conditions needed for a soundness proof. First notice that in [8] the compact definition of locality and the full exchange law are used to get validity of the small exchange law for local predicate transformers. The small exchange law reads

$$(P * Q) ; R \;\leq\; (P ; R) * Q$$

for programs P, Q, R and the refinement order $\leq$ of a locality bimonoid. Moreover this law is equivalent to soundness of the frame rule in such a structure. In our approach locality has the same definition, but the small exchange law does not hold. Therefore again a further sufficient condition is needed to simulate the relevant part of the small exchange law.

In [2] we used a relational variant of the frame property to prove the frame rule. We will use it in this paper in a simplified form. It is obtained from the concurrency property for Q and skip, which would spell out to

$$(\Delta Q \times \mathsf{skip}) ; \rhd ; (Q * \mathsf{skip}) \;\subseteq\; (Q \times \mathsf{skip}) ; \rhd \; .$$

In the special case where Q is local this reduces to a simpler form. Therefore we define

Definition 9.5. A command Q has the *frame property* iff

$$(\Delta Q \times \mathsf{skip}) ; \rhd ; Q \;\subseteq\; (Q \times \mathsf{skip}) ; \rhd \; .$$

An equational form analogous to the one in Definition 6.1 is possible. As a special case of the concurrency property (cf. Definition 6.1), this property analogously provides a fault-avoiding assignment of resources to commands. The skip part characterises the resources that remain untouched by Q. Finally, we can relate the frame property again to a special case of the exchange law that will allow us to prove the frame rule in our approach.

Lemma 9.6. *Assume that Q has the frame property and $p \subseteq \Delta Q$. Then for all test r we have following special case of the small exchange property:*

$$(p * r) ; Q \;\subseteq\; (p ; Q) * r \; .$$

Proof.

$$
\begin{array}{ll}
 & (p * r)\,;Q \\
= & \quad \{\![\text{ assumption }]\!\} \\
 & ((p\,;\Delta Q) * r)\,;Q \\
= & \quad \{\![\text{ definition of } * \text{ and } ;/\times \text{ exchange (2) }]\!\} \\
 & \lhd\,;(p \times r)\,;(\Delta Q \times \mathsf{skip})\,;\rhd\,;Q \\
\subseteq & \quad \{\![\text{ frame property }]\!\} \\
 & \lhd\,;(p \times r)\,;(Q \times \mathsf{skip})\,;\rhd \\
= & \quad \{\![\,;/\times \text{ exchange (2) and definition of } * \,]\!\} \\
 & (p\,;Q) * r\ .
\end{array}
$$

□

The premise $p \subseteq \Delta Q$ informally states that p already ensures enough resources for the execution of Q.

Now we can easily prove a corresponding frame rule.

Theorem 9.7 (Frame Rule II). *Let Q be local and have the frame property. Then*

$$\frac{\{p\}\,Q\,\{q\} \quad p \subseteq \Delta Q}{\{p * r\}\,Q\,\{q * r\}}\ .$$

Proof. Apply Concurrency Rule II to $Q_1 = Q, Q_2 = \mathsf{skip}, p_1 = p, p_2 = r, q_1 = q, q_2 = r$ and use locality of Q. □

By the close connection to Concurrency Rule II we have again a weak reverse implication.

Lemma 9.8. *Assume validity of the Frame Rule. Then for all local commands P and tests q, r we have*

$$(P\,;q) * r \ \subseteq\ P\,;(q * r)\ .$$

Proof. This is immediate from Lemma 5.7 by setting $P_1 = P, r_1 = q, P_2 = \mathsf{skip}, r_2 = r$ and using locality of P. □

Next, we compare the structure of our proofs with the corresponding ones in [8] to point out the main differences. Since the small exchange law is not valid in our relational setting (not even for local commands), it was necessary to constrain the set of commands considered in the frame rule by an additional assumption. It turned out in [2] that the relational version of the frame property was an adequate substitute that already ensured the relevant part of the small exchange law. Structurally, the proof of this frame rule becomes as simple as the one for the predicate transformer approach in [8]. Due to the angelic character of relations, the rule itself needs the additional premise $p \subseteq \Delta Q$.

It is again worth mentioning that all proofs in this section do not require the assumption of compatible commands. The assumption of locality of a command is also independent of its compatibility with skip. To see this, consider the heap allocation command defined in Equation (3) which is local. It is not difficult to show that $\mathsf{arb\_alloc}\,(1)$ is not compatible with skip. The reverse implication does not hold either, since each test p is compatible with skip by Corollary (4.2) but not every test satisfies $p * \mathsf{skip} \subseteq p$ as, e.g., $1 \mapsto 2$.

As a further remark, the approach of [8] requires special functions for the semantics of Hoare triples. They are called *best predicate transformers* and are used as an adequate substitute for assertions. Intuitively these functions simulate the allocation of resources that are characterised by pre- and postconditions. In our calculus this can be handled by composing tests with the universal relation. However, since we have a non-trivial test algebra in the relational setting, tests by themselves already admit a suitable representation of pre- and postconditions.

10. Dual Correctness Triples

The previous sections presented an approach to include the concurrency and frame rules in the given relational approach to separation logic [2] by requiring additional assumptions and hence restricting the proof rules. In this section we present some further applications for the reverse exchange law. We link it with some definitions of triples dual to the one of Hoare triples. By this we will again see that the concurrency and frame rules for those triples can easily be derived using the reverse exchange law.

Definition 10.1. As in [16], for commands P, Q, R we define *Plotkin* triples by

$$\langle P, Q\rangle \to R \;\; \Leftrightarrow_{df} \;\; R \subseteq P\,;Q$$

and dual partial correctness triples by

$$P\,[Q]\,R \;\; \Leftrightarrow_{df} \;\; P \subseteq Q\,;R\;.$$

Intuitively, the former characterises possible final states satisfying the postcondition R after execution of Q starting from the precondition P. One can think of labelled transition systems, where Q represents some sequence of actions that possibly leads from a configuration or state in P to some final configuration of R. The notation is inspired by Plotkin's structural operational semantics [17] in which $\langle s, C\rangle \to t$ means that evaluation of term C starting in state s may lead to term t.

The dual partial correctness triples describe possible starting states of P that end in R after execution of Q. According to [16], dual partial correctness triples can, e.g., be used as a method for the generation of test cases. Assuming R represents erroneous final states of Q then P characterises some conditions that will lead to such error situations. Plotkin triples can be used for a dual application.

Using the relationship between tests and commands given in Section 5, in our calculus the dual partial correctness triples transform into

$$(p\,;U)\,[Q]\,(q\,;U) \;\Leftrightarrow\; p\,;U \subseteq Q\,;(q\,;U) \;\Leftrightarrow\; p \subseteq (Q\,;q)\,;U \;\Leftrightarrow\; p \subseteq \Delta Q\,;q$$

and, symmetrically, Plotkin triples into

$$\langle U\,;p\;,Q\rangle \to U\,;q \;\Leftrightarrow\; U\,;q \subseteq (U\,;p)\,;Q \;\Leftrightarrow\; q \subseteq U\,;(p\,;Q) \;\Leftrightarrow\; q \subseteq (p\,;Q)\nabla\;,$$

where $\_\nabla$ denotes the codomain operator. We concentrate on dual partial correctness triples and use the abbreviation $p\;[\![\,Q\,]\!]\;q \;\Leftrightarrow_{df}\; (p\,;U)\,[Q]\,(q\,;U) \;\Leftrightarrow\; p \subseteq \Delta Q\,;q$. Dual results hold for Plotkin triples.

The central interest of these new triples lies in the following result.

Lemma 10.2. *The concurrency rule for dual partial correctness or Plotkin triples holds iff the reverse exchange law holds.*

A proof for this lemma can be derived dually to [8]. Unfortunately, in our setting the reverse exchange law does not hold unconditionally. However, we will see that under an assumption of compatibility the concurrency and frame rules can still be derived. Note that it was not needed to assume compatibility for the proof rules with Hoare triples, since tests already come with that property. In contrast, the new triples do not need additional assumptions besides the compatibility condition.

Lemma 10.3. *If Q_1, Q_2 are forward compatible then the concurrency rule for dual partial correctness triples holds, i.e., for tests p_1, p_2, q_1, q_2*

$$\frac{p_1\;[\![\,Q_1\,]\!]\;q_1 \qquad p_2\;[\![\,Q_2\,]\!]\;q_2}{p_1 * p_2\;[\![\,Q_1 * Q_2\,]\!]\;q_1 * q_2}\;.$$

Again this holds also when Q_1 and Q_2 are backward compatible and Plotkin instead of dual partial correctness triples are used.

Proof. By assumption we have $p_1 \subseteq \Delta Q_1 \,;\, q_1$, $p_2 \subseteq \Delta Q_2 \,;\, q_2$ and the restricted variant of the reverse exchange law. Hence

$$\begin{array}{ll} & p_1 * p_2 \\ \subseteq & \Delta Q_1 \,;\, q_1 * \Delta Q_2 \,;\, q_2 \\ = & \Delta(Q_1 \,;\, q_1) * (Q_2 \,;\, q_2) \\ \subseteq & \Delta(Q_1 * Q_2) \,;\, (R_1 * R_2) \;. \end{array}$$

□

We characterised the behaviour of the triples "dual" on purpose, since the calculations given above are symmetric to the algebraic approach of [8]. It is not hard to see that a further application of the compact characterisation of locality presented in Section 9 also gives the following result.

Lemma 10.4. *If Q is local and forward compatible with* skip *then the frame rule for dual partial correctness triples holds, i.e.,*

$$\frac{p \;[\![Q]\!]\; q}{p * r \;[\![Q]\!]\; q * r} \;.$$

(A dual result again holds for Plotkin triples).

Proof. Assume $p \subseteq \Delta Q \,;\, q$ for a command Q and test q. Hence

$$\begin{array}{ll} & p * r \\ \subseteq & \Delta(Q \,;\, q) * (\mathsf{skip} \,;\, r) \\ = & \Delta(Q \,;\, q) * \Delta\mathsf{skip} \,;\, r \\ = & \Delta((Q \,;\, q) * (\mathsf{skip} \,;\, r)) \\ \subseteq & \Delta((Q * \mathsf{skip}) \,;\, (q * r)) \\ \subseteq & \Delta(Q \,;\, (q * r)) \;. \end{array}$$

□

11. Further Variations

Both proof rules of the previous section have the restriction that compatible pairs of commands are needed. The reason for this is that, by Lemma 2.6, in the relational approach only $\# \subseteq \rhd \,;\, \lhd$ holds. If we would have $\mathsf{skip} \times \mathsf{skip} \subseteq \rhd \,;\, \lhd$ the proof of the reverse exchange law would not have any restrictions. However this requires an extension of the definition of $\rhd$ such that the composition $\rhd \,;\, \lhd$ also copes with non-combinable pairs of states.

An idea would be to extend the underlying separation algebra by an extra element to denote incompatible state combination, i.e., in particular to redefine state combination as

$$\sigma \bullet \tau = \sigma_\perp \qquad \text{iff} \quad \neg\, \sigma \# \tau$$

for a new state $\sigma_\perp$. The new carrier set is then defined by $\Sigma_\perp =_{df} \Sigma \,\dot\cup\, \{\sigma_\perp\}$.

This modification allows a relational model of the algebraic structure of a *locality bimonoid* defined in [8].

Definition 11.1. A *locality bimonoid* is an algebraic structure $(S, \leq, *, 1_*, ;, 1_;)$ where $(S, \leq)$ is partially ordered and $*$, ; are monotone operations on S. Moreover, $(S, *, 1_*)$ needs to be a commutative monoid and $(S, ;, 1_;)$ a monoid. Additionally, the structure has to satisfy the exchange law and $1 * 1 = 1$.

To obtain a relational model for this structure one may interpret the order $\leq$ as the reverse set inclusion order $\supseteq$. Of course, by this the mentioned reverse exchange law turns into the normal one and the relational approach into a refinement-based setting.

In summary, using Lemma 2.8, we have the following result.

Lemma 11.2. $(\mathcal{P}(\Sigma_\perp \times \Sigma_\perp), \supseteq, *, \mathsf{emp}, ;, \mathsf{skip})$ *forms a locality bimonoid.*

Note that by reversing the set inclusion order turns $\sqcup$ and $\sqcap$ into $\cap$ and $\cup$. At the same time, the test subalgebra is used as an algebraic counterpart to model assertions. Hence, the interpretation of the notion of a test becomes very unnatural, since, e.g., $p \wedge q$ will be identified, unusually, in the algebra with $p \sqcup q$ and $p \vee q$ with $p \sqcap q$. Algebraically these modifications of the model entail simplifications. There are no additional constraints needed to validate the reverse exchange law and hence the original concurrency and frame rules hold for this particular model. The reason is the inequation $\mathsf{skip} \times \mathsf{skip} \subseteq \rhd ; \lhd$ that requires the introduction of an extra failure-state to capture the join of non-combinable states. However, considering this extra failure-state makes the whole approach more complicated and artificial from the model-theoretical view.

12. Conclusion

Although neither the full nor small exchange law holds in the relational calculus, we were still able to obtain reasonable variants of the concurrency and frame rules. The proofs greatly benefit from the (restricted) reverse exchange law and hence are almost as simple as the ones in [8]. We have also obtained some new results on compatibility as well as a more systematic and symmetric presentation of the different variants of concurrency and frame rules.

Further work on this approach will include investigations on so-called interference relations [18]. The intention with these is to provide admissible behaviour of commands in a concurrent context so that unwanted interference between these commands is excluded. By studying that area we hope to extend our relation-algebraic approach with its many advantages to a new application domain.

Acknowledgements: We are grateful to Tony Hoare for fruitful discussions and comments and to Andreas Zelend for valuable remarks. Moreover, we thank all reviewers for their comments that helped to significantly improve the presentation of this paper. This research was partially funded by the DFG project *MO 690/9-1 AlgSep — Algebraic Calculi for Separation Logic.*

References

[1] H.-H. Dang, B. Möller, Reverse Exchange for Concurrency and Local Reasoning, in: J. Gibbons, P. Nogueira (Eds.), Mathematics of Program Construction - 11th International Conference, MPC 2012, Madrid, Spain, June 25-27, 2012. Proceedings, volume 7342 of *LNCS*, Springer, 2012, pp. 177–197.

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

[3] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, 2002, pp. 55–74.

[4] S. Brookes, A semantics for concurrent separation logic, Theoretical Computer Science 375 (2007) 227–270.

[5] P. W. O'Hearn, J. C. Reynolds, H. Yang, Local reasoning about programs that alter data structures, in: L. Fribourg (Ed.), CSL '01: Proceedings of the 15th International Workshop on Computer Science Logic, volume 2142 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 1–19.

[6] C. Calcagno, P.-W. O'Hearn, H. Yang, Local Action and Abstract Separation Logic, in: Proc. of the 22nd Symposium on Logic in Computer Science, IEEE Press, 2007, pp. 366–378.

[7] C.-A.-R. Hoare, B. Möller, G. Struth, I. Wehrman, Concurrent Kleene Algebra and its Foundations, Journal of Logic and Algebraic Programming 80 (2011) 266–296.

[8] C.-A.-R. Hoare, A. Hussain, B. Möller, P.-W. O'Hearn, R.-L. Petersen, G. Struth, On Locality and the Exchange Law for Concurrent Processes, in: J.-P. Katoen, B. König (Eds.), CONCUR 2011, volume 6901 of *LNCS*, Springer, 2011, pp. 250–264.

[9] V. Vafeiadis, Concurrent separation logic and operational semantics, Electr. Notes Theor. Comput. Sci. 276 (2011) 335–351.

[10] B. Möller, Kleene getting lazy, Science of Computer Programming 65 (2007) 195–214.

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

[12] P. W. O'Hearn, J. C. Reynolds, H. Yang, Separation and information hiding, ACM Trans. Program. Lang. Syst. 31 (2009) 1–50.

[13] P.-W. O'Hearn, Resources, Concurrency, and Local Reasoning, Theoretical Computer Science 375 (2007) 271–307.

[14] C. Hoare, B. Möller, G. Struth, I. Wehrman, Concurrent Kleene Algebra and its Foundations, Journal of Logic and Algebraic Programming 80 (2011) 266–296.

[15] H.-H. Dang, P. Höfner, Variable side conditions and greatest relations in algebraic separation logic, in: H. de Swart (Ed.), Proceedings of the 12th international conference on Relational and Algebraic Methods in Computer Science, volume 6663 of *LNCS*, Springer, 2011, pp. 125–140.
[16] C.-A.-R. Hoare, An Algebra for Program Designs, Notes on Summer School in Software Engineering and Verification in Moscow, 2011. URL: `http://research.microsoft.com/en-us/um/redmond/events/sssev2011/slides/tony_1.pptx`.
[17] G. D. Plotkin, A structural approach to operational semantics, J. Log. Algebr. Program. 60–61 (2004) 17–139.
[18] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, V. Vafeiadis, Concurrent abstract predicates, in: T. D'Hondt (Ed.), ECOOP 2010 — Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings, volume 6183 of *LNCS*, Springer, 2010, pp. 504–528.

13. Appendix: Deferred Proofs

Proof of Lemma 2.6.
For the first claim we calculate as follows.

$$
\begin{array}{ll}
& (\sigma_1,\sigma_2)\ \#\ (\tau_1,\tau_2) \\
\Leftrightarrow & \quad \{\![\text{ Definition 2.4 }]\!\} \\
& \sigma_1\#\sigma_2 \ \wedge\ \sigma_1=\tau_1 \ \wedge\ \sigma_2=\tau_2 \\
\Leftrightarrow & \quad \{\![\text{ logic }]\!\} \\
& \sigma_1\#\sigma_2 \ \wedge\ \tau_1\#\tau_2 \ \wedge\ \sigma_1=\tau_1 \ \wedge\ \sigma_2=\tau_2 \\
\Leftrightarrow & \quad \{\![\text{ Definition 2.1 }]\!\} \\
& \sigma_1\#\sigma_2 \ \wedge\ \exists\,\sigma : \sigma_1\bullet\sigma_2=\sigma \ \wedge\ \tau_1\#\tau_2 \ \wedge\ \exists\,\tau : \tau_1\bullet\tau_2=\tau \ \wedge\ \sigma_1=\tau_1 \ \wedge\ \sigma_2=\tau_2 \\
\Leftrightarrow & \quad \{\![\ \sigma_1=\tau_1 \ \wedge\ \sigma_2=\tau_2 \text{ implies } \sigma=\tau \ ]\!\} \\
& \exists\,\sigma :\ \sigma_1\#\sigma_2 \ \wedge\ \sigma_1\bullet\sigma_2=\sigma \ \wedge\ \tau_1\bullet\tau_2=\sigma \ \wedge\ \tau_1\#\tau_2 \ \wedge\ \sigma_1=\tau_1 \ \wedge\ \sigma_2=\tau_2 \\
\Leftrightarrow & \quad \{\![\text{ logic }]\!\} \\
& \exists\,\sigma :\ (\sigma_1,\sigma_2)\ \rhd\ \sigma \ \wedge\ \sigma\ \lhd\ (\tau_1,\tau_2) \ \wedge\ \sigma_1=\tau_1 \ \wedge\ \sigma_2=\tau_2 \\
\Leftrightarrow & \quad \{\![\text{ definition of } ; \ ]\!\} \\
& (\sigma_1,\sigma_2)\ (\rhd\ ;\lhd)\ (\tau_1,\tau_2) \ \wedge\ \sigma_1=\tau_1 \ \wedge\ \sigma_2=\tau_2 \\
\Leftrightarrow & \quad \{\![\text{ definition of } \mathsf{id} \ ]\!\} \\
& (\sigma_1,\sigma_2)\ (\rhd\ ;\lhd\ \cap\ \mathsf{id})\ (\tau_1,\tau_2)
\end{array}
$$

The remaining claims are immediate from the definitions. □

Proof of Lemma 2.9.
For arbitrary σ we have

$$
\begin{array}{ll}
& \sigma\ \Delta(P*Q)\ \sigma \\
\Leftrightarrow & \quad \{\![\text{ definitions of } * \text{ and domain }]\!\} \\
& \exists\,\sigma_1,\sigma_2,\tau_1,\tau_2\,.\ \sigma_1\#\sigma_2 \ \wedge\ \sigma=\sigma_1\bullet\sigma_2 \ \wedge\ \tau_1\#\tau_2 \ \wedge\ \sigma_1\ P\ \tau_1 \ \wedge\ \sigma_2\ Q\ \tau_2 \\
\Rightarrow & \quad \{\![\text{ omitting conjunct } \tau_1\#\tau_2 \text{ and shifting quantification over } \tau_1,\tau_2 \ ]\!\} \\
& \exists\,\sigma_1,\sigma_2\,.\ \sigma_1\#\sigma_2 \ \wedge\ \sigma=\sigma_1\bullet\sigma_2 \ \wedge\ \exists\,\tau_1,\tau_2\,.\,\sigma_1\ P\ \tau_1 \ \wedge\ \sigma_2\ Q\ \tau_2 \\
\Leftrightarrow & \quad \{\![\text{ definition of domain }]\!\} \\
& \exists\,\sigma_1,\sigma_2\,.\ \sigma_1\#\sigma_2 \ \wedge\ \sigma=\sigma_1\bullet\sigma_2 \ \wedge\ \sigma_1\ \Delta P\ \sigma_1 \ \wedge\ \sigma_2\ \Delta Q\ \sigma_2 \\
\Leftrightarrow & \quad \{\![\text{ definition of } * \ ]\!\} \\
& \sigma\ (\Delta P * \Delta Q)\ \sigma\ .
\end{array}
$$

□

Proof of Lemma 7.4.
The $\subseteq$-direction follows from isotonicity. For the other direction we calculate

$$p*q \cap p*r$$

$$
\begin{array}{ll}
= & \quad \{\![\text{ ; on tests equals } \cap \ ]\!\} \\
& (p * q)\,;(p * r) \\
= & \quad \{\![\text{ definition of } *, \text{ neutrality }]\!\} \\
& \lhd\,;(\mathsf{skip}\,;p \times q\,;\mathsf{skip})\,\rhd\,;\lhd\,;(p\,;\mathsf{skip} \times \mathsf{skip}\,;r)\,;\rhd \\
= & \quad \{\![\ ;/\times \text{ - exchange }]\!\} \\
& \lhd\,;(\mathsf{skip} \times q)\,;(p \times \mathsf{skip})\,\rhd\,;\lhd\,;(p \times \mathsf{skip})\,;(\mathsf{skip} \times r)\,;\rhd \\
\subseteq & \quad \{\![\ p \text{ satisfies (5) }]\!\} \\
& \lhd\,;(\mathsf{skip} \times q)\,;(p \times \mathsf{skip})\,;(\mathsf{skip} \times r)\,;\rhd \\
= & \quad \{\![\ ;/\times \text{ - exchange, neutrality }]\!\} \\
& \lhd\,;(p \times (q\,;r))\,;\rhd \\
= & \quad \{\![\text{ definition of } *\,, \text{ ; on tests equals } \cap \ ]\!\} \\
& p * (q \cap r)\ .
\end{array}
$$

□

In Def. 9.3 we stated that a command Q *preserves* a test r iff

$$\lhd\,;(Q \times r)\,;\#\ \subseteq\ Q\,;\lhd\,;(\mathsf{skip} \times r)$$

and called a command Q *local* iff Q preserves all tests. We first list a few useful properties in connection with these notions.

Lemma 13.1.

1. skip *preserves* skip .
2. *For arbitrary* Q *and* r *we have*
$$\lhd\,;(Q \times r)\,;\#\ \ \subseteq\ \ (Q * \mathsf{skip})\,;\lhd\,;(\mathsf{skip} \times r)\ .$$
3. *If* Q *preserves a test* r *then* $Q * r\ \subseteq\ Q\,;(\mathsf{skip} * r)$.
In particular, $\mathsf{skip} * \mathsf{skip} \subseteq \mathsf{skip}$. *Hence if* Q *is local then* $Q * \mathsf{skip}\ \subseteq\ Q$.

Proof.

1. The claim follows immediately by setting $Q = \mathsf{skip} = r$ in Definition 9.3.
2. We calculate:
$$
\begin{array}{ll}
& \lhd\,;(Q \times r)\,;\# \\
= & \quad \{\![\text{ neutrality of } \mathsf{skip} \ ]\!\} \\
& \lhd\,;(Q\,;\mathsf{skip} \times \mathsf{skip}\,;r)\,;\# \\
= & \quad \{\![\ ;/\times \text{ exchange (2) }]\!\} \\
& \lhd\,;(Q \times \mathsf{skip})\,;(\mathsf{skip} \times r)\,;\# \\
= & \quad \{\![\text{ by Definition 2.4 }]\!\} \\
& \lhd\,;(Q \times \mathsf{skip})\,;\#\,;(\mathsf{skip} \times r) \\
\subseteq & \quad \{\![\ \# \subseteq \rhd\,;\lhd \text{ and isotony }]\!\} \\
& \lhd\,;(Q \times \mathsf{skip})\,;\rhd\,;\lhd\,;(\mathsf{skip} \times r) \\
= & \quad \{\![\text{ definition of } * \ ]\!\} \\
& (Q * \mathsf{skip})\,;\lhd\,;(\mathsf{skip} \times r)\ .
\end{array}
$$
3. The first claim is immediate from the definition of locality by right-composing both sides of the inclusion with $\rhd$, isotony and the definition of $*$. Hence the second claim is trivial by isotony. The third claim follows by setting $r = \mathsf{skip}$ and using $\mathsf{skip} * \mathsf{skip} = \mathsf{skip}$.

□

We can now give the

Proof of Theorem 9.4.
The direction (⇒) is just Lemma 13.1.3. For (⇐) we obtain by Lemma 13.1.3 and the assumption, for arbitrary test r,

$$\lhd\,;\,(Q\times r)\,;\#\;\subseteq\;(Q*\mathsf{skip})\,;\lhd\,;\,(\mathsf{skip}\times r)\;\subseteq\;Q\,;\lhd\,;\,(\mathsf{skip}\times r)\;.$$

□

Corollary 13.2. $Q*\mathsf{skip}\subseteq Q\;\Leftrightarrow\;\lhd\,;(Q\times\mathsf{skip})\,;\#\subseteq Q\,;\lhd\,.$

Proof. The direction ($\Leftarrow$) follows from isotony. For the other direction we immediately get by definition and isotony $\lhd\,;(Q\times\mathsf{skip})\,;\rhd\,;\lhd\subseteq Q\,;\lhd$, since $Q*\mathsf{skip}\subseteq Q$. Now the claim follows from Lemma 2.6 using $\#\subseteq\rhd\,;\lhd$. □

Next we turn to Section 4. To prove Lemma 4.1 we first sum up a few results.

Corollary 13.3. $\#\,;(U\times U)\,;\rhd=\rhd\,;U.$

For a proof we refer to [2].

Lemma 13.4. *If commands P,Q are forward compatible then* $(P\,;U)*(Q\,;U)=(P*Q)\,;U.$

Proof. We calculate

$$\begin{array}{ll}
 & (P\,;U)*(Q\,;U)\\
= & \quad\{\![\text{ definition of }*\,]\!\}\\
 & \lhd\,;(P\,;U\times Q\,;U)\,;\rhd\\
= & \quad\{\![\text{ Lemma 2.6, Equation (2) }]\!\}\\
 & \lhd\,;\#\,;(P\times Q)\,;(U\times U)\,;\rhd\\
\subseteq & \quad\{\![\ P,Q\text{ forward compatible }]\!\}\\
 & \lhd\,;(P\times Q)\,;\#\,;(U\times U)\,;\rhd\\
= & \quad\{\![\text{ Corollary 13.3 }]\!\}\\
 & \lhd\,;(P\times Q)\,;\rhd\,;U\\
= & \quad\{\![\text{ definition of }*\,]\!\}\\
 & (P*Q)\,;U\;.
\end{array}$$

The reverse inequation follows similarly from Corollary 13.3 and isotony. □

Finally we are able to prove Lemma 4.1.

Proof of Lemma 4.1.
First note that $\Delta P=P\,;U\cap\mathsf{skip}$. The same holds for Q. By this we calculate

$$\Delta P*\Delta Q=(P\,;U\cap\mathsf{skip})*(Q\,;U\cap\mathsf{skip})\subseteq(P\,;U)*(Q\,;U)=(P*Q)\,;U\;.$$

Moreover $\Delta P*\Delta Q\subseteq\mathsf{skip}$, since both are tests. Hence we can conclude $\Delta P*\Delta Q\subseteq(P*Q)\,;U\cap\mathsf{skip}=\Delta(P*Q)$.

The reverse inclusion was shown in Lemma 2.9. □