Developments in concurrent Kleene algebra

Tony Hoare, Stephan van Staden, Bernhard Möller, Georg Struth, Huibiao Zhu

Angaben zur Veröffentlichung / Publication details:

Hoare, Tony, Stephan van Staden, Bernhard Möller, Georg Struth, and Huibiao Zhu. 2016. "Developments in concurrent Kleene algebra." *Journal of Logical and Algebraic Methods in Programming* 85 (4): 617–36. https://doi.org/10.1016/j.jlamp.2015.09.012.

Developments in Concurrent Kleene Algebra

Tony Hoare[a], Stephan van Staden[b,1], Bernhard Möller[c], Georg Struth[d], Huibiao Zhu[e,2]

[a] *Microsoft Research, Cambridge, UK*
[b] *Department of Computer Science, University College London, UK*
[c] *Institut für Informatik, Universität Augsburg, Germany*
[d] *Department of Computer Science, The University of Sheffield, UK*
[e] *Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China*

Abstract

This report summarises the background and recent progress in the research of its co-authors. It is aimed at the construction of links between algebraic presentations of the principles of programming and the exploitation of concurrency in modern programming practice. The signature and laws of a Concurrent Kleene Algebra (CKA) largely overlap with those of a Regular Algebra, with the addition of concurrent composition and a few simple laws for it. They are re-interpreted here in application to computer programs. The inclusion relation for regular expressions is re-interpreted as a refinement ordering, which supports a stepwise contractual approach to software system design and to program debugging.

The laws are supported by a hierarchy of models, applicable and adaptable to a range of different purposes and to a range of different programming languages. The algebra is presented in three tiers. The bottom tier defines traces of program execution, represented as sets of events that have occurred in a particular run of a program; the middle tier defines a program as the set of traces of all its possible behaviours. The top tier introduces additional incomputable operators, which are useful for describing the desired or undesired properties of computer program behaviour. The final sections outline directions in which further research is needed.

Keywords: Concurrent Kleene Algebra, Laws of Programming, Trace Algebra, Semantic Models, Refinement, Unifying Theories

1. Introduction

Concurrency has many manifestations in computer system architecture of the present day. It is provided in networks of distributed systems and mobile phones on a world-wide scale; and on a microscopic scale, it is implemented in the multicore hardware of single computer chips. In addition to these differences of scale, there are many essential (and inessential) differences in detail. As in other areas of basic scientific research, we will postpone consideration of many interesting variations, and try to construct a mathematical model which captures the essence of concurrency at every scale and in all its variety.

Concurrency also has many manifestations in modern computer programming. It has been embedded into the structure of numerous new and experimental languages, and in languages for specialised applications, including hardware and network design. It is provided in more widely used general-purpose languages by a choice of thread packages and concurrency libraries. Further variation is introduced by a useful range of published concurrency design patterns, from which a software architect can select one that reconciles the needs of a particular application with the capabilities and performance of the particular hardware available for implementation. Our laws and principles will be illustrated

Email addresses: a-tohoar@microsoft.com (Tony Hoare), s.vanstaden@cs.ucl.ac.uk (Stephan van Staden), bernhard.moeller@informatik.uni-augsburg.de (Bernhard Möller), g.struth@sheffield.ac.uk (Georg Struth), hbzhu@sei.ecnu.edu.cn (Huibiao Zhu)

[1] Stephan van Staden was supported by the SNSF.

[2] Huibiao Zhu was supported by National Natural Science Foundation of China (No. 61361136002 and No. 61321064).

by examples drawn from many of these sources, and we will refrain from designing any particular language of our own.

Concurrency is also a pervasive phenomenon of the real world in which we live. Consequently, a general mathematical model of concurrency shares many concepts and principles with human understanding of the real world. For example, we model the behaviour of an object engaging together with other objects in events of various kinds that occur at various points in space and at various instants in time. We observe also the fundamental principle of causality, which states that no event can occur before an event on which it causally depends. Another principle is that of separation, which states that separate objects occupy separate regions of space. It is these principles that guide definitions of sequential and concurrent composition of programs in a model of Concurrent Kleene Algebra (CKA) [37].

These principles also provide evidence for a claim that CKA is an algebraic presentation of common-sense (non-metric) spatial and temporal reasoning about the real world. Thus the algebra may be used to define and explore the behaviour of a computer system embedded in its wider environment, including even human users. The investigation of space and time has long been the province of philosophers, including St. Thomas Aquinas, William of Ockham, and Aristotle. We would like to think of this work as a contribution to that tradition.

1.1. Application

The purpose of CKA is to support reliable predictions about the behaviour of computer programs when executed. It presents a set of very general algebraic laws, which are intended to be applicable to the execution of all computer programs that can ever be specified, designed, implemented in its notations. The free variables of the laws therefore stand for programs, but they are equally valid for program executions and program specifications. The operators of the algebra describe the way in which programs can be composed out of other smaller subprograms. The constants (or generators) of the algebra stand for the basic or atomic commands of the program. The algebraic laws express programming concepts and principles, which feature (under various notational disguises) in widely used computer programming languages of the present day.

Algebraic laws play a fundamental role in expressing the principles that underlie many branches of human knowledge. For example the laws of arithmetic embody the principles of numerical calculation, and the algebraic axioms of group theory serve as a definition of this branch of mathematics. Fundamental discoveries in the natural sciences are often expressed as laws, which later find application in engineering. Maxwell's equations of electro-magnetism are essential in the design of the electronic components of computers, just as Boole's Laws of thought are the basis of the design of computing circuits implemented electronically. The laws of programming should perhaps be recognised as providing a similar foundation for Computer Science and its application in Software Engineering.

Algebraic laws are expressed in a very limited subset of mathematical notations. They take the form of equations (or inequalities) between terms expressed by an explicitly restricted set of functions and constants. The variables in the laws are all (implicitly) quantified universally. In addition to the axioms of transitivity and reflection, the only required rules of reasoning are the instantiation of the variables and the substitution of terms already proven to be equal or to be related by an ordering. That is why computers and even humans are so good at algebraic reasoning and some humans even appreciate its sparse elegance.

It is remarkable how many important ideas of natural science and mathematics are definable within the limitations of algebra. The universality of algebra make it applicable not only to many different phenomena in our own universe, but also to many alternative universes. Consequently, there are many features and properties of our universe that cannot be expressed algebraically. Newton's laws of motion describe the orbits of the planets in general, but they do not predict the position in the sky at which any particular planet of our own solar system can be seen at any given time. For this, it is necessary to know the mass and momentum of the heavenly bodies, and their distance from each other. This information is given by Kepler's mathematical model of the planetary system. This may be illustrated by a working model, an orrery, perhaps made in brass and driven by cogs and clockwork instead of gravity and momentum.

In any conventional scientific theory, the relevant laws are supported by mathematical models of the behaviour of aspects of one or more parts of the physical universe, which have been shown by observation and experiment to satisfy the laws. Similarly, in mathematics, the Laws of Arithmetic are supported by discovery of a set-theoretic model of the numbers to which the laws apply. The achievement of research into the Foundations of Mathematics has been to discover this model, and to prove that it satisfies the laws.

1.2. A hierarchy of models

The standard model of arithmetic is constructed in a number of tiers. For example, the real numbers occupy the top tier, then the fractions and the integers, with the natural numbers at the bottom tier. Each tier is defined in terms of the next tier below it; for example, reals are defined as downward closed sets of fractions (Dedekind cuts), and fractions are defined as pairs of integers. The operators applicable to each kind of number are similarly defined in terms of the operators of the next tier. Numbers in the higher tiers have a larger range of operators defined on them, for example, exact division is available for fractions but not for integers. Nevertheless, for operators which they share, any two kinds of number are proved from each model to obey the same laws, or very similar ones. That is what justifies the use of the same notations for operators defined at each tier of the hierarchy.

We make a hierarchy of models for different kinds of program description. The top tier consists of specifications, which describe properties and behaviour of a computer system while executing a program. These may be desirable properties of a program that is not yet written, or undesirable properties of a program that is still under test.

Formally, a specification is modelled as a set containing program behaviours that satisfy the property. It may be expressed in any meaningful mathematical notation, including arbitrary set unions, and intersections, and even negation. Specifications may also include programming operators. For example, concurrent composition of two specifications describes the behaviour of the concurrent composition of two programs (threads), each of which satisfies the corresponding operand of the specification.

The extra expressive power of mathematics is needed to make specifications clearly comprehensible, so that they can be checked against the intended users' understanding of the relevant requirements. As in all branches of applied mathematics, such a formal description should be accompanied by informal prose, describing how the formal notations relate to the real world.

The middle tier of description consists of programs themselves. A program is modelled formally as a precise description of the exact range of all its own possible behaviours when executed by computer. It is expressed in a highly restricted notation, namely its programming language. The language excludes negation and other operators that are incomputable in the sense of Turing and Church. As a result, a program text can be input directly by computer, and (after various mechanised transformations, typically ending in machine instructions) it can be directly executed to cause the machine to produce just one of the behaviours specified. The selection of which execution to produce is determined, or at least influenced, by dynamic interactions (for example, input/output) with the immediate environment which surrounds the executing computer system. Inefficiency of implementation is another good reason for omission from a programming language of the more general operators useful in a specification. For example, intersection (logical conjunction) is the most useful operator for assembling large sets of design requirements into a single specification. It is available in declarative languages, including functional languages which are deterministic, and logic languages which are not. Their programs are structured as a conjunction of declarations of functions or predicates. However, intersection is usually excluded from a procedural language, because its implementation in general requires massive back-tracking.

In the model at the bottom tier, a single trace (such as that produced by a single program test) describes just one particular execution of a particular program at a particular time on a particular computer system or network. This execution itself is also modelled as a set that contains all the events that occurred during that execution, including events in the real world environment immediately surrounding the computer system.

Execution of each structural (syntactic) component of a program, perhaps a loop or a method body, consists of a subset of the events that occurred in execution of the complete program. We will call it a tracelet. After removal of a tracelet, the rest of the trace forms part of the environment within which the given syntactic component was executed. The operators of sequential and concurrent composition of two disjoint tracelets form a larger tracelet as the union of the events contained in execution of their two operands.

In summary, the free variables in the laws in all three tiers stand for descriptions of program behaviour. For specifications, there is no restriction on notation. For programs, the description must be given in a restricted computable notation. On the bottom tier, the program must not contain conditionals or any other form of choice, so that it necessarily describes only a single trace. In arithmetic, all the kinds of free variables of the laws are called simply numbers. We will follow this example, by using the single word 'program' in all three cases where the distinction does not matter.

1.3. Counter-examples, errors and contracts

Models play an essential role in the development of algebraic theories, and in the practical use of an algebra by mathematicians. They provide evidence (a counterexample) for the invalidity of an inaccurately formulated conjecture, explaining why it can never be proved from a given set of algebraic laws. In pure logical and algebraic research, such evidence is used to show the independence of each axiom of the algebra from all the others. In computer programming, a counterexample is a trace of execution of a test case, which demonstrates that behaviour of a program has deviated from the expectation of its author.

The definition of what counts as an error, and of where it is to be attributed, can be formalised as a contract at the interface between one part of the program and another. For each of its participants, a contract has two sides. One side is a description of the obligations, which any of the other participants may expect to be fulfilled. An example is the post-condition of a method body, which every call of the method will rely on to be true afterwards. The other side is a description of the requirements which each participant may expect of the behaviour of all the other participants taken together. An example is the precondition of the method body. Every calling program is required to make this true before the call, and the method body may rely on this as an assumption.

In addition to violation of contracts, there are various kinds of generic error, which are universally erroneous, independent of the purposes of the program. Examples familiar to sequential programmers are undefined operations, overflows, resource leakages and non-termination. Concurrency has introduced into programming several new classes of generic error, for example, deadlock, livelock, and races (interference). To deal with these errors, we need new kinds of contract, formulated in terms of new concepts such as dependence, resource sharing, ownership, and ownership transfer. A concurrent program can also call for dynamic interactions, perhaps by input or output with its surrounding environment.

A full formal definition (semantics) of a particular programming language will specify the range of generic errors which programs in the language are liable to commit. The semantics itself may be regarded as a kind of contract between the implementer and the user of the language. It will often allocate responsibility for errors that occur in a running program. For example, syntax errors and violations of type security are often avoided by compile-time checks, and the implementer undertakes to ensure that a program which contains any such errors will not be released for execution, even in a test.

Conversely, there are certain intractable errors which the programmer must accept the responsibility to avoid. In the case of an error occurring at run time, the language definition may state explicitly that the effect of execution is 'undefined'. Consequently, the implementer is freed of all responsibility for what does or does not happen after the error has occurred. For example, in the case of deadlock, it is usual that nothing more will happen. An even worse kind of error may make the program susceptible to malware attack, with totally unpredictable and highly unpleasant consequences.

The inclusion of contractual obligations in a model lends it an aspect of deontic logic, the logic of duty and assignment of blame. These concepts have no place in the normal pursuit of pure scientific knowledge. However, they play a vital role in engineering and commercial applications of the discoveries of science.

1.4. Semantic theories of programming

There are four well-known styles for formalising the definition of the meaning (semantics) of a programming language. They are denotational, algebraic, operational and deductive (originally called axiomatic). Each of the styles is useful in the design and exploitation of a different kind of software engineering tool in an Integrated Development Environment (IDE).

A mathematical model of the laws of programming plays the role of a *denotational* semantics (cf. [1]) of an abstract programming language. The denotation of each program component is a mathematical structure, which describes program behaviour at a suitable level of abstraction. In the earliest examples, the objects of the model were mathematical functions, mapping a list of input values to an output value. Later examples included concurrent behaviour, modelled as sets of traces of events. We follow the later examples, extending them to support the discovery and attribution of errors in a program. The denotational models therefore provide an appropriate conceptual basis for the design and implementation of program testing tools, including test case generators, test trace explorers, and error analysers.

The abstract laws themselves present an *algebraic* semantics (advocated, for example by Bergstra and his colleagues [9]) of an abstract programming language. Algebra is useful in all forms of reasoning about programs.

Algebraic proofs are often relatively simple for automatic discovery, and when constructed manually they may even be elegant. The most obvious application of algebra is to validate the transformation of a program into one with the same meaning, but with more efficient executions. An algebraic semantics is therefore a good theoretical foundation for program translators, synthesisers and optimisers.

The rules of an *operational* semantics in the style of Plotkin [62] and Milner [51] show how to derive, from the text of a program plus its input data, the individual steps of just a single execution of the program. This is exactly what any implementation of the language has to do. The operational rules thereby provide a specification of an abstract interpreter, which simulates the execution of the program step by step. Indirectly, they specify the intended function of a compiler for the language: it is to produce machine code whose direct execution will behave in the same way as the interpreter, only more quickly.

The *deductive* semantics, as formalised by Hoare [35], gives proof rules for constructing a proof that a program is correct. Correctness means that no possible execution of the program contains an error. Some of the errors, like an overflow, a race condition or a deadlock, are generic errors. Others are violations of some part of a contract, for example an assertion, written in the program itself. A deductive semantics is widely used as a theoretical basis for program verifiers, analysers and model checkers, whose function is to determine which parts of a program are correct, and find test cases which demonstrate the errors.

In summary, the semantics of a programming language needs to be presented in at least four different styles, to suit the needs of various kinds of software engineering tool. The mutual consistency of the presentations is essential to reinforce confidence in the reliability and correct interworking of the tools. In over thirty years of research, theoretical computer scientists have developed many methods of proving consistency between different pairs of styles. In this way it is now possible to specify and verify the interfaces between the numerous tools of a toolset, and present a proof of their mutual consistency; and to do so even before designing or implementing any of the individual tools.

An easy way to prove consistency of two different formalisations is to prove one of them on the assumption of the validity of the other. For example, Hoare, O'Hearn and Wehrman [70] describe how the laws of the algebraic semantics can be derived independently and rather simply from a denotational model. The rules of deductive semantics and of the operational semantics can be derived solely from the algebra, without any further appeal to the model.

This simple method for proving four-way consistency is based on recognition of the central role of the algebra. Ironically, the algebra is also the simplest method of presenting the semantics. This suggests that algebraic techniques may be the most appropriate for exploring, expressing, and exploiting the principles of programming.

2. The Laws of Programming

The laws of programming are an amalgam of laws obtained from many sources: Tarski's relation algebra (cf. [48]), Kleene's regular languages [42], the regular algebra axiomatised by Salomaa [66] or Conway [18], Kozen's Kleene algebras [43], process algebras [14, 51, 9], Pratt's action algebra [64, 44], and Concurrent Kleene Algebra as introduced by Hoare et al [37]. The pomset model [32, 63], in particular Gischer's axioms for the equational theory of pomsets [29], and shuffle algebra (cf. [11]) have also provided inspiration for the denotational models.

An earlier introduction to the laws for sequential programming has been written by Hoare et al. [36]. This was written for general computer scientists and professional software developers. It contains simple proofs that the laws are satisfied by a relational model of the possible states before and after program execution. Unfortunately, the relational model does not extend easily to concurrency. The models of CKA are therefore based on events (as in process algebra) rather than on machine states.

The purpose of this section is to list a comprehensive (but not complete) selection of the laws applicable to concurrent programming. The laws are motivated informally by describing their consequences and utility. The informal description of the operators gives the most general meaning of each of them, when applied to programs and specifications. The operators applicable to traces are only a subset of those applicable to programs, whereas those applicable to specifications are a superset.

2.1. The basic operators (signature) of the algebra

Constants. Skip (1) describes doing nothing, for example because the desired task has already been completed.

Bottom ($\perp$) is a contradictory specification (false). It also applies to a program containing an error (perhaps of syntax or of type) which the language requires the compiler to detect. As a result, the program will be prevented from running. It also applies to a trace which is physically impossible, for example because it requires an event to occur before an event which caused it.

Caution: Many authorities identify refinement as the converse of our ordering.

Top ($\top$) is a trace or program which contains a generic programming error, for example a null dereference, a race condition or a deadlock. As a specification it may be equated to the predicate true, which permits any behaviour or misbehaviour whatever.

Binary composition operators. Sequential composition ($p; q$) describes execution of both p and q, where p can finish before q starts. It is associative with unit 1; and it has $\perp$ as zero. This definition allows the common optimisation practice of changing the ordering of independent pairs of commands, even though they are separated by semicolon.

Concurrent composition ($p|q$) describes execution of both p and q , where p and q can start together, and can finish together. In between, they can interact with each other and with their common environment. The interactions may be by shared memory, or by a communication, which may be either synchronised or buffered. The operator is associative and commutative with unit 1; and it has $\perp$ as zero.

Refinement. The refinement relation $p \Rightarrow q$ is reflexive and transitive, i.e., a pre-order. It means that p is comparable to q in all relevant respects; and whenever p is possible, it is preferable to q.

The refinement ordering has $\perp$ as bottom. The two binary operators listed above are covariant (also called monotone or isotone) in both arguments. For example,

$$p \Rightarrow q \quad \text{implies that} \quad p; r \Rightarrow q; r.$$

For the engineer, the most important property of refinement is covariance. Refinement of r by r' is expected to mean that r' is better in all relevant respects, and for all purposes, and in all environments of use. Suppose that $p(r)$ is a program or other product containing a component r; and that this component is later replaced by a better component r'. Then it is expected that the resulting product $p(r')$ will be better than the original $p(r)$. If not, then p defines a circumstance in which r' is not better than r, which we have ruled out by our interpretation of 'better', which includes 'all environments of use'.

Exchange. Sequential and concurrent composition satisfy the following analogue of the exchange (or interchange) law of Category Theory, also known as subsumption [29] or subdistribution:

$$(p|q); (p'|q') \Rightarrow (p; p')|(q; q')$$

For further explanation of the exchange law, see section 2.3.

The following operators are not available for traces.

Choice. Choice ($p \cup q$) describes the execution of just one of p or q. The choice may be determined or influenced by the environment, or it may be left wholly indeterminate. The operator is associative, commutative and idempotent, with $\perp$ as unit. Choice admits distribution by both sequential and concurrent composition. Finally, it can be used to define the refinement relation:

$$p \Rightarrow q \quad \text{iff} \quad p \cup q = q.$$

This permits the inequational axioms and theorems of the algebra to be expressed more conventionally as equations.

Iteration. The sequential iteration p^* performs a finite sequential repetition of p, zero or more times. It is covariant, and is the least solution of the usual inductive equations:

$$1 \Rightarrow p^*, \quad p \Rightarrow p^*, \quad \text{and} \quad p^*; p^* \Rightarrow p^*.$$

Similarly, the concurrent iteration $p!$ performs a finite concurrent repetition of p, zero or more times.

The following operators are not available for programs.

Residuals. The weakest prespecification q -; r is the most general specification of a program q which can be executed before q in order to satisfy specification r. Dijkstra's weakest precondition [26] is a special case where r is required to be an assertion, and q is required to be a program. An assertion can be regarded as special case of a program. It is a set of memory states, where a memory state is represented by a trace of a concurrent assignment of constants to all the variables in the memory. The residual is covariant in r and contravariant in q. It cancels sequential composition (and vice-versa), but the cancellation is only approximate (one way or the other) in the refinement ordering:

$$(q \text{ -; } r) \,;\, q \Rightarrow r \quad \text{and} \quad p \Rightarrow (q \text{ -;} (p \,;\, q))).$$

Similar laws apply to the following two specification operators. The specification statement p ;- r (due to Back [7] and Morgan [54]) is the most general specification of a program q that can be executed after p in order to satisfy specification r. Concurrent composition also has a residual (p -| r). In separation logic it is denoted by -* , and is known as the magic wand [59].

Intersection. $p \cap q$ describes just those traces that are described by both p and q. It is a lattice operator.

2.2. Refinement

Refinement is a fundamental relation which is used in a wide range of circumstances.

For example, if p and q are specifications, $p \Rightarrow q$ means that p logically implies q; as a consequence, every trace that satisfies p also satisfies q. Consequently p places stronger constraints on an implementation, which can be more difficult to meet (even impossible, in the case when p is $\bot$). Logical implication is a fundamental concept in mathematics and in logic, so its importance in programming is not exceptional.

If p is a program and q is a specification, the refinement relation means that p meets the specification q, in the sense that everything that p might do is described by the specification q. In this case, p has only a subset of the traces of q, so its behaviour is more deterministic than q; it is therefore easier to predict and control. In the extreme, p may have no traces, indicating that it contains an error that is detected at compile time. The implementation is then responsible for inhibiting execution.

Refinement also holds between traces in the case that p is a specific implementation of q. For example, q may contain a concurrent composition, and p may describe its implementation by means of interleaving, as described in the next section. It also holds in the extreme cases, where p contains a violation of the principle of causality, or if q reveals an error like a race, which is attributed to a faulty program.

2.3. The exchange law: $(p|q); (p'|q') \Rightarrow (p; p')|(q; q')$

This law, which, in the context of concurrency theory, has been considered previously by Gischer [29], relates a concurrent composition to its permitted implementation(s) by interleaving. Inspection of the form of the law (see above) shows that the left hand side of the law describes a subset of the possible interleavings of the atomic actions from the two component threads $(p; p')$ and $(q; q')$ on the right hand side. This subset results from a scheduling decision that the two semicolons shown on the right hand side will be synchronised as the single semicolon on the left.

It is important to recognise that many of the scheduling decisions described above will be impossible, and so represented by $\bot$. For example, suppose all four operands of the exchange law are atomic, and that p is causally dependent on an event in q'. Then the left hand side of the law is impossible, though the right hand side is not. An implementation must choose an interleaving (if any) that respects dependenc. This kind of synchronisation is essential for reliable communication between concurrent threads.

The algorithm for finding a single interleaving of the concurrent operator on the right hand side uses the principle of 'divide-and conquer'. An interleaving of $(p|q)$ before the semicolon on the left hand side is computed by a recursive call, and an interleaving of $(p'|q')$ by another recursive call. The delivered result is just the sequential composition of the results of these two calls. Note that this algorithm preserves (as it should) the ordering of the sequential compositions on the right hand side.

To deal with cases in which there are only two or three operands involved, use can be made of the following frame laws. They can be proved immediately by substituting the common unit 1 for the operand(s) of the exchange law that are missing in the frame law.

1. $(p|q); q' \Rightarrow p|(q; q')$,
2. $p; (p'|q') \Rightarrow (p; p')|q'$,
3. $p; q' \Rightarrow p|q'$.

In the last of these laws, there is no concurrency operator on the left, and therefore no recursive call is needed. The final effect is to eliminate all concurrent compositions from the result of the algorithm.

An example derivation of a complete interleaving is shown below. To avoid clutter, most of the semicolons are suppressed, except those needed for application of the relevant frame or exchange law

$$\begin{aligned} abcd|xyzw &= (a; bcd)|(xy; zw) && \text{(assoc ;)} \\ &\Leftarrow (a|xy); (bcd|zw) && \text{(exchange)} \\ &= (a|x; y); (b; cd|zw) && \\ &\Leftarrow (a|x); y; (b|zw); cd && \text{(frame)} \\ &\Leftarrow xayzbwcd && \text{(frame)} \end{aligned}$$

At each step in this derivation, an associative or commutative law is applied in addition to a frame law. By making these choices in every possible way, it is possible to generate every possible interleaving of the original term. Consider, for example, the simplest case of concurrent composition $p|q$. The last frame law together with commutation yields only the two interleavings, $p; q$ and $q; p$. By the laws for choice, it is possible to conclude

$$p; q \cup q; p \Rightarrow p|q.$$

It is tempting to strengthen this refinement to an equation; but the result of doing so would be invalid. For example, when p and q are themselves sequential compositions, many more than two interleavings are possible. Even when p and q are atomic commands, the strengthening to an equation may be invalid. For example, the two commands may race with each other (conflict) when executed concurrently, so their concurrent execution is erroneous. However, their sequential execution in either order is perfectly correct. In other cases, p and q may be commands, like synchronised input and output, that can only be executed simultaneously.

2.4. Summary of algebraic structures

A CKA has previously been defined [37] as an algebraic structure $(K, \cup, ;, |, \bot, 1, ^*, !)$ such that $(K, \cup, ;, \bot, 1, ^*)$ is a Kleene algebra, $(K, \cup, |, \bot, 1, !)$ is a (multiplicatively) commutative Kleene algebra, and the exchange law holds between sequential and concurrent composition. In the context of CKA the unit of sequential composition and that of concurrent composition is the same. Kleene algebras have been axiomatised by Kozen [43]; they are additively idempotent semirings in which the star operation satisfies unfold and induction laws similar to those mentioned in Section 2.1. Commutative Kleene algebras have been studied by Conway and Pilling [18].

CKA without the two star operators has been studied by Gischer [29] in the context of partially ordered multisets (cf. Section 4.2) as well as by Bloom and Ésik [11]. Axiom systems for action algebras, which are Kleene algebras expanded with the residuals mentioned in Section 2.1, have been proposed by Pratt [64]. Action lattices, which are action algebras expanded by an operation of meet or intersection, have been investigated by Kozen [44].

CKA has also been studied in the context of quantales, which are complete lattices with a monoidal operation of composition that distributes over arbitrary suprema [37]. Concurrent quantales are complete lattices with a monoidal operation of sequential composition and a second commutative monoidal operation for concurrent composition, with units shared, in which the interchange law holds between the two compositions. In quantales, all monotone functions have least and greatest fixpoints and all functions that distribute over arbitrary suprema have upper adjoints. The sequential and concurrent iterations * and ! as well as the sequential and concurrent residuals mentioned in Section 2.1 can thus be defined explicitly in this setting. It follows that every concurrent quantale is a CKA. Most of the models of interest, including shuffle and pomset languages, form concurrent quantales.

3. Models of Trace Algebra

In this section we illustrate the ideas of Sect. 2 with various concrete models of traces and programs that are substantial generalisations of the "standard" model of CKA given in [37]. At the same time we present some general techniques for constructing such models.

3.1. A Trace Model

We start by assuming an infinite set EV, containing all events, i.e., occurrences of primitive actions, that are possible as a result of execution of any program. A trace of execution of a particular program records just those events which have occurred as a result of that execution. It also records which pairs of events have occurred sequentially, and which have occurred concurrently.

Definition 3.1 Assume a set EV of *events*. A *trace* is a triple (p, s, c) where $p \subseteq \mathrm{EV}$, $s, c \subseteq p \times p$, with c symmetric and $s \cap c = \emptyset$, and $\times$ is the usual Cartesian product operator. The relations s and c record which of the events in p have occurred in sequence or concurrently. The simplest trace is $1 =_{df} (\emptyset, \emptyset, \emptyset)$; it describes that nothing happens. A trace with just a single event e has the form $(\{e\}, 0, 0)$.

Note that we do not need to postulate that s is a transitive relation, because the associativity of sequential composition is not dependent on transitivity. We will further deal with this issue below.

Next, we endow traces with a refinement relation $\Rightarrow$.

Definition 3.2 [Refinement] $(p, s, c) \Rightarrow (p', s', c') \iff_{df} p = p' \wedge s' \subseteq s \wedge c \subseteq c'$.

Hence (p, s, c) refines (p', s', c') if both traces have the same events, but the left operand (more refined) has more pairs of events (those in s) separated by ; and less pairs (those in c) separated by $|$. Thus a trace encodes certain essential information about the structure of the term of which it is the trace, but forgets just enough to validate the laws that we want.

We have

$$(p, s, c) \Rightarrow 1 \iff 1 \Rightarrow (p, s, c) \iff (p, s, c) = 1 .$$

Corollary 3.3 *The relation $\Rightarrow$ is a partial order between traces.*

Let + denote the disjoint union of sets (both of single events or pairs of such, i.e., relations): the value of $S + T$ is $S \cup T$ if S and T are disjoint and undefined otherwise. We assume that $\times$ binds tighter than $\cup$, ; and $|$.

Definition 3.4 [Sequential and Concurrent Composition] Let $p, p' \subseteq \mathrm{EV}$ be disjoint.

$$\begin{array}{lll} (p, s, c) \,;\, (p', s', c') & =_{df} & (p + p', s + s' + p \times p', c + c') , \\ (p, s, c) \,|\, (p', s', c') & =_{df} & (p + p', s + s', c + c' + p \times p' + p' \times p) . \end{array}$$

Note that both $(p, s, c) ; (p', s', c')$ and $(p, s, c) | (p', s', c')$ are undefined if p and p' are not disjoint. In general there are many more traces (triples, as defined above) than there are terms of the trace algebra to denote them.

Theorem 3.5 (Laws of Trace Algebra)

1. *Both* ; *and* $|$ *are associative and have* 1 *as a shared unit. Moreover,* $|$ *is commutative.*
2. *The exchange law and therefore also the frame laws (see Sect. 2.3) hold.*

The proof is given in Appendix A.

The above definitions can be readily extended by adding a component that counts pairs separated by any desired additional operators of the programming language. The definition of refinement (3.2) can be made more abstract, by applying any covariant function to s and s' before testing the inclusion, and another function to c and c'. Suggestions for additional or alternative operators are made in section 3.3. Many of them have been incorporated in familiar programming languages and theories. Even if they are not yet widely used, the extra generality of multiple operators is algebraically interesting.

Call a trace *generated* if it can be obtained by a finite number of applications of sequential and concurrent composition, starting from 1 and single-event traces. For generated traces it can be shown by structural induction on the generation that s is transitive. Moreover, for generated traces the c component is redundant, since it can be calculated by the equation $c = p \times p - Id - s - s^{\smile}$, where Id is the identity relation and $s^{\smile}$ is the converse of s; the proof proceeds again by structural induction. For that reason, for generated traces one can use the simplified representation (p, s) and the simplified operators

$$\begin{array}{lll} (p, s) \,;'\, (p', s') & =_{df} & (p + p', s + s' + p \times p')\ , \\ (p, s) \,|'\, (p', s') & =_{df} & (p + p', s + s')\ . \end{array}$$

This coincides with Gischer's definition [29] of series-parallel pomsets, so that the sets of generated traces in our model and in Gischer's are isomorphic.

3.2. From Traces to Programs: Lifting

As mentioned in Sect 2, a general principle of Trace Algebra is to consider programs as sets of traces. To make this work in a uniform way, one has to give a general method for lifting operators from the level of traces to that of programs. We do this by extending the trace operators *pointwise* to programs.

Definition 3.6 Let U be the set of all traces. A *program* over U is a subset of U that is downward closed w.r.t. the refinement relation $\Rightarrow$. The function dc forms the downward closure of a set S of traces, i.e.,

$$dc(S) =_{df} \{t' \mid \exists\, t \in S : t' \Rightarrow t\}\ .$$

For a single element $t \in U$ we abbreviate $dc(\{t\})$ by $dc(t)$. Hence a program is a set P of traces with $P = dc(P)$.

If $\circ : U \times U \rightarrow U$ is a, possibly partial, binary operator on U then its *pointwise lifting* to programs P, P' is defined as

$$P \circ P' =_{df} dc(\{t \circ t' \mid t \in P, t' \in P' \text{ and } t \circ t' \text{ is defined}\})\ .$$

Two distinguished programs are $\bot =_{df} \emptyset$ and $\mathsf{skip} =_{df} dc(\{1\}) = \{1\}$.

The reason for requiring downward closure will become clear soon.

The idea of pointwise lifting, of course, makes sense only if also the laws for the trace-level operators lift to programs. While it is clear what equality means for programs, i.e., downward closed sets of traces, there are several ways to extend refinement to sets. We choose the following definition:

$$P \Rightarrow P' \iff_{df} \forall\, p \in P : \exists\, p' \in P' : p \Rightarrow p'\ . \qquad (1)$$

By this, a program P refines a specification P' if each of its traces refines a trace admitted by the specification. Downward closure implies that $\Rightarrow$ in fact coincides with inclusion $\subseteq$ between programs.

A sufficient condition for lifting an inequational law $P \Rightarrow P'$ from traces to programs is *linearity*, viz. that every variable occurs at most once on both sides of the law and that all variables in the left hand side P also occur in the right hand side P'. Examples are the frame and exchange laws. For equations a sufficient condition is *bilinearity*, meaning that both inequations that constitute an equation are linear. Examples are associativity, commutativity and neutrality. The main result is as follows.

Theorem 3.7 *If a linear law $p \Rightarrow p'$ holds for traces then it also holds when all variables in p, p' are replaced by variables for programs and the operators are interpreted as the liftings of the corresponding trace operators.*

We illustrate the gist of the proof for the case of the frame law $P \,;\, P' \Rightarrow P \mid P'$. Assume $r \in P \,;\, P'$. By the above definition there are $t \in P, t' \in P'$ such that $r \Rightarrow t \,;\, t'$. Since the frame law holds at the trace level, we have $t \,;\, t' \Rightarrow t \mid t'$. Moreover, $t \mid t'$ is in $dc(\{(t \mid t') \mid t \in P, t' \in P'\}) = P \mid P'$ and we are done.

The full proof for general preorders can be found in Appendix B; it shows clearly where the various subconditions of linearity enter. Related results were presented in [28, 33, 30] (see also [12] for a survey).

Corollary 3.8 (Laws of Trace Algebra for Programs) *The liftings of* ; *and* | *to programs are associative and have* 1 *as a shared unit. Moreover,* | *is commutative and the exchange law and therefore also the frame laws hold.*

There are further useful consequences of our definition of programs. The set $\mathcal{P}$ of all programs forms a complete lattice w.r.t. the inclusion ordering; it has been called the *Hoare power domain* in the theory of denotational semantics (e.g. [71, 49, 13]). The least element is the empty program $\bot = \emptyset$, while the greatest element is the program U consisting of all traces. Infimum and supremum coincide with intersection and union, since downward closed sets are also closed under these operations.

Therefore we can define (unbounded) choice between a set $Q \subseteq \mathcal{P}$ of programs as

$$\square Q =_{df} \bigcup Q$$

with binary choice as the special case

$$P \mathbin{\square} P' =_{df} P \cup P' .$$

The lifted versions of covariant trace operators are covariant again (see Lm. B.1 in Appendix B), but even distribute through arbitrary choices between programs.

Covariance of the lifted operators, together with completeness of the lattice of programs and the Tarski-Knaster fixed point theorem guarantees that recursion equations have least and greatest solutions. More precisely, let $f : \mathcal{P} \to \mathcal{P}$ be a covariant function. Then f has a least fixed point μf and a greatest fixed point νf, given by the following formulas:

$$\mu f = \bigcap \{P \mid f(P) \subseteq P\} , \qquad \nu f = \bigcup \{P \mid P \subseteq f(P)\} .$$

With our operator ; this can be used to define the Kleene star (see e.g. [18]), i.e., unbounded finite sequential iteration, of a program P as $P^* =_{df} \mu f_P$, where

$$f_P(X) =_{df} \mathsf{skip} \mathbin{\square} (P ; X) .$$

Since f_P, by the above remark, distributes through arbitrary choices between programs, it is even continuous and Kleene's fixed point theorem tells us that $P^* = \mu f_P$ has the iterative representation

$$P^* = \bigcup \{f_P^i(\emptyset) \mid i \in \mathbb{N}\} , \tag{2}$$

which transforms into the well known representation of star, viz.

$$P^* = \bigcup \{P^i \mid i \in \mathbb{N}\}$$

with $P^0 =_{df} \mathsf{skip}$ and $P^{i+1} =_{df} P ; P^i$.

Infinite iteration P^ω can be defined as the greatest fixed point νg_P where

$$g_P(X) =_{df} P ; X .$$

However, there is no representation of that similar to (2) above, because semicolon does not distribute through intersection; we only have the inequation

$$P^\omega \subseteq \bigcap \{P^i ; U \mid i \in \mathbb{N}\} .$$

To achieve equality, in general the iteration and intersection would need to be transfinite.

Sometimes it is convenient to work with an iteration operator that leaves it open whether the iteration is finite or infinite; this can be achieved by Back and von Wright's operator [8]

$$P^{\widehat{\omega}} =_{df} P^\omega \mathbin{\square} P^* ,$$

which is the greatest fixed point of the above function f_P.

Along the same lines, unbounded finite and infinite concurrent iteration of a program can be defined.

3.3. More Abstract Models

Over one and the same set of traces, several refinement relations may be defined. We have fixed one as a specimen in Sect. 3.1, but there are other interesting variants as discussed below. Therefore by a *model* we now mean a trace algebra together with a refinement relation. Since choice has been defined as the union of downward closed sets, that definition extends to all models as well. A model is regarded as an abstraction of another if it has a weaker refinement relation. This necessarily preserves all the algebraic axioms and theorems of the more concrete theory. Consider a weaker refinement relation $\Rightarrow'$ with $\Rightarrow \subseteq \Rightarrow'$. Then, trivially, every pair of traces p, p' with $p \Rightarrow p'$ also satisfies $p \Rightarrow' p'$, and hence all valid inequational laws of the form $E \Rightarrow E'$ with trace-valued expressions E, E' also are valid when $\Rightarrow$ is replaced by $\Rightarrow'$. However, this account does not apply to proof rules (equations with equational side-conditions). That is why Pratt gave a purely equational treatment of iteration [64].

We illustrate this technique by showing how to introduce least and greatest elements into an algebra of programs.

Example 3.9 Assume disjoint nonempty sets of traces L, H such that L (Low) and H (High) are downward and upward closed w.r.t. $\Rightarrow$, respectively. Let again U be the set of all traces and define

$$\Rightarrow' \;=_{df}\; L \times U \;\cup\; U \times H \cup \Rightarrow \;.$$

Clearly, $\Rightarrow'$ is a weakening of $\Rightarrow$.

When the resulting trace algebra (with $\Rightarrow'$ in place of $\Rightarrow$) is lifted to sets, L and H are least and greatest elements, respectively, in the resulting space of specifications and may therefore be denoted by the standard symbols $\perp$ and $\top$. □

Lemma 3.10 *The weakened relation* $\Rightarrow'$ *is a pre-order again.*

Proof. Reflexivity is immediate from $\Rightarrow \subseteq \Rightarrow'$. For transitivity we first distribute ; through $\cup$ in $\Rightarrow' ; \Rightarrow'$, which gives nine clauses. Then we use covariance of Cartesian product and universality of U to deal with five of the cases:

$$\begin{array}{ll}
L \times U \,;\, L \times U & \subseteq L \times U \,, \\
U \times H \,;\, U \times H & \subseteq U \times H \,, \\
L \times U \,;\, U \times H & \subseteq L \times H \subseteq L \times U \text{ (and also } \subseteq U \times H) \,, \\
L \times U \,; \Rightarrow & \subseteq L \times U \,, \\
\Rightarrow ; U \times H & \subseteq U \times H \,.
\end{array}$$

The following covers the remaining four cases.

$$\begin{array}{lll}
U \times H \,;\, L \times U = \emptyset & & \text{because } H \text{ and } L \text{ are disjoint} \,, \\
U \times H \,; \Rightarrow & \subseteq U \times H & \text{because } H \text{ is upward closed} \,, \\
\Rightarrow ; L \times U & \subseteq L \times U & \text{because } L \text{ is downward closed} \,, \\
\Rightarrow ; \Rightarrow & \subseteq \Rightarrow & \text{because} \Rightarrow \text{is transitive} \,.
\end{array}$$

□

Let now Par stand for the symmetric relation that holds between any pair of events that are allowed to occur on either side of concurrent composition, with violation attributed to the program. Let Seq stand for the relation which holds between any pair of events that can possibly appear on either side of a sequential composition, with violation attributed to the implementation of the language.

These two suppositions are embodied in the following definitions of L and H:

$$\begin{array}{lll}
L & =_{df} & dc(\{(p, s, c) \mid s \not\subseteq \mathsf{Seq}\}) \,, \\
H & =_{df} & uc(\{(p, s, c) \mid c \not\subseteq \mathsf{Par}\} - L) \,,
\end{array}$$

where *uc* denotes upward closure w.r.t. $\Rightarrow$.

In the remainder of this section, we will give some examples of possible meanings for the Seq and Par relations. They may be treated as alternatives, or they may introduce new and useful operators into the algebra.

Let B stand for a causal relationship that holds between two events in the same trace when the first of them has been a necessary cause for the occurrence of the other. It is an abbreviation of the ‘happens before’ relation used in the specification of memory models with relaxed consistency. We assume that it is an irreflexive and transitive relation (as it is in real life). Similarly, let S stand for the relation that holds between two events that correspond to actions of the same object; it is an equivalence relation. With these, we formulate and discuss several variants of Seq and Par.

1. $\mathsf{Seq} = \mathsf{B}$. This gives a common definition of a strict sequential composition, which requires that all the events of the first operand must occur before any events of the second operand. If both operands are collections of concurrent threads, implementation of this definition requires a barrier synchronisation. This is the basis of the PRAM model of computational complexity [69].
2. $\mathsf{Seq}' = \overline{\mathsf{B}^{\smile}}$. This gives the definition of a relaxed sequential composition. It permits an implementation to execute independent events in either order, when this is not prevented by causality. This practice is almost universal in language implementations of the present day. The strict definition is still useful, because it ensures that an if condition of a conditional command is evaluated before any of the events that appear in the then clause or the else clause between which it selects.
3. $\mathsf{Seq}'' = \mathsf{Par} - \mathsf{B}^{\smile}$. This may be executed either sequentially or concurrently. Communication is allowed only from the first operand to the second. It is provided in CSP by the chaining operator $\gg$. It guarantees absence of deadlock, and provides the basic design paradigm for systolic algorithms, implemented in hardware or in software [45, 16].
4. $\mathsf{Par} = \overline{\mathsf{S}}$. This means that no object can be shared between concurrent threads. This is the standard rule for concurrency in the occam programming language [47], widely implemented on transputers.
5. $\mathsf{Par}' = \overline{\mathsf{B} \cap \mathsf{B}^{\smile}}$. This forbids a causal cycle to cross the boundary between the events of two threads. Such a cycle is a programming error, and therefore permits an implementation to stop execution (which is in fact what happens naturally). The phenomenon is widely known as ‘deadlock’ or ‘hang’.
6. $\mathsf{Par}'' = \mathsf{Seq}'' \cap \mathsf{Seq}''^{\smile}$. This defines any communication or interaction between the operands to be a programming error. It can be implemented on two disconnected processors, or by arbitrary interleaving. It is denoted in CSP by the interleaving operator $|||$.

This list is just a sample of useful operators that are simply definable from two simple relations, S and B. Further operators may be defined, possibly with the aid of further primitive semantic relations between events. The fact that new operators can be introduced late in the design of a programming language is evidence for the modularity of our method of language design. However there are operators that cannot be introduced in this way. For example, all the above operators share the same unit, whereas the important external choice operator $\square$ of CSP (or + in CCS) has a different unit (STOP in CSP, or 0 in CCS). We have been unable to find a neat definition of this operator. Further thoughts on this are presented in Sect. 4.3.

3.4. Specifications

In this section we give more details about the specification constructs mentioned in Sect. 2.1.

Intersection is mathematically simple, since the intersection $\bigcap\limits_{i \in I} P_i$ of a family $(P_i)_{i \in I}$ of downward closed sets is automatically downward closed and hence a program again. However, this construct is not feasibly implemented, not even in its binary variant $P \cap P'$.

As for the residuals, their existence is guaranteed by the distributivity of lifted covariant operators and completeness of the lattice of downward closed programs. They can be defined by the Galois connections

$$\begin{array}{lll} p \Rightarrow q \mathrel{-;} r & \text{iff} & p \,;\, q \Rightarrow r\,, \\ q \Rightarrow p \mathrel{;-} r & \text{iff} & p \,;\, q \Rightarrow r\,. \end{array}$$

This independent characterisation is necessary, since these operators cannot reasonably be defined as the liftings of corresponding ones at the trace level. An analogous definition can be given for the magic wand $\mathrel{-|}$. The semi-cancellation laws of Sect. 2.1 are immediate consequences of these definitions.

Residuals enjoy many more useful properties, but we forego the details.

3.4.1. Action Algebras and Kleene Algebras

Instead of defining the Kleene star via fixed point theory one may give an axiomatisation. Whereas the traditional axioms for star use conditional rules to express the minimality of the fixed point, Pratt in his *action algebra* gives purely inequational axioms [64] in terms of residuals. His theory has later been refined into *action lattices* in [44]. The advantage is that the axioms are valid in any homomorphic image of the algebra, whereas conditional rules are not. Here are the axioms of the star operator, in addition to the semiring axioms, with the convention that ; binds tighter than ▯ :

$$\begin{gathered} 1 \;\square\; a \;\square\; (a^* \,;\, a^*) \;\Rightarrow\; a^* \,, \\ a^* \;\Rightarrow\; (a \;\square\; b)^* \,, \\ (x \;\text{;-}\; x)^* = x \;\text{;-}\; x \,, \\ (x \;\text{-;}\; x)^* = x \;\text{-;}\; x \,. \end{gathered}$$

With Prover9/Mace4 [50] one can show that the second axiom (covariance of star) cannot be omitted.

The last two axioms can be motivated intuitively as follows. According to Sect. 2.1, $x \;\text{;-}\; x$ is the most general program p for which x is an invariant. But then x is also an invariant for p^* which means that $(x \;\text{;-}\; x)^* \Rightarrow x \;\text{;-}\; x$ should hold. The reverse inequation follows from the first axiom of action algebra.

Adding the analogous axioms with | operator gives the arbitrary finite iteration of a program using concurrent composition. Therefore both iteration operators enjoy all the properties well known from the theory of regular expressions over a given alphabet.

4. Further Developments

4.1. CKA-Based Concurrency Verification Tools

We have started to design and implement lightweight tools for program verification and correctness based on algebras of programs which can be prototyped rapidly and effectively within the Isabelle/HOL proof assistant [58]. At the moment our tools support the verification and refinement of sequential programs and of concurrent programs within the rely/guarantee method [17]. Another tool supports program verification with separation logic [15].

Our approach benefits from Isabelle's support for engineering algebraic hierarchies and their models, and its emphasis on proof automation through the integration of automated theorem proving and counterexample search technology, which can deal with algebraic proofs efficiently. This and our own libraries for variants of Kleene algebras, including Kleene algebras with tests, action algebras, quantales, demonic refinement algebras, modal Kleene algebras, CKA and Tarski's relation algebras and their most important models [5, 34, 4, 3] make the approach simple and modular. In addition, Isabelle provides large libraries for data structures and their properties. Program construction and verification with our tools is therefore well supported and feasible at least for educational purposes and research.

The main principle behind our approach to tool design is to use variants of Kleene algebras as an abstract semantics for the control flow of programs. The data flow, which appears in assignment statements, tests and assertions, is cleanly separated from this layer. It can be captured within appropriate models such as relations or predicate transformers for sequential programs or traces and pomsets for concurrent ones. At the data-flow level we can link once more into Isabelle's extensive libraries for states, stores and data structures. The two layers are related through a soundness proof in Isabelle. Abstract theorems are then picked up automatically by the tool to reason in the concrete model.

Based on this principle we have implemented a verification tool for while-programs based on Kleene algebras with tests and extended it to a program construction/refinement tool [6] by adding one single algebraic axiom.

We have also developed a rely/guarantee-style concurrency verification tool based on bi-Kleene algebras with sequential and concurrent composition operations and CKA [2] with algebraic axioms for interference constraints. We are currently implementing more refined versions with an algebra for infinite transition traces, cyclic inference rules and inductive assertions, as needed for verification purposes (cf. [31]).

Finally, we have implemented a separation logic tool based on predicate transformers, as given by modal Kleene algebra [25], and an assertion quantale in which separating conjunction is modelled algebraically as an operation of convolution [27]. Modal box operators over this assertion quantale then correspond to predicate transformers: in the formula $[R]P$, the function $\lambda x.[R]$ is the predicate transformer associated to relation R applied to assertion P. This new approach is inspired by the state transformer semantics of abstract separation logic [15]. It yields an instance of the more general and complex category-theoretic treatment in the logic of bunched implications [59] that is particularly suitable for implementation in tools like Isabelle.

In all our verification approaches, verification conditions are generated automatically from Hoare logics, which can be derived generically from the algebraic layer, except for assignment rules, which must be justified within the data flow semantics. In particular, the frame rule of separation logic and the concurrency rules of rely/guarantee can be derived under suitable algebraic assumptions.

An integration into CKA-based verification tools for concurrent programs, which combine rely/guarantee with separation logic, is the next step in the tool chain. These are intended to support the verification of lock-free and wait-free data structures or multi-core programs.

In parallel with tool development, we have conducted algorithmic cases studies on program construction and verification with Isabelle. Links to the complete Isabelle formalisation can be found in the corresponding articles [6, 2, 27]. While correctness proofs for simple sequential algorithms and basic separation logic examples work quite smoothly, the verification of concurrent algorithms requires further optimisation.

4.2. Completeness, Decidability and Complexity

Algebraic reasoning about programs becomes even more effective when decidable fragments are identified and decision procedures are integrated into verification tools. The equational theories of Kleene algebra and (multiplicatively) commutative Kleene algebra, for instance, are known to be decidable [43, 18]. Verification applications often require reasoning under assumptions, which in Kleene algebra is generally undecidable; but interesting decidable subclasses have been identified. The decision procedure for Kleene algebra, in particular, is based on a soundness and completeness result [43] according to which an universally quantified equation, or identity, is derivable from the Kleene algebra axioms if and only if the two terms in the identity denote the same regular language. The operations of Kleene algebra are, of course, the regular operations and Kleene algebra terms correspond to regular expressions.

Concurrent Kleene algebras are both Kleene algebras and commutative Kleene algebras; they share the operations of addition, additive unit and multiplicative unit, and satisfy the exchange law. In this case completeness and decidability remains open. Candidate structures are pomset languages or sets of pomsets [32, 64]. These form a standard model of concurrency. They arise from labelled partial orders with vertices corresponding to events, labels to actions that occur at events and the order modelling the causal dependencies between events. Pomsets form isomorphism classes of labelled partial orders in which the names of events have been forgotten.

The following partial results are known (cf. [46]). First, pomset languages form bi-Kleene algebras (CKAs without the exchange axiom). This implies the soundness result that every identity that can be derived from the bi-Kleene algebra axioms holds in the pomset language model.

Second, Laurence and Struth have shown that bi-Kleene algebras are sound and complete with respect to the equational theory of the so-called series-parallel rational pomset languages [46]. Hence an identity is derivable from the bi-Kleene algebra axioms if and only if the two terms in that identity denote the same language in that class. It follows that the equational theory of bi-Kleene algebras is decidable, but the precise complexity remains unknown.

Third, Grabowski and Gischer [32, 29] have defined an order on pomsets according to which smaller pomsets are more sequential. Gischer has shown that pomset languages that are downward closed with respect to that order form CKAs, which implies soundness relative to CKA identities. The constructions are similar to those in Section 3.

Fourth, and finally, Gischer [29] has proved completeness of CKA identities without the sequential and concurrent star relative to the so-called downward closed series-parallel pomset languages.

We conjecture that CKA is sound and complete with respect to the equational theory of downward closed series-parallel rational pomset languages and that CKA identities are decidable due to this result. This result would also establish the downward closed series-parallel rational pomset languages as the free algebras in the variety of concurrent Kleene algebras. The decision procedure is of particular interest for automated reasoning with CKA in concurrency verification tools, as outlined in Section 4.1.

4.3. External Choice

In the process calculus CSP, the choice operator $\cup$ is called internal choice, and is denoted by $\sqcap$. The intention is that the choice is made autonomously and at any time during the execution of the program, or even at compile time. The choice cannot be influenced in any way by any other part of the program. That is why the choice is sometimes called demonic: the programmer cannot complain if an erroneous alternative is always selected in preference to a correct one.

Various process algebras have introduced a different choice operator, known as external choice, because it can be influenced or even decided by a thread running concurrently with the choice. It is denoted by $\Box$ in CSP and by $+$ in CCS. The following laws are quoted from the algebra of CSP (most of them are bisimulations in CCS). Like the internal choice, $\Box$ is associative, commutative and idempotent. It has a unit STOP , which stands for deadlock. It distributes through $\cup$, and $\cup$ distributes through it. Sequential composition distributes backwards through $\Box$, but not forwards.

The operands of external choice are guarded commands (denoted by $g \rightarrow C$ in CSP or $g.C$ in CCS). The guard g may cause a delay in execution of the command C, until the guard is satisfied. A typical guard is an input or an output command. An input guard is satisfied by simultaneous or previous execution of an output guard, which sends the required message from some other thread. If more than one guard of an external choice is satisfied, it becomes internally non-deterministic which of the guarded commands will be executed.

Following the example of ACP, let us introduce an associative and commutative operator $*$, to be applied to guards. The result of $g * g'$ is either another guard, or an atomic action which is performed as the first action of the chosen alternative. The result may also be STOP (indicating that the two guards do not match), or SKIP (indicating that the match succeeds, but has no effect). For these last two cases we have the laws (STOP $\rightarrow C$) = STOP, stating that nothing happens after deadlock, and $C \Rightarrow$ (SKIP $\rightarrow C)\Box D$, stating that C is one of the ways of implementing the external choice and we don't have to wait for the guard of D to be satisfied before making the choice to do C straight away. As in the case of the other algebraically defined operators, the choice of the meaning of the matching operator $*$ is left open, and different programming languages will make different choices.

The final and most important law is called an expansion law; it describes how concurrent composition distributes through $\Box$. Consider, for example, the term

$$(g \rightarrow C \,\Box\, g' \rightarrow C') \parallel (h \rightarrow D \,\Box\, h' \rightarrow D')$$

According to the expansion law, this is equal to

$$g * h \rightarrow (C \parallel D) \,\Box\, g * h' \rightarrow (C \parallel D') \,\Box\, g' * h \rightarrow (C' \parallel D) \,\Box\, g' * h' \rightarrow (C' \parallel D')$$

In many common cases, the expansion is reduced because most of the pairs of guards do not match. The unit law for STOP then causes the whole clause to disappear

It has been shown [65] that the laws of CSP are strong enough to reduce every term of the language into a head normal form, which does not contain concurrent composition. In this form, $\Box$ is the outermost operator, $\cup$ is next, and the innermost operands are either SKIP or STOP or a basic command sequentially composed with the rest of the program which remains to be executed. This too can be reduced to the head normal form. This reduction essentially gives a denotational semantics to the programming language, in which the model is a recursively defined higher-order function, which maps a set of initial events or guards to a set of similar functions. More formally, the denotational semantics of the language is the solution of a recursive domain equation, as suggested initially by Scott. This form of semantics can also be translated directly into an operational semantics in the style of Milner. However, this model is not so useful for program debugging as the trace model.

4.4. Three Event-driven Languages

This section reports the application of algebra in three event-driven concurrent programming languages (Verilog, SystemC and PTSC) [72, 74, 73, 76]. They include all three operators of CKA. They also include an external choice operator. They all have explicit control of relative timing, either real or simulated, and synchronization through simultaneous execution of guards in different threads (processes). Interaction between processes is mainly achieved by sharing of memory. The model of concurrency is that of interleaving of instantaneous (atomic) actions from all the threads.

Verilog is a Hardware Description Language that has been widely used in industry. It became an IEEE standard in 1995 as IEEE Standard 1364-1995 [39] and was revised in 2001 as IEEE Standard 1364-2001 [40]. The guards can be combined by the Boolean operations of conjunction, disjunction and even negation. Also, a guard may be placed on any shared variable of the program, which will cause the guard to be activated on an increase or a decrease in the value of that variable. Guard $@(\uparrow v)$ is triggered by the increase of the value of v, whereas $@(\downarrow v)$ is triggered by a decrease in v. Any change of v awakens the guard $@(v)$.

SystemC [60] is a system-level hardware modelling language which can be used to model a system at different abstract levels. It possesses several new and interesting features [73], including delayed notifications, notification cancelling, notification overriding and delta-cycle. There are three kinds of event notifications: immediate event notifications, delta-cycle delayed notifications and timed notifications. Delayed notifications can be cancelled via cancel statements before they are triggered. The current delayed notification on an event may override the previous delayed notification on the same event. Events can also be generated due to all pending channel update requests. A process may wait for the arrival of an event. A guard can be fired by event notification or generation.

Our third language PTSC [75] was designed as an experimental system specification language, combining probability, time and shared-variable concurrency. Probability is reflected by probabilistic nondeterministic choice, probabilistic guarded choice and probabilistic scheduling of actions from different concurrent components in a program. The scheduling probability for instantaneous actions is taken into account when doing parallel expansion. The probability feature of PTSC focuses on the selecting (or scheduling) probability of components (or actions).

The concept of head normal form has been applied in deriving the operational semantics. If the head normal form of a process belongs to one of the several types of guarded choice, its transition rules can be defined as the transition rules for the corresponding type of guarded choice. Based on the derivation strategy, a set of transition rules is derived, which can be considered as a transition system (i.e., operational semantics). The derivation strategy is proved to be equivalent with our transition system, which shows that our transition system is complete with respect to the derivation strategy.

In the next step, we would like to explore the algebraic semantics for a large variety of domain specific languages, investigating the common laws for these languages by applying the associative and commutative operator "*" introduced in Section 4.3. Meanwhile, we would also like to explore the verification rules (Hoare Logic) for these languages, with the aim to prove the soundness of these rules by using our achieved algebraic laws.

4.5. Program Testing with Labelled Graphs

The purpose of our family of trace models is to provide a theoretical framework to assist in program testing. Such a framework is essential to the design of a tool that displays to the programmer a trace of execution of a program, particularly an erroneous one. As in the standard models of Kleene algebra, the trace can be pictured as a labelled graph, in which the nodes stand for the events that occurred in the execution, and the arrows indicate a causal relationship between the events. An erroneous event should be indicated, for example by colouring it red for a program error (top), or blue for an environment error (bottom).

The tool should have a zooming capability to permit chosen threads and objects to be represented by a single line, and method calls to be reduced to a single point. It should also provide immediate navigation from an event node in the display to the command in the program whose execution called for occurrence of the given event. To help in tracing possible causes of an error, we need navigation forward and backward along the dependenc arrows connected to a given node. Backward navigation from a node which detects an error will permit a scan of all the places at which a change to the program may prevent the error from happening again. Forward navigation assists in checking that a change does not immediately introduce another error. It also helps to detect a false alarm, when there is no actual harm observed after the detection of a supposed error. In this surprisingly common case, it is the detecting event that needs correction.

The design and implementation of a useful tool for display of traces is left as a useful and important challenge for research.

4.6. An Interface Model: Graphlets, Hoare Triples and Modal Operators

Sometimes, in programming theory, it is interesting to reason about things like states or sets of such. We will show how this might be done in an additional model of CKA, the *interface model* presented in [52]; we sketch the main results of that paper.

Every dependence relation, such as the ones discussed in Sect. 3.3, defines a graph with events as nodes and arrows corresponding to dependences like causal or temporal succession and the like. The basic idea of the interface model is to consider subgraphs, called *graphlets*, of that overall graph as events of their own and make them into a CKA by defining sequential and concurrent composition suitably. The sets of incoming and outgoing arrows of a graphlet constitute its *input/output interface*.

If one endows the arrows with individual identity (and has them not just record existence of a dependence), a set of arrows can, for instance, be viewed as describing a set of variable/value associations. So, for the purpose of this section, let us call a set of arrows a *state* (provided certain healthiness conditions hold, such as functionality of these associations, or the like) and a set of states a *predicate*. The *pre-/post-restriction* of program Q to a predicate S retains only those graphlets in Q whose input/output state is contained S, respectively. Now a *standard Hoare triple* $\{S\}\,Q\,\{S'\}$ with a program Q and predicates S, S' is defined to mean that the pre-restriction of Q to S is contained in the post-restriction of Q to S'. Hence if Q is started in an input state in S it is guaranteed that all corresponding output states are in S'. See [23] for a closely related early relational definition. This way the well known Hoare calculus with all its inference rules results.

We briefly link this to another, more recent view of Hoare triples [37]: for programs P, P' and Q one sets $P\;\{\!|Q|\!\}\;P' \iff_{df} P\,;\,Q \subseteq P'$. This says that, after any graphlet in "pre-history" P, execution of Q guarantees an overall graphlet in P'. For the case of programs as relations between states this definition appears already in [68]. These new triples can be defined in any ordered monoid and enjoy many pleasant properties; see again [37] for details. Let now US and US' be the post-restrictions of the universal program U to S and S', respectively. Then roughly $\{S\}\,Q\,\{S'\} \iff US\,;\,Q \subseteq US'$ (in fact, with a slightly strengthened variant of $;$), i.e., a pre-history with an output state in S followed by Q leads to an overall history with an output state in S'. Dually, using pre-restrictions of U gives a connection between standard Hoare triples and the analogous variant of Milner triples, see [38].

There is another interesting connection. Lifting the *in* operator from graphlets to programs we obtain a function that computes an "enabledness" predicate and satisfies the characteristic property of an abstract domain operator as known from modal semirings [24], namely that $in(Q)$ is the least preserver of program Q under pre-restriction. Based on this, one obtains diamond and box operators $|Q\rangle$ and $|Q]$ in the standard way. The latter corresponds to Dijkstra's wlp operator, all of whose laws can be derived algebraically from these definitions. For the relational case analogous definitions appear already in [10, 67]. The topic of modal operators in CKA is also prepared by the recent model of [41] in which non-trivial test elements exist.

Finally, since the interface model has a quantale structure, the results of [53] can be used to define variants of the temporal logics CTL^* and CTL suitable for the description and analysis of (quasi-)sequential subthreads of events. In the case of CTL the above modal operators can be usefully employed in this.

It will be the topic of further research to expand these connections. In particular, one should use the above ideas about temporal logics to develop a spatial logic in terms of concurrent composition and finally a combination into a spatial-temporal one.

4.7. Relaxed Memory

Program execution on modern multicore processors is often accelerated by weakening the assumption of a sequentially consistent shared memory. Programmers are then forced to deal with increasingly complicated relaxed (or weak) memory models. For example, if a thread writes to a location, then this update need not be immediately visible to all the other threads. This decoupling of threads can speed up execution, but programmers now need to insert explicit barrier instructions in programs where stronger synchronisation is necessary.

The abstract models we have constructed of CKA are based on events, and make no mention of memory. They are therefore equally applicable to relaxed memory as well as strict memory. For example, it seems possible to construct a non-deterministic memory, which stores in each location a set of all values that have not yet been overwritten by explicit barriers. A read instruction may fetch any value currently in this set.

In our event-based model, we represent memory as a subset (antichain) of the dependence relation, containing dependencies between the events of writing to and reading from each of the objects in the memory. Dependency is defined with the aid of a bottom trace ($\perp$) to state the impossibility of events occurring in the causally inconsistent order. Our dependence relation is the same as the 'happens-before' relation, familiar from current relaxed memory models. It will be an interesting challenge to check whether other relations needed for these relaxed models can also be given an algebraic interpretation, perhaps by introducing new constants to distinguish them.

In our approach, sequentially consistent memory is an abstraction implemented by particular patterns of barriers. In general, execution of the barriers is orders of magnitude slower than the actual fetch and store instructions. This often makes sequential consistency an unaffordable abstraction, and other more efficient abstractions may be sought.

These abstractions are embodied in collections of concurrent design patterns, designed to support collaboration between threads. For example, there are communication design patterns which simulate in main memory a network of channels of various kinds, perhaps multiplexed or simplex, buffered or synchronised. These patterns are then implemented by cleverly selected barriers, which may be significantly more efficient than the barriers for simulation of sequentially consistent memory.

The correctness of the program then depends on all threads maintaining the protocols that support the abstraction. We have introduced a technique for introducing a special top element into our models, which may be used in programming tools to check that all threads actually observe the protocols of the chosen design pattern.

We conjecture that the algebraic approach, as outlined in this paper, may help programmers to harness the power of relaxed memory by supporting simplified reasoning in many cases. Algebra can capture relationships between programming operators that transcend particular hardware architectures. For example, the laws of CKA are not tightly coupled to a particular memory model, yet they can be used to design and refine programs from specifications.

We expect that, as in many other disciplines, abstraction will be important for the successful understanding and practical application of relaxed memory concurrency. Abstractions can be constructed in a stratified way where different laws hold at different levels. Each of the aforementioned levels, i.e. the trace, program and specification levels, may further be refined into sublevels. Moreover, not all vendors need to conform to all the laws. Algebraic laws can serve to describe and explore the differences and commonalities between them. A practical framework would also allow programmers to exploit lower-level properties of a particular memory model whenever the need arises.

4.8. Further Applications

We have already mentioned connections between CKA and (Concurrent) Separation Logic (SL). An important goal is to couple our algebraic formulations of the latter in modal semirings (or the special case of quantales) [19, 20, 21] with the models presented in Sects. 3 and 4.6. This should allow algebraic derivations of actual concurrent algorithms, for instance concurrent garbage collectors as in [61] and, more general, of concurrent algorithms with complex data structures under transfer of ownership.

Another investigation should examine how the algebraic treatment of Jones's rely/guarantee calculus given in [37] works out in the models in Sect. 3 and 4.6. It will be interesting to see what invariants in the sense of [37] look like and whether they admit a characterisation of model aspects similar to that of the dependence relation in that paper.

In [22] we have used ideas from CKA and separation logic to derive a modal algebra for Petri nets. It turns out that sets of place markings form a separation algebra, and that operators of sequential and concurrent composition can be defined that satisfy a reverse exchange law in which the order of refinement is reversed. The induced modal algebra is used to prove the correctness a simple mutex net, including liveness and fairness aspects.

As additional further research, we plan to employ CKA in a project dealing with a special variant of multiagent systems. Agents are rational, autonomous functional processes that interact with their environment. Multi-agent systems consist of sets of such rational agents that individually and by interaction together attain a common goal or exhibit an otherwise desired behaviour. Specifically, the project will concern the task of learning specifications of single agents and agent systems from observations of environment changes and, subsequently, to discover interfering processes [56]. An earlier implementation that shall serve as a starting point is a relational interpretation of multi-agent systems [57]. First steps towards formalising and conforming ensemble learning approaches to our setting were published in [55] We hope to achieve formal specifications and perhaps even derivations of at least parts of the planned architecture, and that this will lead to further insights into the role of CKA as well as new and useful algebraic laws.

Appendix A. Proofs for Th. 3.5

We use associativity and commutativity of + without explicit mention.

Moreover, to abbreviate the calculations we define for sets $p, q \subseteq \mathrm{EV}$,

$$p \mathbin{\widehat{\times}} q \;=_{df}\; p \times q + q \times p \ .$$

It is clear that $\widehat{\times}$ is commutative and distributes through + in both arguments. With its help the definition of | abbreviates to

$$(p, s, c) \mid (p', s', c') \;=\; (p + p', s + s', c + c' + p \mathbin{\widehat{\times}} p') \ .$$

Associativity

We first note that the terms at the left and right hand sides of the associativity laws are either both undefined or both defined. Now we treat ; , assuming that both sides are defined; | is analogous.

$$
\begin{array}{cl}
 & ((p,s,c)\,;(p',s',c'))\,;(p'',s'',c'') \\
= & \quad \llbracket \text{ By Def. 3.4 } \rrbracket \\
 & (p+p',\ s+s'+p\times p',\ c+c')\,;(p'',s'',c'') \\
= & \quad \llbracket \text{ By Def. 3.4 } \rrbracket \\
 & (p+p'+p'',\ s+s'+s''+p\times p'+(p+p')\times p'',\ c+c'+c'') \\
= & \quad \llbracket \text{ distributivity } \rrbracket \\
 & (p+p'+p'',\ s+s'+s''+p\times p'+p\times p''+p'\times p'',\ c+c'+c'') \\
= & \quad \llbracket \text{ distributivity } \rrbracket \\
 & (p+p'+p'',\ s+s'+s''+p\times(p'+p'')+p'\times p'',\ c+c'+c'') \\
= & \quad \llbracket \text{ By Def. 3.4 } \rrbracket \\
 & (p,s,c)\,;(p'+p'',\ s'+s''+p'\times p'',\ c'+c'') \\
= & \quad \llbracket \text{ By Def. 3.4 } \rrbracket \\
 & (p,s,c)\,;((p',s',c')\,;(p'',s'',c''))\ .
\end{array}
$$

Commutativity of | *and Neutrality of* 1

These are straightforward from Def. 3.4 and Def. 3.1.

The Exchange Law

Again, the terms at the left and right hand sides of the law are either both undefined or both defined. We spell out both terms according to Def. 3.4 and rearrange using associativity and commutativity of +. First,

$$
\begin{array}{cl}
 & ((p,s,c)\mid(p',s',c'))\,;((q,r,d)\mid(q',r',d')) \\
= & (p+p',\ s+s',\ c+c'+p\widehat{\times}p')\,;(q+q',\ r+r',\ d+d'+q\widehat{\times}q') \\
= & (p+p'+q+q',\ s+s'+r+r'+(p+p')\times(q+q'), \\
 & \ c+c'+d+d'+p\widehat{\times}p'+q\widehat{\times}q')\ .
\end{array}
$$

Second,

$$
\begin{array}{cl}
 & ((p,s,c)\,;(q,r,d))\mid((p',s',c')\,;(q',r',d')) \\
= & (p+q,\ s+r+p\times q,\ c+d)\mid(p'+q',\ s'+r'+p'\times q',\ c'+d') \\
= & (p+q+p'+q',\ s+r+s'+r'+p\times q+p'\times q', \\
 & \ c+c'+d+d'+(p+q)\widehat{\times}(p'+q'))\ .
\end{array}
$$

Hence, by definition of ⇒ and distributivity of × and $\widehat{\times}$ over + the exchange law holds.

Appendix B. The Lifting Theorem

Appendix B.1. Lifting Functions

Let B, C be pre-ordered sets. To ease notation we write $\sqsubseteq$ for both pre-orders on B and C. We lift every function $f : B \rightarrow C$ to a function $\hat{f} : \mathbb{P}(B) \rightarrow \mathbb{P}(C)$ on sets ($\mathbb{P}$ is the power set operator) by

$$\hat{f}(P) \ =_{df} \ dc(f(P))\ ,$$

where $f(P)$ is the image set of P under f. Note that $\hat{f}$ is strict w.r.t. $\emptyset$, i.e., $\hat{f}(\emptyset) = \emptyset$, since $f(\emptyset) = \emptyset$.

Lemma B.1 If f is covariant then $\hat{f}$ is covariant w.r.t. the lifted pre-orders $\sqsubseteq$ on $\mathbb{P}(B)$ and $\mathbb{P}(C)$.

Proof. Assume $P \sqsubseteq P'$ for downward closed $P, P' \subseteq B$. Consider a $q \in \hat{f}(P)$. By definition of $\hat{f}$ and of $\sqsubseteq$ on sets there must be $p \in P$ with $q \sqsubseteq f(p)$. Since $P \sqsubseteq P'$, there must be a $p' \in P'$ with $p \sqsubseteq p'$. By covariance of f we have $f(p) \sqsubseteq f(p')$. Therefore $q \sqsubseteq f(p') \in \hat{f}(P')$. □

The general case of n-ary functions is covered by this, too, since downward closed sets are closed under Cartesian products.

Appendix B.2. Terms and Their Values

Given a set of names for variables and set of *base functions*, terms are defined as usual. A *point valuation* v maps every variable x to an element $v(x) \in B_x$ of some pre-ordered set B_x, whereas a *set valuation* V maps it to a set $V(x) \in \mathbb{P}(B_x)$. For a term t the *value* $t[v]$ is computed from the v-values of the variables in t using the original functions, while the value $t[V]$ is computed from the V-values of the variables in t using the lifted functions. Hence under a set valuation, different occurrences of the same variable may receive different values.

A term is *linear* if no variable occurs twice in it.

We say that a point valuation v is *admitted* by a set valuation V and write $v \preceq V$ if for all variables x we have $v(x) \in V(x)$. This allows stating the following connections.

Lemma B.2 Assume that all base functions are covariant and let t be a term.

1. If $v \preceq V$ then $t[v] \in t[V]$.
2. If t is linear then $t[V] = dc(\{t[v] \,|\, v \prec V\})$.

Proof. Both parts are shown by induction on the structure of terms.

1. This is straightforward from the definitions and left to the reader.
2. The inclusion ($\subseteq$) follows from Part 1. For the reverse inclusion the base cases of constants and variables are again immediate from the definitions. In the inductive case $t = f(t_1 \ldots, t_n)$ we have $t[V] = \hat{f}(t_1[V], \ldots, t_n[V])$. So for every $a \in t[V]$ there must be $a_i \in t_i[V]$ such that $a \sqsubseteq f(a_1, \ldots, a_n)$. By the induction hypothesis we also have $a_i = t_i[v_i]$ for some valuations $v_i \preceq V$. Since t is linear, every variable occurs in at most one of the t_i. Therefore we can construct a new well defined valuation w from v by setting

$$w(x) =_{df} \begin{cases} v_i(x) & \text{if } x \text{ occurs in } t_i \\ v(x) & \text{if } x \text{ does not occur in } t \end{cases}$$

Then $w \prec V$ again and $a_i = t_i[v_i] = t_i[w]$. Therefore $f^A(a_1, \ldots, a_n) \in \{t[v] \mid v \prec V\}$ as well. □

Appendix B.3. Lifting Laws

The aim of this section is to show that linear inequational laws lift from the level of elements to that of downward closed sets of elements.

Consider terms l, r of the same type over some set of base functions. The law $l \sqsubseteq r$ *holds pointwise* if $l[v] \sqsubseteq r[v]$ for all point valuations of the variables in the terms. It *holds setwise* if $l[V] \sqsubseteq r[V]$ for all set valuations V.

A law $l \sqsubseteq r$ with terms l, r is called *linear* if l and r are linear and $Var(r) \subseteq Var(l)$, where $Var(t)$ is the set of variables occurring in term t. The condition $Var(r) \subseteq Var(l)$ serves to cope with the strictness of the lifted operations: for a set valuation V that assigns $\emptyset$ to any variable in $Var(r)$ we have $r[V] = \emptyset$. Hence we can have $l[V] \sqsubseteq r[V]$ only if $l[V] = \emptyset$ as well. This is guaranteed by $Var(r) \subseteq Var(l)$.

Example B.3 Consider the algebra of natural numbers. The law $x \cdot 0 \leq 0$ holds both pointwise and setwise, whereas the reverse law $0 \leq x \cdot 0$ holds only pointwise. Similarly, the law $x \leq x + y$ holds only pointwise, not setwise.

Theorem B.4 Assume a set of covariant base functions. If a linear law $l \sqsubseteq r$ holds pointwise then it also holds setwise.

Proof. Consider a set valuation V. By the above discussion it suffices to treat the case where $V(x) \neq \emptyset$ for all $x \in Var(r)$. Since, as above, $l[V] = \emptyset$ if V assigns $\emptyset$ to any variable in $Var(l)$ we trivially have $l[V] \sqsubseteq^A r[V]$ in that case. Also, if $Var(l) = \emptyset$ (and hence $Var(r) = \emptyset$ as well) we trivially have $l[V] \sqsubseteq^A r[V]$.

Hence we only need to study the case where $Var(l) \neq \emptyset$ and $V(x) \neq \emptyset$ for all $x \in Var(l)$. Then $\{v \mid v \prec V\} \neq \emptyset$.

By linearity of l and r and Lemma B.2.2 we have

$$l[V] \sqsubseteq r[V] \iff dc(\{l[v] \,|\, v \prec V\}) \subseteq dc(\{r[v] \,|\, v \prec V\}) .$$

According to the definition (1) the right hand side is equivalent to

$$\{l[v] \mid v \prec V\} \sqsubseteq \{r[v] \mid v \prec V\} ,$$

which, by the definition of $\sqsubseteq$ on sets, follows from the assumption that $l \sqsubseteq r$ holds pointwise. □

An equational law is *bilinear* if the laws $l \sqsubseteq r$ and $r \sqsubseteq l$ are linear. In particular, then $Var(l) = Var(r)$.

Corollary B.5 Assume again a set of covariant base functions. If a bilinear law $l = r$ holds pointwise then it also holds setwise.

For implications of inequations, the situation is simpler: covariance of the operators is no longer necessary, because lifted operators are by definition $\subseteq$-covariant on general sets and hence also $\sqsubseteq$-covariant on downward closed sets. An example is the covariance law:

$$P \sqsubseteq Q \implies f(P) \sqsubseteq f(Q).$$

Appendix B.4. How to Use the Results

First we note that in the case where the pre-order on the base set B is discrete, i.e., the identity relation, we retrieve the classical result by Gautam [28].

Second, if the base set B has a least element $\bot$ it is advisable to use not the full power set $\mathbb{P}(B)$ but only $\mathbb{P}(B)\backslash\{\emptyset\}$. The reason is that then $\emptyset \sqsubseteq \{\bot\}$ but not $\{\bot\} \sqsubseteq \emptyset$, which means that there are two kinds of modelling erroneous programs although the semantic difference between $\emptyset$ and $\{\bot\}$ is not really clear.

Eliminating $\emptyset$ from consideration in this case seems the appropriate choice, as the lengthy discussion at the beginning of the proof of Th. B.4 shows.

This decision covers the case of the standard totalisation of a partial function f that maps arguments at which f is undefined to $\bot$ and makes f strict w.r.t. $\bot$, i.e., sets $f(\bot) = \bot$.

This is of particular interest if the order $\sqsubseteq$ on B is the *flat order*

$$x \sqsubseteq y \iff_{df} x = \bot \lor x = y \, .$$

Then the non-empty downward closed subsets X of B are characterised by $\bot \in X$. For these we have, with the pointwise lifting,

$$X \sqsubseteq Y \iff X \subseteq Y \iff X - \{\bot\} \subseteq Y - \{\bot\} \, ,$$

i.e., they are order-isomorphic to the standard power set of $B - \{\bot\}$ under inclusion. Hence in the case of a flat base order, downward closure can be omitted from the power domain.

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, M. Gabbay, D, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume III. Oxford University Press, 1994.

[2] A. Armstrong, V. B. F. Gomes, and G. Struth. Algebraic principles for rely-guarantee style concurrency verification tools. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *FM 2014*, volume 8442 of *LNCS*, pages 78–93. Springer, 2014.

[3] A. Armstrong, V. B. F. Gomes, and G. Struth. Algebras for program correctness in Isabelle/HOL. In P. Höfner, P. Jipsen, W. Kahl, and M. E. Müller, editors, *RAMICS 2014*, volume 8428 of *LNCS*, pages 49–64. Springer, 2014.

[4] A. Armstrong and G. Struth. Automated reasoning in higher-order regular algebra. In W. Kahl and T. G. Griffin, editors, *RAMICS 2012*, volume 7560 of *LNCS*, pages 66–81. Springer, 2012.

[5] A. Armstrong, G. Struth, and T. Weber. Programming and automating mathematics in the Tarski-Kleene hierarchy. *Journal of Logical and Algebraic Methods in Programming*, 83(2):87–102, 2014.

[6] G. Armstrong, V. B. F. Gomes, and G. Struth. Lightweight program construction and verification in Isabelle/HOL. In *SEFM 2014*, LNCS. Springer, 2014. (In press.).

[7] R.-J. Back. On the correctness of refinement steps in program development. Technical Report A-1978-4, Deptartment of Computer Science, University of Helsinki, 1978.

[8] R.-J. Back and J. von Wright. *Refinement Calculus - A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[9] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[10] A. Blikle. A comparative review of some program verification methods. In J. Gruska, editor, *MFCS 1977*, volume 53 of *LNCS*, pages 17–33. Springer, 1977.

[11] S. L Bloom and Z. Ésik. Free shuffle algebras in language varieties. *Theoretical Computer Science*, 163(1&2):55–98, 1996.

[12] C. Brink. Power structures. *Algebra Universalis*, 30(2):177–216, 1993.

[13] C. Brink and I. Rewitzky. *A Paradigm for Program Semantics: Power Structures and Duality*. CSLI Publications, 2001.

[14] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *JACM*, 31:560–599, 1984.

[15] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS 2007*, pages 366–378. IEEE Computer Society, 2007.

[16] K. Mani Chandy and Jayadev Misra. Systolic algorithms as programs. *Distributed Computing*, 1(3):177–183, 1986.

[17] J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. *J. Log. Comput.*, 17(4):807–841, 2007.

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

[19] H.-H. Dang, P. Höfner, and B. Möller. Algebraic separation logic. *Journal of Logic and Algebraic Programming*, 80(6):221–247, 2011.

[20] H.-H. Dang and B. Möller. Transitive separation logic. In W. Kahl and T. Griffin, editors, *RAMICS 2012*, volume 7560 of *LNCS*, pages 1–16. Springer, 2012.

[21] H.-H. Dang and B. Möller. Concurrency and local reasoning under reverse exchange. *Science of Computer Programming*, 85:204–223, 2014.

[22] H.-H. Dang and B. Möller. Modal algebra and petri nets. *Acta Informatica*, 52(2-3):109–132, 2015.

[23] J. de Bakker and L. Meertens. On the completeness of the inductive assertion method. *Journal of Computer and System Sciences*, 11(3):323–357, 1975.

[24] J. Desharnais, B. Möller, and G. Struth. Kleene algebra with domain. *ACM Transactions on Computational Logic*, 7(4):798–833, 2006.

[25] J. Desharnais and G. Struth. Internal axioms for domain semirings. *Science of Computer Programming*, 76(3):181–203, 2011.

[26] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[27] B. Dongol, V. B. F. Gomes, and G. Struth. A program construction and verification tool for separation logic. In R. Hinze, editor, *MPC 2015*, LNCS, 2015. To appear.

[28] N. Gautam. The validity of equations of complex algebras. *Arch. Math. Logik Grundl. Mat.*, 443:117–124, 1957.

[29] J. L. Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 61(2–3):199–224, 1988.

[30] R. Goldblatt. Varieties of complex algebras. *Annals of Pure and Applied Logic*, 44:173–242, 1989.

[31] A. Gotsman, B. Cook, M. J. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In Z. Shao and B. C¿ Pierce, editors, *POPL 2009*, pages 16–28. ACM, 2009.

[32] J. Grabowski. On partial languages. *Fundamenta Informaticae*, 4(2):427–498, 1981.

[33] G. Grätzer and S. Whitney. Infinitary varieties of structures closed under the formation of complex structures. *Colloquium Mathematicae*, 48:1–5, 1984.

[34] W. Guttmann, G. Struth, and T. Weber. Automating algebraic methods in Isabelle. In S. Quin and Z. Qiu, editors, *ICFEM 2011*, volume 6991 of *LNCS*, pages 617–632. Springer, 2011.

[35] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[36] C. A. R. Hoare, I. J. Hayes, J. He, C. Morgan, A. W. Roscoe, J. W. Sanders, I. Holm Sørensen, J. M. Spivey, and B. Sufrin. Laws of programming. *Commun. ACM*, 30(8):672–686, 1987.

[37] T. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene algebra and its foundations. *Journal of Logical Algebraic Programming*, 80(6):266–296, 2011.

[38] T. Hoare and S. van Staden. The laws of programming unify process calculi. *Science of Computer Programming*, 85:102–114, 2014.

[39] IEEE. *IEEE Standard Hardware Description Language based on the Verilog Hardware Description Language*, volume IEEE Standard 1364-1995. IEEE, 1995.

[40] IEEE. *IEEE Standard Hardware Description Language based on the Verilog Hardware Description Language*, volume IEEE Standard 1364-2001. IEEE, 2001.

[41] P. Jipsen. Concurrent Kleene algebra with tests. In P. Höfner, P. Jipsen, W. Kahl, and M. E. Müller, editors, *RAMICS 2014*, volume 8428 of *LNCS*, pages 37–48. Springer, 2014.

[42] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, chapter 1956, pages 3–41. Princeton University Press, 1956.

[43] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[44] D. Kozen. On action algebras. In J. van Eijck and A. Visser, editors, *Logic and Information Flow*, pages 78–88. MIT Press, 1994.

[45] H. Kung and C. E. Leiserson. Algorithms for vlsi processor arrays. In Carver Mead and Lynn Conway, editors, *Introduction to VLSI Systems*, pages 271–292. Addison-Wesley, 1979.

[46] M. R. Laurence and G. Struth. Completeness theorems for bi-Kleene algebras and series-parallel rational pomset languages. In P. Höfner, P. Jipsen, W. Kahl, and M. E¿ Müller, editors, *RAMICS 2014*, volume 8428 of *LNCS*, pages 65–82. Springer, 2014.

[47] Inmos Limited. *Occam 2 Reference Manual*. Prentice Hall, 1988.

[48] R. D. Maddux. *Relation Algebras*. Elsevier, 2006.

[49] M. Main. A powerdomain primer — a tutorial for the bulletin of the EATCS 33. Technical Report CU-CS-375-87 (1987). Paper 360, Univ. Colorado at Boulder, Dept of Computer Science, 1987.

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

[51] R. Milner. *A Calculus of Communication Systems*. Number 92 in LNCS. Springer, 1980.

[52] B. Möller and C. A. R. Hoare. Exploring an interface model for CKA. In R. Hinze and J. Voigtländer, editors, *MPC 2015*, LNCS. Springer, 2015. To appear.

[53] Bernhard Möller, Peter Höfner, and Georg Struth. Quantales and temporal logics. In M. Johnson and V. Vene, editors, *AMAST 2006*, volume 4019 of *LNCS*, pages 263–277. Springer, 2006.

[54] C. C. Morgan. The specification statement. *ACM TOPLAS*, 10(3):403–419, 1988.

[55] M. E. Müller. Modalities, relations, and learning. In R. Berghammer, B. Möller, and A. Jaoua, editors, *RAMICS 2009*, volume 5827 of *LNCS*, pages 260–275. Springer, 2009.

[56] M. E. Müller. *Relational Knowledge Discovery*. Cambridge University Press, 2012.

[57] M. E. Müller, F. Krebs, and F. Hielscher. Relational cognitive structures for intelligent agent and robot control. In *Proc. IEEE International Conference on Systems, Man and Cybernetics, Singapore, 12-15 October 2008*, pages 3156–3163. IEEE, 2008.

[58] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[59] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[60] Open SystemC Initiative (OSCI). *SystemC 2.0.1 Language Reference Manual*, 2003.

[61] D. Pavlovic, P. Pepper, and D. Smith. Formal derivation of concurrent garbage collectors. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *MPC 2010*, volume 6120 of *LNCS*, pages 353–376. Springer, 2010.

[62] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-16, Computer Science Department, Aarhus

University, 1981.

[63] V. R. Pratt. Some constructions for order-theoretic models of concurrency. In R. Parikh, editor, *Logic of Programs*, volume 193 of *LNCS*, pages 269–283. Springer, 1985.

[64] V. R. Pratt. Action logic and pure induction. In J. van Eijck, editor, *Logics in AI*, volume 478 of *LNCS*, pages 97–120. Springer, 1991.

[65] A. W. Roscoe. *A mathematical theory of communicating processes*. PhD thesis, Oxford University, 1982.

[66] A. Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, 1966.

[67] G. Schmidt. Programme als partielle Graphen. TU Munich, FB Mathematik. Habilitation Thesis, 1977.

[68] A. Tarlecki. A language of specified programs. *Science of Computer Programming*, 5(1):59–81, 1985.

[69] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[70] I. Wehrman, C.A.R. Hoare, and P. OHearn. Graphical models of separation logic. *IPL*, 109(17):1002–1004, 2009.

[71] G. Winskel. On powerdomains and modality. *Theoretical Computer Science*, 36:127–137, 1985.

[72] H. Zhu, J. He, and J. P. Bowen. From algebraic semantics to denotational semantics for Verilog. *Innovations in Systems and Software Engineering: A NASA Journal*, 4(4), 2008.

[73] H. Zhu, J. He, S. Qin, and P. J. Brooke. Denotational semantics and its algebraic derivation for an event-driven system-level language. *Formal Aspects of Computing*, 27(1):133–166, 2015.

[74] H. Zhu, P. Liu, J. He, and S. Qin. Mechanical approach to linking operational semantics and algebraic semantics for Verilog using Maude. In *Proc. UTP 2012*, volume 7681 of *LNCS*, pages 164–185. Springer-Verlag, 2012.

[75] H. Zhu, S. Qin, J. He, and J. P. Bowen. PTSC: probability, time and shared-variable concurrency. *Innovations in Systems and Software Engineering: A NASA Journal*, 5(4):271–284, 2009.

[76] H. Zhu, F. Yang, J. He, J. P. Bowen, J. W. Sanders, and S. Qin. Linking operational semantics and algebraic semantics for a probabilistic timed shared-variable language. *Journal of Logic and Algebric Programming*, 81(1), 2012.