# Building Structured Theories

Bernhard Möller

Institut für Informatik, Universität Augsburg, D-86135 Augsburg, Germany
bernhard.moeller@informatik.uni-augsburg.de

**Abstract.** We provide a set of syntactic tools for structuring large collections of logical theories. Their use is demonstrated by a formalisation of algebras that are used in describing the semantics of concepts in programming languages, but also of more general systems.

## 1 Introduction

Within the series of RelMiCS, AKA and now RAMiCS conferences we have seen many algebraic theories, starting with relation and Kleene algebras, which have diversified considerably to cover more and more application areas. Still, many of them share a significant common core, and hence it seems adequate to think about their connections in a systematic way. At the same time, some of the theories are quite complex. This is similar to the situation in programming, where one tries to cope with that using suitable structuring mechanisms, such as inheritance and encapsulation.

In the present paper we attempt a similar structured presentation of some essential RAMiCS theories. While there is already some work in that direction in connection with treating these theories with automatic theorem provers [6, 15, 29, 30], we try to modularise further in a number of new and perhaps unusual ways to pinpoint more clearly which parts of the theories depend on which others.

Of course, there is already a lot of work on structuring larger formal theories. There is the long series of languages designed in the field of *algebraic specification*, like CLEAR [3], CIP-L [2], ASL [67], ACT ONE [14] and CASL [4]. They all comprise some sort of structuring mechanism, and many show notational similarity to what we will use in the present paper. However, by their nature they are mostly restricted to first-order equational logic, whereas we will be more liberal. There is also work on structuring specifications in Edinburgh LCF [40, 56]. General structured specification frameworks based on category theory appear in [12, 16, 20, 58, 60, 61]. And there is the interesting dependently typed functional language Agda [1] with proof assistant, which also allows expressing structured theories.

What we present here deviates from these approaches in that we introduce a number of additional construction mechanisms. Moreover, we forego the definition of a semantics in terms of operations on model classes or of pushouts/colimits. Rather, we view our structuring tools as syntactic devices that abbreviate certain compounds of formulas and can be re-used and instantiated to exhibit common and recurring parts of specifications. For their meaning we rely on the standard semantics of first-order and higher-order logic.

In motivating the particular ingredients of the theories we present we frequently resort to their use in specifying the semantics of transition systems and the like. However, as it has been demonstrated in many excellent papers throughout this series of conferences, the theories have much wider applicability, and we hope that our methods of structuring will help in extending the algebraic treatment to many further areas.

## 2 Theories and Definitions

A *theory* has a name and may have an imports clause that specifies on which other theories it depends, a list of *sorts* (i.e., names for carrier sets), a list of operators and a list of predicates, each with their typing, a list of axioms (which should be independent) and a list of properties, starting with the keyword derives, that follow from the axioms. We will only write down the non-empty ones of these; list items are separated by the symbol ¦ or line breaks, sometimes also by a horizontal line. For space limitations we usually list only a few of the more interesting/important derived properties. The operators and predicates are called the *constituents* of the theory. Occasionally we will mark certain constituents as hidden, since they only have auxiliary character for formulating certain axioms in a more convenient and generic way. All non-hidden constituents are visible to the outside and can be imported by other theories. A theory may also contain a list of typed variables that are used in the axioms or derived properties. We omit the explicit definition of variables whose type can be inferred from the typing of the operators and predicates that are applied to them. We use the standard convention that all free variables in a logical formula are implicitly universally quantified.

*Definitions* are similar to theories except that they do not contain axioms. Rather they give, following the keywords defined by, definitional equalities or equivalences for each of their constituents. The only exception are new constants that may be added without giving particular properties for them.

The distinction between theories and definitions is purely for documentation purposes. For brevity we will refer to them uniformly as *(building) blocks*. Blocks may be freely *imported* and/or *instantiated*, possibly under renaming. For the latter we use simple positional notation, listing the new names between parentheses after the block name. The meaning of an import is simple replacement of the block name by its body (with renaming if specified). If no renaming list is given, the block is imported with its original names. Hence upon import of several blocks into another one, identical names mean identical constituents.

An instantiated block may also be used in the axioms, defined by or derives parts of other blocks; in this case its constituent information is ignored and only the logical formulas in its body are copied in (under renaming if specified). In this case the block serves as a function from constituent names to sets of formulas. By this twofold use of blocks we achieve a certain notational economy, as will be seen in the examples.

Types may be simple identifiers or Cartesian product, function or power set types. Mostly, however, we will use the higher types only in auxiliary blocks to improve the structuring; they will then disappear again after instantiation of these blocks. Only at the very end of the paper, when we talk about quantales, some higher types persist. A unary predicate is identified with the subset of elements that satisfy it. Use of such a predicate in the position of a type then achieves subsorting. In particular, if variables are declared to be of such a subsort type, quantifiers involving them range over the subsort only.

As first examples to show our notation at work we specify some aspects of comparison, in particular, of preorders and partial orders. First we just introduce the type of the comparison predicate.

| theory | COMPARE |
|---|---|
| sorts | $S$ |
| predicates | $\leq \subseteq S \times S$ |

Next, even without any assumptions on the predicate $\leq$, we define the concepts of isotony and antitony. This already involves predicates of higher type that take functions as arguments.

| definition | ISO_ANTI |
|---|---|
| imports | COMPARE |
| predicates | $isotone, antitone \subseteq S \to S$ |
| defined by | $isotone(f) \Leftrightarrow_{df} \forall x, y \,.\, x \leq y \Rightarrow f(x) \leq f(y)$<br>$antitone(f) \Leftrightarrow_{df} \forall x, y \,.\, x \leq y \Rightarrow f(y) \leq f(x)$ |

We now introduce a general mechanism for propagating properties like isotony and antitony to binary operators. This again involves higher-order concepts.

| theory | LEFT_ARG |
|---|---|
| sorts | $S$ |
| operators | $g : S \times S \to S$ |
| predicates | $P \subseteq S \to S$ |
| hidden | $right\_const : S \to (S \to S)$ |
| axioms | $right\_const(y)(x) = g(x, y)$<br>$\forall y \,.\, P(right\_const(y))$ |

Now for instance LEFT_ARG$(T, \circ, isotone)$ expresses that an operator $\circ : T \times T \to T$ on some set $T$ is isotone in its left argument. A symmetrical theory RIGHT_ARG propagates a predicate to the right argument of a binary operator. Below we will also use this mechanism to express left and right distributivity of a binary operator in terms of distributivity of a unary one.

Next, we specify preorders and partial orders.

| theory | PREORDER |
|---|---|
| imports | COMPARE |
| axioms | $x \leq x$<br>$x \leq y \wedge y \leq z \Rightarrow x \leq z$ |

| theory | POSET |
|---|---|
| imports | PREORDER |
| axioms | $x \leq y \wedge y \leq x \Rightarrow x = y$ |

The fact that a set $T$ with a binary relation $\preceq$ on it forms a partial order can then be expressed as POSET$(T, \preceq)$.

With a similar theory BOOLEAN_ALG$(S, \sqcup, \sqcap, \neg, \bot, \top)$ one specifies Boolean algebras; we omit the detailed axioms and properties. We also introduce some standard notation for Boolean algebras without listing derived properties:

| definition | BOOLEAN_OPS |
|---|---|
| imports | BOOLEAN_ALG |
| operators | $-, \rightarrow : S \times S \rightarrow S$ |
| defined by | $x - y =_{df} x \sqcap \neg y \mid x \rightarrow y =_{df} \neg x \sqcup y$ |

## 3 Sequential Composition

We start our treatment of semantic theories with sequential composition which occurs in many quite different contexts. Sequentiality can concern time and space, like in sequences of events or elements of a list or an array.

We will denote sequential composition abstractly by $\cdot$. Concrete instances are concatenation of formal languages (with finite or infinite words), relational composition or gluing of sets of trajectories or program sequencing.

For now we do not require any laws about sequential composition. This is captured by our first block:

| theory | GROUPOID |
|---|---|
| sorts | $S$ |
| operators | $\cdot : S \times S \rightarrow S$ |

Already with this extremely general structure we can describe interesting and important computational phenomena, as will be shown in the next section.

But first we specify commutativity and idempotence:

| theory | COMMUTATIVE |
|---|---|
| imports | GROUPOID |
| axioms | $x \cdot y = y \cdot x$ |

| theory | IDEMPOTENT |
|---|---|
| imports | GROUPOID |
| axioms | $x \cdot x = x$ |

## 4 Annihilation

An element $x$ is a *left annihilator* w.r.t. to sequential composition if composition with any element on the right does not change it.

| definition | LEFT_ANNI |
|---|---|
| imports | GROUPOID |
| predicates | $left\_anni \subseteq S$ |
| defined by | $left\_anni(x) \Leftrightarrow_{df} \forall y \,.\, x \cdot y = x$ |

Left annihilation means absolute domination. It can be used to model catastrophic failure: after an annihilating element "nothing else can happen". Note, however, that sometimes left annihilation is a highly desirable property: when studying infinite computations, like the ones initiated by (hopefully) always continuing operating systems, we usually do not want to take "behaviour after infinity" into account, hence the desired sets of behaviours of such systems should be

left annihilators. In general, there may be various left annihilators in a groupoid. For instance, in UTP [21] both the totally undefined and the totally unreliable process are left annihilators. Symmetrically one specifies a right annihilator using a definition RIGHT_ANNI with a predicate *right_anni*.

## 5 Characterising Failure

Although it seems almost paradoxical, something useful can be achieved with annihilators. The ideas here are inspired by [11, 48] and were generalised in [44].

We assume that there is a distinguished left annihilator. It is intended to represent systems about which nothing definitive can be said and which hence can be viewed as "failing" in some sense. Since we are denoting sequential composition by $\cdot$, a fitting notation for such an element is 0.

| theory | FAILURE |
|---|---|
| imports | LEFT_ANNI |
| operators | $0 : S$ |
| axioms | $left\_anni(0)$ |

We will now, in our diction, adopt the view that an element $x$ is "failing" iff it represents a system that fails to terminate. Hence, as discussed above, $x$ should be a left annihilator. As all the computations of such a system are infinite, we will call $x$ *purely infinite*. Dually, an element $x$ will be called *purely finite* if it "notices" subsequent nontermination, i.e, if $x \cdot 0 = 0$. Notice that 0 is both purely infinite and purely finite, but is the only such element. A semantic algebra is *strict* if all its elements are purely finite, i.e., iff 0 is also a right annihilator.

| definition | FIN_INF |
|---|---|
| imports | FAILURE |
| predicates | $purely\_inf \subseteq S$<br>$purely\_fin \subseteq S$ |
| defined by | $purely\_inf(x) \Leftrightarrow_{df} left\_anni(x)$<br>$purely\_fin(x) \Leftrightarrow_{df} x \cdot 0 = 0$ |

| theory | STRICT_COMP |
|---|---|
| imports | FAILURE ¦ RIGHT_ANNI |
| axioms | $right\_anni(0)$ |

## 6 Further Aspects of Sequential Composition

Typically one requires at least associativity of sequential composition. This is captured by our next blocks. We specify left associativity; a symmetric theory R_ASSOC provides the predicate *right_assoc* of right associativity.

| theory | L_ASSOC |
|---|---|
| imports | GROUPOID |
| predicates | $left\_assoc \subseteq S$ |
| axioms | $left\_assoc(x) \Leftrightarrow_{df} \forall y, z \,.\, x \cdot (y \cdot z) = (x \cdot y) \cdot z$ |

Left-associativity and pure infiniteness show interesting connections:

| theory | L_ASSOC_INF |
|---|---|
| imports | FAILURE ¦ L_ASSOC |
| derives | $purely\_inf(x) \Rightarrow left\_assoc(x)$<br>$left\_assoc(x) \Rightarrow (purely\_inf(x) \Leftrightarrow x = x \cdot 0)$ |

Using the associativity predicates we can talk about semigroups.

| theory | SEMIGROUP |
|---|---|
| imports | L_ASSOC ¦ R_ASSOC |
| axioms | $left\_assoc(x)$ |
| derives | $right\_assoc(y)$ |

Frequently, one also assumes a unit 1 of composition. Concrete instances are the language $\varepsilon$ consisting just of the empty word, the identity relation or the empty program skip. This leads to the next block, which can further be combined with pure finiteness.

| theory | MONOID |
|---|---|
| imports | SEMIGROUP |
| operators | $1 : S$ |
| axioms | $1 \cdot x = x = x \cdot 1$ |

| theory | ONE_FIN |
|---|---|
| imports | MONOID ¦ FIN_INF |
| derives | $purely\_fin(1)$ |

## 7 Concurrency

A well studied algebraic framework for concurrency are the various process calculi (ACP, CCS, CSP, ...), with varying properties of choice and sequential composition. We will here treat some aspects of the recent approach of *concurrent Kleene algebras* [22]. Next to sequential composition $\cdot$ these offer a parallel composition $\parallel$. A basic idea is that the elements of such an algebra abstractly represent sets of traces of some kind. These traces consist of events that are occurrences of certain primitive actions such as communications or assignments. One assumes that there are certain causal or temporal dependences between events. Sequential composition has to respect these dependences, i.e., in a composition $x \cdot y$ no event in a trace in $x$ may depend on a "future" event in a trace in $y$. Parallel composition is much more liberal in that it does not impose such a restriction (at the expense of allowing "hazardous" programs with race conditions on the resources involved).

To capture this algebraically, one introduces a comparison relation $\leq$ where $x \leq y$ expresses that $y$ is more liberal than $x$. Let us now look at the terms $(x \parallel y) \cdot (z \parallel u)$ and $(x \cdot z) \parallel (y \cdot u)$. The first of these is a sequential composition of two parallel compositions. Therefore neither $x$ nor $y$ may depend on $z$ or $u$. The second one is a parallel composition of two sequential compositions with the same basic constituents. This is more liberal than the first one, since only dependence of $x$ on $z$ and of $y$ on $u$ must be excluded. This fundamental property is the basis of the algebraic axiomatisation.

| theory | CONCURRENT_BIGROUPOID |
|---|---|
| imports | GROUPOID$(S, \cdot)$ ¦ GROUPOID$(S, \|\|)$ ¦ COMPARE |
| axioms | $(x \parallel y) \cdot (z \parallel u) \ \leq \ (x \cdot z) \parallel (y \cdot u)$ |

Since one wants to construct longer derivation chains, one frequently requires $\leq$ to be a partial order to admit transitivity steps. Moreover, in the basic model of [22] these operators are associative and there is an idle process 1 which is a common unit of $\cdot$ and $\parallel$.

| theory | CONCURRENT_BIMONOID |
|---|---|
| imports | CONCURRENT_BIGROUPOID |
| operators | $1 : S$ |
| axioms | MONOID$(S, \cdot, 1)$ ¦ MONOID$(S, \|\|, 1)$ |

An extension of this basic theory admits, a.o., a simple algebraic treatment of the rely/guarantee calculus of [33]. The details would lead too far here. Further applications are under way.

## 8 The Frame Rule and Separation Logic

Over every ordered groupoid one can define a very general form of Hoare triple, for which the classical rules of sequencing and weakening hold:

| definition | GROUPOID_HOARE_TRIPLE |
|---|---|
| imports | GROUPOID ¦ POSET |
| predicates | $\_\{\_\}\_ \subseteq S \times S \times S$ |
| defined by | $x \{y\} z \ \Leftrightarrow_{df} \ x \cdot y \leq z$ |
| derives | $x \{u \cdot v\} z \ \Leftrightarrow \ \exists y \,.\, x \{u\} y \ \wedge \ y \{v\} z$<br>$x \leq u \ \wedge \ u \{y\} v \ \wedge \ v \leq z \ \Rightarrow \ x \{y\} z$ |

If the groupoid is even a monoid one can also infer the classical rule for skip, viz. $x \{1\} z \ \Leftrightarrow \ x \leq z$. In the presence of parallel composition one obtains the concurrency rule, and the so-called frame allows modular reasoning where a disjoint context may be added in parallel to a program without invalidating the reasoning using triples:

| theory | CONCURRENT_HOARE_TRIPLE |
|---|---|
| imports | CONCURRENT_BIGROUPOID ¦ GROUPOID_HOARE_TRIPLE |
| derives | $x \{y\} z \ \wedge \ u \{v\} w \ \Rightarrow \ (x \parallel u) \{y \parallel v\} (z \parallel w)$<br>$x \{y\} z \ \Rightarrow \ (u \parallel x) \{y\} (u \parallel z)$ |

A variant of the frame rule is of particular interest in the so-called separation logic [54] which allows modular reasoning about shared and mutable data structures with pointers. Again, the details would lead too far.

## 9 Choice

The second fundamental concept that occurs in many circumstances is that of choosing — in a more or less biased way — between a number of possibilities. If

the number of these possibilities is finite one speaks of *bounded choice*, otherwise of *unbounded choice*. Bounded choice is mostly represented by a binary operator for choosing between two alternatives, which under suitable assumptions allows an inductive definition of choosing between any positive number of possibilities. We will see below how to deal with zero possibilities. Frequent notations for that operator are $[]$ , $\sqcup$, $\sqcap$ and $+$, of which we will use the latter. Its typical axioms are associativity, commutativity and idempotence, which are the axioms for a meet or join semilattice.

| theory | SEMILATTICE |
|---|---|
| sorts | $S$ |
| operators | $+ : S \times S \to S$ |
| axioms | SEMIGROUP$(S, +)$ ¦ COMMUTATIVE$(S, +)$ ¦ IDEMPOTENT$(S, +)$ |

It is well known that every semilattice induces an order. In this paper we interpret $+$ as a join operator and write $\leq$ for the induced order, which is a derived concept and hence not specified by a theory but by a definition. It develops its full power in combination with a semilattice.

| definition | SUBSUMPTION |
|---|---|
| imports | GROUPOID$(S, +)$ |
| predicates | $\leq \subseteq S \times S$ |
| defined by | $x \leq y \Leftrightarrow_{df} x + y = y$ |

| theory | SEMILATTICE_WITH_SUBSUMPTION | |
|---|---|---|
| imports | SEMILATTICE ¦ SUBSUMPTION ¦ ISO_ANTI | |
| derives | POSET$(S, \leq)$ | |
| | $x \leq x + y$ | $x + y \leq z \Leftrightarrow x \leq z \wedge y \leq z$ |
| | LEFT_ARG$(S, +, isotone)$ | RIGHT_ARG$(S, +, isotone)$ |

In many cases, a semilattice of sets under union as join is used; the subsumption order there coincides with set inclusion.

Over semilattices we can specify the property of distributivity:

| definition | DIST |
|---|---|
| imports | SEMILATTICE_WITH_SUBSUMPTION |
| predicates | $distributive, super\_distributive \subseteq S \to S$ |
| defined by | $distributive(f) \Leftrightarrow_{df} \forall x, y \,.\, f(x + y) = f(x) + f(y)$ |
| | $super\_distributive(f) \Leftrightarrow_{df} \forall x, y \,.\, f(x) + f(y) \leq f(x + y)$ |
| derives | $distributive(f) \Rightarrow isotone(f)$ |
| | $super\_distributive(f) \Leftrightarrow isotone(f)$ |

Often a unit for choice is assumed. It represents the choice between zero possibilities. Its interpretation ranges from catastrophic error over failure to chaos. Since we denote choice by $+$, a fitting notation for its unit is 0. The definition of the subsumption order implies that 0 is its least element. Moreover, the fact that 0 is the unit of choice means that choice is angelic; a 0 branch will always be eliminated in favour of the other branch: $0 + x = x = x + 0$.

| theory | SEMILATTICE_WITH_MIN |
|---|---|
| imports | SEMILATTICE |
| operators | $0 : S$ |
| axioms | MONOID$(S, +, 0)$ |
| derives | $x + y = 0 \Rightarrow x = 0 = y$ |

| definition | SUBSUMPTION_WITH_MIN |
|---|---|
| imports | SEMILATTICE_WITH_MIN<br>SUBSUMPTION |
| derives | $0 \leq x$ |

## 10 Choice and Composition: Idempotent Semirings

An idempotent left semiring combines choice and composition such that composition is distributive in its left argument and isotone in its right one. As usual, we have composition bind tighter than choice.

| theory | IL_SEMIRING |
|---|---|
| imports | SEMILATTICE_WITH_MIN<br>MONOID ¦ FAILURE ¦ SUBSUMPTION ¦ ISO_ANTI ¦ DIST |
| predicates | $right\_dist \subseteq S$ |
| axioms | LEFT_ARG$(S, \cdot, distributive)$ ¦ RIGHT_ARG$(S, \cdot, isotone)$ |

In a left idempotent semiring we can study pure infiniteness a bit further.

| definition | IL_SEMIRING_INF |
|---|---|
| imports | IL_SEMIRING ¦ FIN_INF |
| operators | inf $: S \to S$ |
| defined by | inf $x =_{df} x \cdot 0$ |
| derives | $purely\_inf(x) \Rightarrow right\_dist(x)$<br>inf $x \leq x$ ¦ $purely\_inf(y) \wedge y \leq x \Rightarrow y \leq$ inf $x$ |

The last property means that inf $x$ is the greatest purely infinite element below $x$. Therefore we call inf $x$ the *purely infinite part* of $x$.

A number of applications use weak idempotent semirings, in which composition is right-distributive while 0 need not be a right annihilator:

| theory | WEAK_I_SEMIRING |
|---|---|
| imports | IL_SEMIRING |
| axioms | RIGHT_ARG$(S, \cdot, distributive)$ |

If additionally 0 is also a right annihilator and composition also distributes over choice in its right argument one speaks of an *idempotent semiring*.

| theory | I_SEMIRING |
|---|---|
| imports | WEAK_I_SEMIRING ¦ STRICT_COMP |

## 11 Converse and Relation Algebra

In many cases one is interested in reverting the "flow of control" as underlying sequential composition. To this end one uses the converse $x^\smile$ of an element $x$:

| theory | CONVERSE |
|---|---|
| imports | GROUPOID |
| operators | $\_^\smile : S \to S$ |
| axioms | $(x \cdot y)^\smile = y^\smile \cdot x^\smile$ |

Abstract relation algebra results from combining converse with a Boolean idempotent semiring:

| theory | RELATION_ALGEBRA |
|---|---|
| imports | BOOLEAN_ALG ¦ I_SEMIRING$(S, \sqcup, \bot, \cdot, 1)$ ¦ CONVERSE<br>SUBSUMPTION ¦ ISO_ANTI |
| axioms | $(x \sqcup y)^\smile = x^\smile \sqcup y^\smile$ ¦ $x \cdot \overline{x^\smile \cdot \overline{y}} \leq y$ |
| derives | $isotone(\_^\smile)$<br>$x \cdot y \sqcap z = \bot \Leftrightarrow x^\smile \cdot z \sqcap y = \bot \Leftrightarrow z \cdot y^\smile \sqcap x = \bot$ |

There is no need to tell the RAMiCS audience that there are many more interesting and useful consequences of the axioms.

## 12 Iteration

Following Kleene's seminal work [37], arbitrary finite iteration of an element $x$ is denoted by $x^*$. The axiomatisation follows [38].

| theory | LEFT_KLEENE_ALG |
|---|---|
| imports | IL_SEMIRING ¦ SUBSUMPTION ¦ ISO_ANTI |
| operators | $\_^* : S \to S$ |
| axioms | $1 + x \cdot x^* \leq x^*$ ¦ $y + x \cdot z \leq z \Rightarrow x^* \cdot y \leq z$ |
| derives | $isotone(\_^*)$<br>$x^* \cdot x^* = x^* = (x^*)^*$ ¦ $(x \cdot y)^* \cdot x = x \cdot (y \cdot x)^*$ ¦ $(x + y)^* = x^* \cdot (y \cdot x^*)^*$<br>$x \leq 1 \Rightarrow x^* = 1$ |

The symmetrical axioms that describe "iteration at the right" may need adjustments due to the application circumstances (e.g. in probabilistic algebras) [42, 64, 43]. Infinite iteration is added using seminal ideas from [53]; the axiomatisation follows [5].

| theory | LEFT_OMEGA_ALG |
|---|---|
| imports | LEFT_KLEENE_ALG ¦ SUBSUMPTION ¦ ISO_ANTI |
| operators | $\_^\omega : S \to S$ |
| axioms | $x^\omega = x \cdot x^\omega$ ¦ $z \leq y + x \cdot z \Rightarrow z \leq x^\omega + y \cdot x^*$ |
| derives | $isotone(\_^\omega)$<br>$0^\omega = 0$ ¦ $x^* \cdot x^\omega = x^\omega$ ¦ $x^\omega \cdot y \leq x^\omega$ ¦ $(x^\omega)^\omega \leq x^\omega$<br>$(x \cdot y)^\omega = x \cdot (y \cdot x)^\omega$ ¦ $(x + y)^\omega = (x^* \cdot b)^\omega + (x^* \cdot b)^* \cdot x^\omega$<br>$x \leq 1^\omega$ |

The last derived property motivates the following definition.

| definition | OMEGA_TOP |
|---|---|
| imports | LEFT_OMEGA_ALG |
| operators | $\top : S$ |
| defined by | $\top =_{df} 1^\omega$ |
| derives | $x \leq x \cdot x \Rightarrow x^\omega = x \cdot \top \mid x^\omega = x^\omega \cdot \top$ |

The latter property makes $x^\omega$, e.g., not adequate for the precise description of Zeno effects in hybrid systems. Hence again an adjustment may be needed.

## 13 Tests: Modelling Sets of States

Elements of semirings frequently represent sets of transitions. To represent sets of states one may use special transitions that abstract `assert` statements as known from programming. A statement `assert B` skips (i.e., leaves the state unchanged) if `B` holds and aborts otherwise. Considered as a relation, it is a subset of the identity relation on program states. Hence sets of program states or predicates characterising such sets are in one-to-one correspondence with subidentity relations. A central property is that for them intersection and composition coincide. All this lays the basis for an algebraic representation of general sets of states. Such an approach was presented, e.g., in [41] by distinguishing particular semiring elements which, following [39], we call *tests*. Since we want the tests, the algebraic counterparts of predicates, to form a Boolean algebra, we first specify complementation.

| definition | IL_SEMIRING_WITH_COMPL |
|---|---|
| imports | IL_SEMIRING |
| predicates | $are\_complements \subseteq S \times S$<br>$test \subseteq S$ |
| defined by | $are\_complements(p,q) \Leftrightarrow_{df} p + q = 1 \wedge p \cdot q = 0 \wedge q \cdot p = 0$<br>$test(p) \Leftrightarrow_{df} \exists\, q \,.\, are\_complements(p,q)$ |
| derives | $are\_complements(p,q) \Leftrightarrow are\_complements(q,p)$<br>$are\_complements(p,q) \Rightarrow p \leq 1 \wedge q \leq 1$<br>$are\_complements(p,q) \wedge are\_complements(p,r) \Rightarrow q = r$<br>$are\_complements(0,1)$ |

Now we can specify the notion of an idempotent left semiring with tests.

| theory | IL_SEMIRING_WITH_TESTS |
|---|---|
| imports | IL_SEMIRING_WITH_COMPL |
| operators | $\neg : test \to test$ |
| axioms | COMMUTATIVE$(test, \cdot)$<br>$\neg p = q \Leftrightarrow_{df} are_complements(p,q)$ |
| derives | BOOLEAN_ALG$(test, +, \cdot, \neg, 0, 1)$<br>I_SEMIRING $\Rightarrow$ COMMUTATIVE$(test, \cdot)$ |

The commutativity requirement for tests is equivalent to distributivity of composition over choice also in its right argument on the subset of tests. However, in concrete algebras it usually is more onerous to check distributivity, a property involving three variables, than commutativity, which only involves two.

The above specification is somewhat unsatisfactory in that it is not purely equational and involves subsorting. This makes automatic verification quite cumbersome or even excludes the use of some automatic verification systems. We will discuss alternative specifications below.

Using tests we can give algebraic semantics to a simple programming language. Composition and choice are already present in semirings. We can enrich this by case distinction:

| definition | IFTHENELSE |
|---|---|
| imports | IL_SEMIRING_WITH_TESTS |
| operators | if_then_else : $test \times S \times S \to S$ |
| defined by | if $p$ then $x$ else $y$ $=_{df}$ $p \cdot x + \neg p \cdot y$ |

Using finite iteration we can also define a while loop:

| definition | WHILE |
|---|---|
| imports | IL_SEMIRING_WITH_TESTS ¦ LEFT_KLEENE_ALG |
| operators | while_do_ : $test \times S \to S$ |
| defined by | while $p$ do $x$ $=_{df}$ $(p \cdot x)^* \cdot \neg p$ |

Moreover, we can give an algebraic definition of standard Hoare triples; it appears in [39] and admits a simple algebraic soundness proof of Hoare logic.

| definition | HOARE_TRIPLE |
|---|---|
| imports | IL_SEMIRING_WITH_TESTS |
| predicates | $\{\_\}\ \_\ \{\_\} \subseteq test \times S \times test$ |
| defined by | $\{p\}\ x\ \{q\}$ $\Leftrightarrow_{df}$ $p \cdot x \cdot \neg q = 0$ |

## 14 Domain and Antidomain

An important concept for transition systems is the set of *enabled* states, i.e., the set of states from which transitions are possible. For a transition relation $R$, this is the *domain* of $R$. We apply this nomenclature also to the general case. Using tests, a *predomain* operator can be characterised algebraically by quite simple equational axioms [9]. The domain operator shows a stronger interplay between predomain and composition, which can be used, e.g., for an algebraic proof of relative completeness of the Hoare calculus [46].

| theory | PREDOMAIN |
|---|---|
| imports | IL_SEMIRING_WITH_TESTS |
| operators | $\ulcorner\_ : S \to test$ |
| variables | $p : test$ ¦ $x, y : S$ |
| axioms | $x \leq \ulcorner x \cdot x$ ¦ $\ulcorner(p \cdot x) \leq p$ |
| derives | $\ulcorner(x + y) = \ulcorner x + \ulcorner y$ ¦ $\ulcorner(x \cdot y) \leq \ulcorner(x \cdot \ulcorner y)$<br>$\ulcorner(p \cdot x) = p \cdot \ulcorner x$ ¦ $\ulcorner p = p$ |

| theory | DOMAIN |
|---|---|
| imports | PREDOMAIN |
| axioms | $\ulcorner(x \cdot \ulcorner y) \leq \ulcorner(x \cdot y)$ |

In a similar fashion one can specify a codomain operator. If one assumes an idempotent semiring with tests, the axioms are just the mirror images of the ones

for domain. In case of a general idempotent left semiring, however, distributivity of codomain over choice needs to be stated as an additional axiom [44].

Let us now briefly discuss the mentioned alternative axiomatisation of tests and the domain operator. This has been carried out in a form specific to relation algebra in [31] and in the general semiring setting in [8]. The idea is to avoid explicit subsorting and to characterise the tests implicitly as the image of an antidomain operator @ which yields the negation of the domain of its argument. Over a full idempotent semiring the axiomatisation is surprisingly simple — however, to fully grasp it, good knowledge about tests and domain is almost mandatory. This is why we introduced their theories beforehand. In using the antidomain theory, rather than quantifying over a variable $p : test$ one uses a variable $x : S$ and writes $@x$ instead of $p$. For the case of general left semirings additional axioms are necessary; this is the subject of ongoing work.

| theory | ANTIDOMAIN |
|---|---|
| imports | I_SEMIRING |
| operators | $@ : S \to S$ |
| axioms | $@x \cdot x = 0 \mid @x + @@x = 1$<br>$@(x \cdot y) \leq @(x \cdot @@y)$ |

| definition | DERIVED_DOMAIN |
|---|---|
| imports | ANTIDOMAIN |
| operators | $\ulcorner : S \to S \mid \neg : test \to test$ |
| predicates | $test \subseteq S$ |
| defined by | $test(x) \Leftrightarrow_{df} \exists\, y \,.\, x = @y$<br>$\ulcorner x =_{df} @@x \mid \neg p = @@p$ |
| derives | DOMAIN |

## 15 Modal Operators: Diamond and Box

Many properties of transition systems can be described by the modal operators diamond and box, which express existential and universal quantification over the successor or predecessor states of a given set of states. We show exemplarily how to define the forward modal operators in terms of domain; the backward ones can be defined analogously in terms of codomain. Given a transition system $x$, a state $s$ satisfies the predicate $\langle x \rangle q$ iff $s$ has a successor under $x$ that satisfies $q$. This is equivalent to saying that $s$ lies in the inverse image of $q$ under $x$. The box operator $[x]q$ is the De Morgan dual of diamond. This is the basis of the following specification.

| definition | FORWARDMODAL |
|---|---|
| imports | DOMAIN |
| operators | $\langle \_ \rangle \_, [\_] \_ : S \times test \to test$ |
| defined by | $\langle x \rangle q =_{df} \ulcorner(x \cdot q) \mid [x]q =_{df} \neg\langle x \rangle \neg q$ |
| variables | $u : purely\_fin \mid z : right\_dist \mid p, q : test \mid x, y : S$ |
| derives | $\langle u \rangle 0 = 0 \mid [u]1 = 1$<br>$\langle z \rangle (p + q) = \langle z \rangle p + \langle z \rangle q \mid [z](p \cdot q) = [z]p \cdot [z]q$<br>$\langle z \rangle p - \langle z \rangle q \leq \langle z \rangle (p - q) \mid [z](p \to q) \leq [z]p \to [z]q$<br>$\langle x + y \rangle p = \langle x \rangle p + \langle y \rangle p \mid [x + y]p = [x]p \cdot [y]p$<br>$\langle x \cdot y \rangle p = \langle x \rangle (\langle y \rangle p) \mid [x \cdot y]p = [x]([y]p)$<br>$\{p\}\ x\ \{q\} \Leftrightarrow p \leq [x]q$ |

The last property shows that $[x]q$ is the algebraic counterpart of the weakest liberal precondition operator $\mathsf{wlp}.x.q$ used in program correctness calculi [10].

Equivalently, one can axiomatise one of the modal operators directly and define the other one and domain in terms of it. For instance, one can use the last property above to axiomatise the box operator. Then the diamond operator is defined as the De Morgan dual of box: $\langle x\rangle q =_{df} \neg[x]\neg q$. Finally, domain can be retrieved as $\ulcorner x =_{df} \langle x\rangle 1$.

Interestingly, in presence of star no special axioms are needed to establish star induction for diamond and box:

| theory | MODAL_STAR |
|---|---|
| imports | LEFT_KLEENE_ALG ¦ FORWARDMODAL |
| variables | $p, q : test$ ¦ $x : S$ |
| derives | $p \leq q \wedge p \leq [x]p \Rightarrow p \leq [x^*]q$<br>$(\{p \cdot r\}\ x\ \{p\}) \wedge p \cdot \neg r \leq q \Rightarrow (\{p\}\ \mathsf{while}\, p\, \mathsf{do}\, x\ \{q\})$ |

The second property is a special case of the first one and corresponds to the familiar inference rule for the while loop. More generally, box calculus does not only admit an algebraic soundness proof of Hoare logic, but also one of relative completeness [46].

Moreover, box can also be used to model total and general correctness and the wp operator. In fact, wp turns out to be the box operator in an algebra of commands [47]. Hence the abstract relative completeness result can immediately be re-used to show relative completeness of wp-based Hoare logic.

## 16 Logics of Knowledge and Belief

In this section we use some of our blocks to build algebraic theories of knowledge and belief. The idea is to abstract the access relations of Kripke models for multiagent systems to elements of an idempotent semiring with tests and to represent the knowledge and belief operators as instances of the general box operator with suitable additional axioms. The monomodal case is obtained by setting set $\Box p =_{df} [x]p$ for some fixed transition element $x$.

| definition | $\mathsf{MULTIAGENT}_n$ |
|---|---|
| imports | MODAL_ISEMIRING ¦ MODAL_STAR |
| operators | $a_1, \ldots, a_n, a : S$ ¦ $\mathsf{K}_1, \ldots, \mathsf{K}_n, \mathsf{E}, \mathsf{C} : test \to test$ |
| defined by | $\mathsf{K}_1 p =_{df} [a_1]p$ ¦ $\ldots$ ¦ $\mathsf{K}_n p =_{df} [a_n]p$<br>$a =_{df} a_1 + \cdots + a_n$ ¦ $\mathsf{E}p =_{df} [a]$ ¦ $\mathsf{C}p =_{df} [a^+]$ |
| derives | $\mathsf{C}p \leq \mathsf{C}(\mathsf{C}p)$ ¦ $\mathsf{C}p \cdot \mathsf{C}q \leq \mathsf{C}(\mathsf{C}p \cdot \mathsf{C}q)$ ¦ $\mathsf{C}p \cdot \mathsf{C}q \leq \mathsf{C}(\mathsf{C}p \cdot q)$ |

The positive and negative introspection axioms are captured as follows.

| definition | INTROSPECTION |
|---|---|
| imports | IL_SEMIRING_WITH_TESTS |
| predicates | $sat\_posintro, sat\_negintro \subseteq test \to test$ |
| axioms | $sat\_posintro(f) \Leftrightarrow_{df} \forall p\,.\, f(p) \leq f(f(p))$<br>$sat\_negintro(f) \Leftrightarrow_{df} \forall p\,.\, \neg f(p) \leq f(\neg f(p))$ |

This allows specifying belief logic:

| theory | $\mathsf{MULTIBELIEF}_n$ |
|---|---|
| imports | $\mathsf{MULTIAGENT}_n$ ¦ INTROSPECTION |
| axioms | $sat\_posintro(\mathsf{K}_1)$ ¦ $\cdots$ ¦ $sat\_posintro(\mathsf{K}_n)$<br>$sat\_negintro(\mathsf{K}_1)$ ¦ $\cdots$ ¦ $sat\_negintro(\mathsf{K}_n)$ |

The axiom of truth (or reflexivity of the access elements) is expressed by

| definition | TRUTH |
|---|---|
| imports | IL_SEMIRING_WITH_TESTS |
| predicates | $sat\_truth \subseteq test \to test$ |
| axioms | $sat\_truth(f) \Leftrightarrow_{df} \forall p \,.\, f(p) \leq p$ |

Then multiagent knowledge logic is specified by

| theory | $\mathsf{MULTIKNOW}_n$ |
|---|---|
| imports | $\mathsf{MULTIBELIEF}_n$ ¦ TRUTH |
| axioms | $sat\_truth(\mathsf{K}_1)$ ¦ $\cdots$ ¦ $sat\_truth(\mathsf{K}_n)$ |
| derives | $\mathsf{C}(p \to \mathsf{E}p) \leq p \to \mathsf{C}p$ ¦ $\mathsf{C}p \cdot \mathsf{C}q = \mathsf{CC}p \cdot \mathsf{CC}q = \mathsf{C}(\mathsf{C}p \cdot \mathsf{C}q)$ |

It should be clear how further special-purpose multimodal logics can be constructed along these lines.

## 17 Quantales and Temporal Logics

Now we really leave the first-order setting. For a number of applications it is important that the underlying left semiring is not only a semilattice, but even a complete lattice in which composition distributes over arbitrary/non-empty suprema in its left/right argument. Such structures are known as left quantales and, in case the underlying semiring is even full, quantales [49, 55]. For systematic reasons we call the supremum operator lub (least upper bound) rather than sup.

| theory | LQUANTALE |
|---|---|
| imports | SEMIGROUP ¦ POSET |
| operators | $\mathsf{lub}\_ : \mathcal{P}(S) \to S$ |
| axioms | $\mathsf{lub}\, T \leq y \Leftrightarrow \forall x \,.\, x \in T \Rightarrow x \leq y$<br>$\mathsf{lub}\, T \cdot y = \mathsf{lub}\, \{x \cdot y \mid x \in T\}$<br>$T \neq \emptyset \Rightarrow y \cdot \mathsf{lub}\, T = \mathsf{lub}\, \{y \cdot x \mid x \in T\}$ |

Again, frequently a unit of composition is useful.

| theory | UL_QUANTALE |
|---|---|
| imports | LQUANTALE ¦ MONOID |

From a unital left quantale we can derive an idempotent left semiring.

| definition | UL_QUANTALE_AS_SEMIRING |
|---|---|
| imports | UL_QUANTALE |
| operators | $0 : S \mid + : S \times S \to S$ |
| defined by | $0 =_{df} \mathsf{lub}\,\emptyset \mid x + y =_{df} \mathsf{lub}\,(\{x\} \cup \{y\})$ |
| derives | IL_SEMIRING |

In a unital left quantale, star and omega can be *defined* as least/greatest fixpoints. To this end we first enrich left quantales by an infimum operation glb (greatest lower bound).

| definition | UL_QUANTALE_WITH_GLB |
|---|---|
| imports | UL_QUANTALE |
| operators | $\mathsf{glb}\,\_ : \mathcal{P}(S) \to S$ |
| defined by | $\mathsf{glb}\,T =_{df} \mathsf{lub}\,\{x \in S \mid \forall\, y \in T \,.\, x \leq y\}$ |

Least and greatest fixpoints are defined using the Tarski/Knaster theorem.

| definition | FIXPOINTS |
|---|---|
| imports | UL_QUANTALE_WITH_GLB ¦ ISO_ANTI |
| operators | $\mu, \nu : isotone \to S$ |
| defined by | $\mu f =_{df} \mathsf{glb}\,\{z \in S \mid f(z) \leq z\}$<br>$\nu f =_{df} \mathsf{lub}\,\{z \in S \mid z \leq f(z)\}$ |

Now we can specify iteration in a quantale.

| definition | UL_QUANTALE_WITH_ITERATION |
|---|---|
| imports | UL_QUANTALE_AS_SEMIRING ¦ FIXPOINTS |
| operators | $^*, ^\omega : S \to S$ |
| hidden | $f : S \times S \to (S \to S)$ |
| defined by | $f(x,y)(z) =_{df} x + y \cdot z$<br>$y^* =_{df} \mu f(1,y) \mid y^\omega =_{df} \nu f(0,y)$ |
| derives | LEFT_KLEENE_ALG<br>FULLY_DIST $\Rightarrow$ LEFT_OMEGA_ALG |

where

| theory | FULLY_DIST |
|---|---|
| imports | UL_QUANTALE_AS_SEMIRING ¦ FIXPOINTS |
| axioms | $\mathsf{glb}\,\{x + y \mid y \in T\} = x + \mathsf{glb}\,T$ |

Further applications of left quantales concern, e.g., hybrid systems and the various temporal logics. For instance, to capture CTL$^*$ one can interpret the quantale elements as abstracting sets of computation paths (the semantics of path formulas) and tests as abstracting sets of states (the semantics of state formulas); a distinguished element n abstracts the single-step transition relation. Then we can define an until operator U as $x\,\mathsf{U}\,y =_{df} \mu z \,.\, y + (x \sqcap \mathsf{n} \cdot z)$. This admits proving all standard CTL laws purely algebraically [45]. Moreover, for the sublogics CTL and LTL the general CTL$^*$ semantics can be transformed into simplified versions in $\omega$-regular form. These do no longer use the full power of quantales but just star and omega. Finally, for LTL even just star is used.

This provides interesting connections between $\mu$-calculus representations and star/omega-algebra. Other logics like ITL, IL, DC or NL can also be captured in this setting. For lack of space we cannot spread out the details in form of structured theories here.

## 18 Conclusion and Outlook

The algebraic structures presented form a comprehensive and flexible framework. They cover various semantic models in a uniform algebraic fashion. Further applications have concerned residuals (e.g., to define generalised modal operators as in [65, 57]), predicate transformer semantics (e.g., as demonic refinement algebra [66, 59] or command/design algebra [47, 18, 19]), probabilistic programs [42, 64, 43], game algebra [51, 52], hybrid systems [26, 27], neighbourhood logic [25] and linked object structures and separation logic [13, 7]. There is even a greater variety of applications outside the realm of program semantics. Many of them use standard relation algebra; for these the ideas in the present paper are not as useful, since the underlying theory is fixed. However, we mention a few that use variants of the semiring setting, for which the idea of building blocks of theories may have some profit. For instance [17, 24] provide algebraic descriptions of some aspects of routing systems. But also the cardinality operator in Dedekind categories and allegories as used for flow problems [35, 36], or collagories [34] could be organised in the form of structure theories as proposed here.

As mentioned in the introduction, machine support for this type of theories has been intensively studied. This has resulted in large, modularised collections of theories and automatic proofs on the web [28, 23, 32, 50, 62]. Moreover, there are strong links with the TPTP project [63].

As a continuation of that and the ideas in the present paper we envisage a system for composing and analysing structured theories, of course with check for syntactic well-formedness including type constraints. Moreover, the system should perform a normalisation of a structured theory into a "flattened" unstructured one and then determine which fragment of logic is actually used, in particular, whether the overall theory is equivalent to a first-order one. It should then make suggestions which of the existing automatic theorem provers look most promising for use with that theory.

We are convinced that there is much more potential in the algebraic approach. What needs to be done is to explore further areas to see whether the structuring mechanisms we have proposed in the present paper are sufficient and maybe can notationally be streamlined further.

**Acknowledgement** I am grateful for valuable comments by H.-H. Dang, R. Glück, P. Höfner, P. Roocks and A. Zelend.

## References

1. Agda. `http://wiki.portal.chalmers.se/agda/pmwiki.php`

2. F.L. Bauer et al.: The Munich project CIP. Volume I: The Wide Spectrum Language CIP-L. LNCS 183. Springer 1985
3. R. Burstall, J. Goguen: The Semantics of CLEAR, A Specification Language. In D. Bjørner (ed.): Abstract Software Specifications. LNCS 86. Springer 1980, 292–332
4. CoFI (The Common Framework Initiative): CASL Reference Manual. LNCS 2960 (IFIP Series). Springer 2004
5. E. Cohen: Separation and Reduction. In: R. Backhouse, J. Oliveira (eds.): Mathematics of Program Construction (MPC'00). LNCS 1837. Springer 2000, 45–59
6. H.-H. Dang, P. Höfner: Automated Higher-Order Reasoning about Quantales In B. Konev, R. Schmidt, S. Schulz (eds.): Proc. Workshop on Practical Aspects of Automated Reasoning, 2010.
7. H.-H. Dang, P. Höfner, B. Möller: Algebraic Separation Logic. J. Log. Algebr. Program. 2011 (in press)
8. J. Desharnais, G. Struth: Modal Semirings Revisited. In P. Audebaud, C. Paulin-Mohring (eds.): Mathematics of Program Construction. LNCS 5133. Springer 2008, 360–387
9. J. Desharnais, B. Möller and G. Struth: Kleene Algebra with Domain. ACM Transactions on Computational Logic 7, 798–833 (2006)
10. E. Dijkstra: A Discipline of Programming. Prentice-Hall 1976
11. R. Dijkstra: Computation Calculus Bridging a Formalization Gap. Sci. Comput. Program. 37, 3–36 (2000)
12. F. Durán, J. Meseguer: Structured Theories and Institutions. Theoretical Computer Science 309, 357–380 (2003)
13. T. Ehm: Pointer Kleene Algebra. In R. Berghammer, B. Möller, G. Struth (eds.): Relational and Kleene-Algebraic Methods in Computer Science (RelMiCS/AKA 03). LNCS 3051. Springer 2004, 99–111
14. H. Ehrig, B. Mahr: Fundamentals of Algebraic Specification I: Equations and Initial Semantics. EATCS-Monographs in Theor. Comp. Sci., Vol. 6. Springer 1985
15. S. Foster, G. Struth, T. Weber: Automated Engineering of Relational and Algebraic Methods in Isabelle/HOL. In H. de Swart (ed.): Relational and Algebraic Methods in Computer Science (RAMiCS12). LNCS. Springer 2011 (this volume)
16. F. Giunchiglia, P. Pecchiari, C. Talcott: Reasoning Theories. J. Autom. Reason. 26, 291–331 (2001)
17. A. Gurney, T. Griffin: Pathfinding Through Congruences. In H. de Swart (ed.): Relational and Algebraic Methods in Computer Science (RAMiCS12). LNCS. Springer 2011 (this volume)
18. W. Guttmann, B. Möller: Modal design algebra. In S. Dunne, B. Stoddart (eds.): Unifying Theories of Programming. LNCS 4010. Springer 2006, 236–256
19. W. Guttmann, B. Möller: Normal design algebra. J. Log. Algebr. Program. 79, 144–173 (2010)
20. R. Harper, D. Sannella, A. Tarlecki: Structured Theory Presentations and Logic Representations. Ann. Pure Appl. Logic 67, 113–160 (1994)
21. J. He, T. Hoare: Unifying Theories of Programming. Prentice Hall 1998
22. T. Hoare, B. Möller, G. Struth, I: Wehrman: Concurrent Kleene Algebra and its Foundations. J. Log. Algebr. Program. 2011 (in press)
23. P. Höfner: Algebraic Reasoning with Prover9. `http://www.kleenealgebra.de`
24. P. Höfner, A. McIver: Towards an Algebra of Routing Tables. In H. de Swart (ed.): Relational and Algebraic Methods in Computer Science (RAMiCS12). LNCS. Springer 2011 (this volume)
25. P. Höfner, B. Möller: Algebraic Neighbourhood Logic. J. Log. Algebr. Program. 76, 35–59 (2008)

26. P. Höfner, B. Möller: An Algebra of Hybrid Systems. J. Log. Algebr. Program. 78, 74–97 (2009)
27. P. Höfner, B. Möller: Fixing Zeno Gaps. Theoretical Computer Science (in press)
28. P. Höfner, M. Müller, S. Zeissler: ATPPortal: A User-friendly Web Based Interface for Automated Theorem Provers and for Automatically Generated Proofs. University of Augsburg, Institute of Computer Science, Report TR1010-08. http://opus.bibliothek.uni-augsburg.de/volltexte/2010/1673/
29. P. Höfner, G. Struth: Automated Reasoning in Kleene Algebra. In F. Pfenning (ed.): Automated Deduction. LNCS 4603. Springer 2007, 279–294
30. P. Höfner, G. Struth, G. Sutcliffe: Automated Verification of Refinement Laws. Annals of Mathematics and Artificial Intelligence 35, 35–62 (2009)
31. M. Hollenberg: An Equational Axiomatization of Dynamic Negation and Relational Composition. Journal of Logic, Language and Information 6, 381-401 (1997)
32. J. Hurd: OpenTheory: Package Management for Higher Order Logic Theories. In G. Dos Reis, L. Théry (eds.): PLMMS '09 — Proc. ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems. ACM 2009, 31–37
33. C. Jones: Development Methods for Computer Programs Including a Notion of Interference. PhD Thesis, University of Oxford. Programming Research Group, Technical Monograph 25, 1981
34. W. Kahl: Collagories — Relation-Algebraic Reasoning for Gluing Constructions. J. Log. Algebr. Program. 2011 (in press)
35. Y. Kawahara: On the Cardinality of Relations. In R. Schmidt (ed.): Relations and Kleene Algebra in Computer Science (RelMiCS/AKA 06). LNCS 4136. Springer 2006, 251-265
36. Y. Kawahara, M. Winter: Cardinal Addition in Distributive Allegories. In R. Berghammer, A. Jaoua, B. Möller (Eds.): Relations and Kleene Algebra in Computer Science (RelMiCS/AKA 09). LNCS 5827. Springer 2009, 227–241
37. S. Kleene: Representation of Events in Nerve Nets and Finite Automata. In C. Shannon,J. McCarthy: Automata Studies. Princeton University Press 1956, 3–42
38. D. Kozen: A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events. Information and Computation 110, 366–390 (1994)
39. D. Kozen: Kleene Algebra with Tests. Trans. Programming Languages and Systems 19, 427–443 (1997)
40. Z. Luo, R. Burstall: A Set-theoretic Setting for Structuring Theories in Proof Development. University of Edinburgh, Laboratory for Foundations of Computer Science, Report ECS-LFCS-92-206, 1992
41. E. Manes, D. Benson: The Inverse Semigroup of a Sum-Ordered Semiring. Semigroup Forum 31, 129–152 (1985)
42. A. McIver, E. Cohen, C. Morgan: Using Probabilistic Kleene Algebra For Protocol Verification. In R. Schmidt (ed.): Relations and Kleene Algebra in Computer Science (RelMiCS/AKA 06). LNCS 4136. Springer 2006, 296–310
43. L. Meinicke, K. Solin: Refinement Algebra for Probabilistic Programs. Electr. Notes Theor. Comput. Sci. 201: 177-195 (2008)
44. B. Möller: Kleene Getting Lazy. Sci. Comput. Program. 65, 195–214 (2007)
45. B. Möller, P. Höfner, G. Struth: Quantales and Temporal Logics. In: M. Johnson and V. Vene (Eds.): Algebraic Methodology and Software Technology (AMAST 2006). LNCS 4019. Springer 2006, 263–277
46. B. Möller, G. Struth: Algebras of Modal Operators and Partial Correctness. Theoretical Computer Science 351, 221-239 (2006)

47. B. Möller, G. Struth: wp is wlp. In W. MacCaull, M. Winter, I. Düntsch (eds.): Relational Methods in Computer Science. LNCS 3929. Springer 2006, 200–211
48. B. Moszkowski: A Complete Axiomatization of Interval Temporal Logic with Infinite Time. LICS 2000, 241–252
49. C. Mulvey: &. Rendiconti del Circolo Matematico di Palermo 12, 99–104 (1986)
50. L. Paulson, T. Nipkow, M. Wenzel: Isabelle. http://www.cl.cam.ac.uk/research/hvg/Isabelle/
51. R. Parikh: Propositional Logics of Programs: New Directions. In M. Karpinski (ed.): Fundamentals of Computation Theory. LNCS 158. Springer 1983, 347–359
52. M. Pauly, R. Parikh: Game Logic – An Overview. Studia Logica 75, 165-182 (2003)
53. D. Park: On the Semantics of Fair Parallelism. In D. Bjørner (ed.): Abstract Software Specifications. LNCS 86. Springer 1980, 504–526.
54. J. Reynolds: An Introduction to Separation Logic. In M. Broy, W. Sitou, T. Hoare (eds.): Engineering Methods and Tools for Software Safety and Security. IOS Press 2009, 285–310.
55. K. Rosenthal: Quantales and their Applications. Pitman Research Notes in Math. No. 234 Longman Scientific and Technical 1990
56. D. Sannella and R.M. Burstall: Structured Theories in LCF. In G. Ausiello, M. Protasi (Eds.): Proc. CAAP'83, Trees in Algebra and Programming, 8th Colloquium, L'Aquila, Italy, March 9–11, 1983. LNCS 159. Springer 1983, 377–391
57. M. Sintzoff: Iterative Synthesis of Control Guards Ensuring Invariance and Inevitability in Discrete-Decision Games. In: O. Owe, S. Krogdahl, T. Lyche (eds.): From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl. LNCS. 2635. Springer 2004, 272–301
58. D. Smith: Automating the Design of Algorithms. In B. Möller, H. Partsch, S. Schuman (eds.): Formal Program Development. LNCS 755. Springer 1993, 324–354
59. K. Solin, J. von Wright: Enabledness and Termination in Refinement Algebra. Sci. Comput. Program. 74, 654–668 (2009)
60. SPECWARE: http://www.specware.org/
61. Y. Srinivas, R. Jüllig: Specware: Formal Support for Composing Software. In B. Möller (ed.): Mathematics of Program Construction. Third International Conference, Kloster Irsee, July 17–21, 1995. LNCS 947. Springer 1995, 399–422
62. G. Struth: Isabelle Repository for Relational and Algebraic Methods. http://staffwww.dcs.shef.ac.uk/people/G.Struth/isa/
63. G. Sutcliffe: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning 43, 337–362 (2009)
64. T. Takai, H. Furusawa: Monodic Tree Kleene Algebra. In R. Schmidt (ed.): Relations and Kleene Algebra in Computer Science (RelMiCS/AKA 06). LNCS 4136. Springer 2006, 402–416. Errata available at http://www.sci.kagoshima-u.ac.jp/~furusawa/person/Papers/correct_ monodic_kleene_algebra.pdf
65. B. von Karger: Temporal Algebra. Mathematical Structures in Computer Science 8, 277–320 (1998)
66. J. von Wright: Towards a Refinement Algebra. Sci. Comput. Program. 51, 23–45 (2004)
67. M. Wirsing: Structured Algebraic Specifications: A Kernel Language. Theor. Comput. Sci. 42, 123–249 (1986)