Algebras for Program Correctness in Isabelle/HOL

Alasdair Armstrong, Victor B. F. Gomes, and Georg Struth

Department of Computer Science, University of Sheffield
{a.armstrong, v.gomes, g.struth}@shefield.ac.uk

Abstract. We present a reference formalisation of Kleene algebra and demonic refinement algebra with tests in Isabelle/HOL. It provides three different formalisations of tests. Our structured comprehensive libraries for these algebras extend an existing Kleene algebra library. It includes an algebraic account of Hoare logic for partial correctness and several refinement and concurrency control laws in a total correctness setting. Formalisation examples include a complex refinement theorem, a generic proof of a loop transformation theorem for partial and total correctness and a simple prototypical verification tool for while programs, which is itself formally verified.

1 Introduction

This article documents the formalisation of computationally important algebraic concepts and structures within a larger project of making variants of Kleene algebras and relation algebras available in the Isabelle proof assistant. It presents variants of test semirings [17] in Isabelle together with their expansions to Kleene algebras and demonic refinement algebras with tests [15, 22, 23]. The latter two algebras have been applied in the verification and correctness of sequential programs; the first one in partial correctness, the second one in a total correctness setting. Demonic refinement algebras have also been used for concurrency verification with action systems [7].

Implementing these algebras with their most important models—the binary relation model for Kleene algebras with tests and the conjunctive predicate transformer model for demonic refinement algebras—yields a basis for building lightweight tools for program verification and correctness in Isabelle. The general approach is quite simple. The algebraic layer captures part of reasoning about programs, in particular their control flow, abstractly and concisely. Other aspects, such as data flow, however, are performed within more concrete models, for example the relational model of program store. In addition, algebra helps at the meta-level to derive inference, refinement or transformation rules and implement tactics or decision procedures. Isabelle allows one to reason seamlessly across these layers, for instance, by programming algebra-driven tactics which automatically generate verification conditions for concrete models or data types or by inferring abstract properties of assignment commands. All these features are provided by our implementation and are illustrated in this article.

More concretely, the main contributions of our formalisation are as follows.

First, based on a comprehensive reference library for variants of dioids and Kleene algebras [4], we have implemented demonic refinements algebras [22, 23] as an extension of a variant of Kleene algebra via Isabelle's type class mechanism, the standard tool for such formalisations. While Kleene algebras provide operations for the programming concepts of abort, skip, sequential composition, nondeterministic choice and finite iteration, demonic refinement algebras add an operation of potentially infinite iteration. We have implemented a library for equational reasoning in this algebra which contains more than 50 facts from the literature.

Second, we have formalised three different approaches to tests for variants of dioids (idempotent semirings) and developed comprehensive libraries for these. The first one is one-sorted. It implements functions for tests and antitests (boolean test complements) as in the standard approach to domain semirings [11]. The second, two-sorted one follows the approach of embedding a boolean algebra of tests into the dioid [17]. Both approaches are purely axiomatic; they do not mention an underlying carrier set. While such axiomatic versions often suffice for verification applications, a third variant with explicit carrier sets is provided as a basis for mathematical investigations. From variants of test dioids, Kleene and demonic refinement algebras with tests are obtained as straightforward expansions. The libraries for these structures contain more than 350 facts. In particular the third variant required implementing a range of background theories.

Third, we illustrate our formalisation through two classical examples from the literature, which any formalisation of these structures should feature. We have formalised proofs of three variants of Back's atomicity refinement theorem for action systems [22, 9, 13] in demonic refinement algebras with tests. We also present a new generic proof of Kozen's transformation theorem for while loops in Kleene algebras with tests [15] and demonic refinement algebras [21]. It is based on an axiomatisation of regular algebras by Conway [10] in which the iteration axioms are too weak to distinguish finite from potentially infinite iteration.

Fourth, we demonstrate how simple prototypical tools for the verification, refinement and transformation of sequential programs can be obtained in a generic way from the algebras considered. By developing the tool in Isabelle, it is itself formally verified. We first derive the inference rules of Hoare logic except the assignment rule in Kleene algebras with tests. To capture assignment in concrete models, we then formalise the relational model of Kleene algebra with tests and the predicate transformer model of demonic refinement algebra and specialise the first model further to program stores. We can then derive assignment laws easily within this model. We have also implemented a tactic for automatically generating the usual verification conditions for while programs and show the approach at work on a simple verification example.

In sum, this formalisation spans the gulf between abstract algebras of programs and concrete tools for program correctness and verification in a simple, coherent, principled way. We are using it as a template for developing more sophisticated and applicable tools for sequential and concurrent programs.

2 Algebraic Preliminaries

A *dioid*, or *idempotent semiring* is a structure $(S, +, \cdot, 0, 1)$ such that $(S, \cdot, 1)$ is a monoid, $(S, +, 0)$ is a semilattice with least element 0, and the distributivity laws $x \cdot (y + z) = x \cdot z + y \cdot z$ and $(x + y) \cdot z = x \cdot z + y \cdot z$ as well as the annihilation laws $0 \cdot x = 0$ and $x \cdot 0 = 0$ hold. Addition and multiplication are isotone with respect to the semilattice order defined by $x \leq y \longleftrightarrow x + y = y$, that is, $x \leq y$ implies $z + x = z + y$, $z \cdot x \leq z \cdot y$ and $x \cdot z \leq y \cdot z$.

A *right Kleene algebra* is a dioid expanded by a Kleene star which satisfies the unfold axiom $1 + x^* \cdot x \leq x^*$ and the iteration axiom $z + y \cdot x \leq y \longrightarrow z \cdot x^* \leq y$. The dual unfold law $1 + x \cdot x^* \leq x^*$ is derivable. A right Kleene algebra is a *Kleene algebra* if the left induction axiom $z + x \cdot y \leq y \longrightarrow x^* \cdot z \leq y$ holds too. For an overview of variants of dioids and Kleene algebras, their most useful laws and their most important models see [4].

In this context it is important to know that binary relations form Kleene algebras. This relational model is discussed further in Section 7. Binary relations, in turn, yield a standard semantics for sequential programs. Addition models nondeterministic choice, multiplication models sequential composition, 1 models **skip**, 0 models **abort**, * models finite iteration.

For modelling conditionals and while loops according to the relational partial correctness semantics, a notion of *test* needs to be added. In the relational model a test is simply an element between the empty and the identity relation. Abstractly, a *test dioid* [17] is a structure (S, B) such that S is a dioid and B a boolean algebra which is embedded into the subalgebra of elements between 0 and 1 of the dioid. There are the following correspondences between operations of the dioid and those of the boolean algebra: 0 corresponds to the least element of the boolean algebra, 1 to its greatest element, $+$ corresponds to join and $\cdot$ to meet. Complementation $-$ has no counterpart in the dioid, it exists only for the subalgebra of tests. A *Kleene algebra with tests* is a test dioid which is also a Kleene algebra. We write x, y, z for general Kleene algebra elements and p, q, r for tests. A technical development can be found in Section 4. Using tests, an abstract algebraic semantics for conditionals and while loops is given by

$$\textbf{if } p \textbf{ then } x \textbf{ else } y = p \cdot x + (-p) \cdot y, \qquad \textbf{while } p \textbf{ do } x = (p \cdot x)^* \cdot (-p).$$

Multiplying a program x with a test p from the left means restricting the input of the program to those states where the test holds; multiplying from the right means an output restriction.

Total program semantics require another variant of Kleene algebra [22, 23]. A *demonic refinement algebra* is a Kleene algebra in which the right annihilation axiom $x \cdot 0 = 0$ is absent and which is expanded by an operation for possibly infinite iterations which satisfies the unfold axiom $1 + x \cdot x^\infty = x^\infty$, the coinduction axiom $y \leq x \cdot y + z \longrightarrow y \leq x^\infty \cdot z$ and the isolation axiom $x^\infty = x^* + x^\infty \cdot 0$.

This captures total correctness where an agent has no control over termination; $(p \cdot x)^\infty \cdot (-p)$, for instance, models a while loop which may not terminate. For similar reasons, $x \cdot 0 = 0$ is invalid due to potentially infinite processes. In

the isolation axiom, $x^* \cdot 0$ annihilates if all processes in x are finite whereas $x^\infty \cdot 0$ projects on the strictly infinite processes in x^∞.

Demonic refinement algebra can be expanded by tests in the obvious way. The semantics of choice, in this case, is a predicate transformer algebra, which we discuss in detail in Section 7.

The refinement community's notation unfortunately deviates from the regular algebra notation. Their refinement order $\sqsubseteq$ is the converse of $\leq$; the symbols $\top$, $\sqcap$, ; and $^\omega$ are used instead of 0, +, $\cdot$ and $^\infty$. Finally tests are known as *guards*.

3 Demonic Refinement Algebra in Isabelle

We now sketch our formalisation of demonic refinement algebra in the theorem proving environment Isabelle/HOL [18]. We introduce some basic features of Isabelle while discussing our formalisation. For additional information about Isabelle we refer to its excellent documentation[1]. The complete Isabelle code of our implementation can be found online[2]. A reference formalisation is available from the Archive of Formal Proofs [3]. We recommend reading these in parallel.

Isabelle is an interactive proof assistant with embedded automatic theorem provers and counterexample generators. It is based on a small logical core to guarantee correctness. It has been used to formalise a wide range of mathematical theories and applied in numerous computing applications, including program correctness and verification. Isabelle/HOL, in particular, is based on a typed higher-order logic which supports reasoning with sets, polymorphic data types, inductive definitions and recursive functions.

Algebraic hierarchies, like those in the previous section, are usually formalised with Isabelle's type class and locale infrastructure. Type classes typically suffice for simple structures with one single type parameter. More advanced formalisations often require locales. Both mechanisms support theory expansion and the formalisation of subclass relationships. Theorems proved for reducts or superclasses thus become automatically available in expansions or subclasses. Within this infrastructure, algebras can be linked formally with their models by instantiation or interpretation statements.

We have integrated our formalisation of demonic refinement algebra into the existing Kleene algebra hierarchy [4]. More precisely, we have formalised demonic refinement algebra as an expansion of Kleene algebra with a left annihilator, adding simply the unfold, coinduction and isolation axiom for $^\infty$. By this expansion, all facts proved for this variant of Kleene algebra become automatically available in demonic refinement algebra.

class *dra* = *kleene-algebra-zerol* + *strong-iteration-op* +
  assumes *iteration-unfoldl*: $1 + x \cdot x^\infty = x^\infty$
  and *coinduction*: $y \leq z + x \cdot y \longrightarrow y \leq x^\infty \cdot z$
  and *isolation*: $x^\infty = x^\star + x^\infty \cdot 0$

[1] http://isabelle.in.tum.de
[2] http://www.dcs.shef.ac.uk/∼victor/ramics2014

We have developed a comprehensive library of theorems of demonic refinement algebra from the literature. Isabelle offers various ways of proving such facts. First, there is a range of built-in tactics, provers and simplifiers. These are generally insufficient for automating algebraic reasoning, but quite powerful for higher-order reasoning with models. Second, Isabelle's Sledgehammer tactic calls external automated theorem provers and SMT solvers and reconstructs their output internally to increase trustworthiness. In this way, many equational algebraic theorems can be proved fully automatically, but the approach is limited to first-order reasoning. Finally, Isabelle offers different modes of interactive reasoning, notably the proof scripting language Isar which supports human-readable proofs, as in the following example.

lemma *iteration-sim*: $z{\cdot}y \leq x{\cdot}z \longrightarrow z{\cdot}y^\infty \leq x^\infty{\cdot}z$
proof
  assume *assms*: $z{\cdot}y \leq x{\cdot}z$
  have $z{\cdot}y^\infty = z + z{\cdot}y{\cdot}y^\infty$
    by (*metis distrib-left mult-assoc mult-oner iteration-unfoldl*)
  also have $\ldots \leq z + x{\cdot}z{\cdot}y^\infty$
    by (*metis assms add-commute add-iso mult-isor*)
  finally show $z{\cdot}y^\infty \leq x^\infty{\cdot}z$
    by (*metis mult-assoc coinduction*)
qed

In this proof, individual proof steps have been proved automatically by Sledgehammer and internally verified by the metis prover. Isar links these steps into a complete proof. In total we have proved 57 theorems about demonic refinement algebra, 43 of which were fully automatic. The remaining 14 facts required user intervention at the level of the previous example.

Isabelle also offers counterexample generators such as *nitpick* and *quickcheck*, which is very important for exploring mathematical theories. The dual simulation law $y \cdot z \leq z \cdot x \longrightarrow y^\infty \cdot z \leq z \cdot x^\infty$, for instance, has been refuted by nitpick with a three-element counterexample, whereas both simulation laws—with $^\infty$ replaced by *—hold in Kleene algebra.

The most interesting and difficult theorems come from demonic refinement algebra with tests. Before discussing these in Section 5 we present three alternative formalisations of test dioids in the following section.

4 Three Formalisations of Tests

The embedding of a boolean test algebras into a Kleene algebra can be formalised in different ways in Isabelle. Our first implementation is based on an unpublished manuscript by Jipsen and Struth. It is inspired by the axiomatisation of domain semirings [11]. The main idea is to add a function t to a semiring or dioid S and axiomatise it in such a way that the image $t(S)$ forms a boolean subalgebra of tests. The function t is assumed to be a retraction, that is, $t \circ t = t$, since then

$p \in t(S)$ if and only if $t(p) = p$. We can use this fixpoint property for typing tests or verifying closure conditions.

For encoding test complementation, however, it is more suitable to axiomatise an *antitest function* n which satisfies $t = n \circ n$:

class *dioid-tests-zerol* = *dioid-one-zerol* + *comp-op* +
  assumes *test-one*: $n\ n\ 1 = 1$
  and *test-mult*: $n\ n\ (n\ n\ x \cdot n\ n\ y) = n\ n\ y \cdot n\ n\ x$
  and *test-mult-comp*: $n\ x \cdot n\ n\ x = 0$
  and *test-de-morgan*: $n\ x + n\ y = n\ (n\ n\ x \cdot n\ n\ y)$

We then abbreviate $t\ x \equiv n\ (n\ x)$ and define $test\ p \equiv t\ p = p$. In fact, if these axioms are added to an arbitrary semiring, idempotence is enforced. It is straightforward to verify that tests satisfy the boolean algebra axioms, but the fact that $t(S)$ forms a boolean algebra cannot be expressed explicitly in Isabelle by a subclass or sublocale statement, simply because the carrier set S is not explicit in an type class. Thus we cannot formally integrate Isabelle's library for boolean algebra and had to build up our own one with the most important boolean theorems for tests. We provide an alternative implementation where this problem can be circumvented.

The expansion of test dioids to Kleene algebras with tests is straightforward and therefore not shown in this paper. We have also verified that our test axioms are independent, using nitpick for finding counterexamples when trying to prove each individual axiom from the remaining ones. Despite its limitations, this formalisation is simple and yields a high degree of automation. Overall, 122 theorems about Kleene algebras with tests and boolean algebra were proved, all of which fully automatically.

Our second formalisation of test dioids integrates Isabelle's boolean algebra type class. In contrast to the previous one-sorted implementation it is therefore two-sorted. This requires locales instead of type classes.

locale *dioid-tests-zerol* =
  fixes *test* :: $'a$::*boolean-algebra* $\Rightarrow$ $'b$::*dioid-one-zerol*
  and *not* :: $'b$::*dioid-one-zerol* $\Rightarrow$ $'b$::*dioid-one-zerol*
  assumes *test-sup*: *test* (*sup* $p\ q$) = ‘$p + q$‘
  and *test-inf*: *test* (*inf* $p\ q$) = ‘$p \cdot q$‘
  and *test-top*: *test top* = *1*
  and *test-bot*: *test bot* = *0*
  and *test-not*: *test* $(-p)$ = ‘$-p$‘
  and *test-iso-eq*: $p \leq q \longleftrightarrow$ ‘$p \leq q$‘

Now the function *test* embeds the boolean algebra into the dioid as usual. A boolean complementation is also defined on the dioid. The other axioms of this locale link the boolean operations with the dioid ones, as described in Section 2. To obtain the typical Kleene algebra with test notation, where the embedding is implicit, we have implemented a syntax translation which automatically recognises tests in formulas. Hence one can write ‘$p + q$‘ for the join of two tests.

With this two-sorted approach, Isabelle's libraries for boolean algebras become automatically available. From an automation point of view, however, we noted little difference between the two approaches. As before, we do not explicitly show the expansion of test dioids to Kleene algebras with tests.

Our third implementation of test dioids provides explicit carrier sets. It follows the general Isabelle recipe for setting up such algebras.

record $'a$ *test-dioid-structure* = $'a$ *dioid* + *test* :: $'a$ *ord*

abbreviation *tests* $A \equiv$ *carrier* (*test* A)

locale *dioid-tests-zerol* =
  fixes A :: $'a$ *test-dioid-structure* (**structure**)
  assumes *is-dioid*: *dioid-tests-zerol* A
  and *test-subset*: *tests* $A \subseteq$ *carrier* A
  and *test-le*: *le* (*test* A) = *dioid.nat-order* A
  and *test-ba*: *boolean-algebra* (*test* A)
  and *test-one*: *top* (*test* A) = *1*
  and *test-zero*: *bot* (*test* A) = *0*
  and *test-join*: $[\![x \in$ *tests* A; $y \in$ *tests* $A]\!] \Longrightarrow$ *join* (*test* A) $x\ y = x + y$
  and *test-meet*: $[\![x \in$ *tests* A; $y \in$ *tests* $A]\!] \Longrightarrow$ *meet* (*test* A) $x\ y = x \cdot y$

This formalisation expands carrier-based formalisations of dioids and boolean algebras. In this setting, algebraic signatures are specified in records. In this case it is said that tests have a pre-defined order type. The axioms yield a dioid without left annihilation where the carrier set of tests is a subset of the main carrier and the operations are embedded as usual.

To support this approach we had to implement several background theories from scratch with more than 250 theorems about lattices, dioids, Kleene algebra and Kleene algebras with tests. Because of the additional constraints, Sledgehammer may struggle to automate simple proofs. Hence there is a trade-off between mathematical precision and automation. This approach has previously been used to implement schematic Kleene algebra with tests and derive flow chart equivalence as well as simple program verification proofs in this setting [5].

In sum, our three formalisations all have their advantages and disadvantages. The one-sorted and two-sorted implementation offer comparable proof automation and might be superior for program verification applications. Which one is preferable in practice remains to be seen. The carrier-based implementation leads to less automatic proofs, but for investigations in universal algebra, for instance, this price needs to be paid.

5 A Program Refinement Example

All axiomatisations from the previous section have been given for dioids without the axiom $x \cdot 0 = 0$. This makes all three formalisations compatible with demonic refinement algebra. The one-sorted formalisation of tests, for instance, is

class *dra-tests* = *dioid-tests-zerol* + *dra*

An expansion to proper test dioids is, of course, given in our Isabelle theory files.

The addition of tests or guards make demonic refinement algebra suitable for program development applications. We have also formalised the dual notion of *assertion*. Assertions are used as context information for weakest precondition reasoning [22, 23] in guarded command languages. We have formalised assertions as $p^o = (-p)\cdot\top + 1$. The constant $\top$ denotes the greatest element of the demonic refinement algebra, which exists in this class and is equal to 1^∞. Intuitively, an assertion p^o aborts when p is false and skips when p is true. We have verified that guards and assertions are adjoints of Galois connections, $p\cdot x \leq y \longleftrightarrow x \leq p^o \cdot y$ and $x \cdot p^o \leq y \longleftrightarrow x \leq y \cdot p$, as well as further properties from the literature.

Demonic refinement algebra is also interesting for modelling concurrency in Back's *action system* framework [7]. As a complex example we have verified three algebraic versions of Back's *atomicity refinement theorem* [6, 22, 23, 9, 13]. For an explanation we refer to these articles. Here we only discuss algebraic aspects and proof automation. Von Wright's variant states that the identity

$$x \cdot (y + z + v + w)^\infty \cdot p \leq x \cdot (yz^\infty p + v + w)^\infty$$

can be derived from the 12 assumptions

$$t\ p = p, \qquad x = x \cdot p, \qquad y = p \cdot y, \qquad p \cdot z = 0, \qquad v \cdot z \leq z \cdot v,$$
$$v \cdot w \leq w \cdot v, \qquad v \cdot p \leq p \cdot v, \qquad y \cdot w \leq w \cdot y, \qquad z \cdot w \leq w \cdot z,$$
$$p \cdot w \leq w \cdot p, \qquad z^\infty = z^*, \qquad v^\infty = v^*.$$

Note that $z^\infty = z^*$ and $w^\infty = w^*$ express that z and w are finite. Von Wright's original proof covers about 3 pages. Our Isabelle proof essentially translates this proof at this level of granularity; a more coarse grained automation seems difficult for metis. The main reason is that the terms appearing in this proof are quite long and many rules can match. This combinatorics is difficult to handle in particular for metis, which is inferior to Sledgehammer's external provers. In fact, a more general proof of this theorem with Prover9 [13] was much more coarse grained but required excessive running times. Theorems like this provide interesting benchmarks for Sledgehammer in particular and automated theorem provers in general. This general version can also be found in our Isabelle files.

Finally, we have verified Cohen's simplified version of the atomicity refinement theorem [9] which derives the equation

$$(x + y + z)^\infty = (p \cdot z)^\infty \cdot (x + (-p) \cdot z + y \cdot (-p))^\infty \cdot (y \cdot p)^\infty$$

from the assumptions $t\ p = p$, $x \cdot 0 = 0$, $y \cdot 0 = 0$, $p \cdot y \cdot (-p) = 0$, $p \cdot z \cdot (-p) = 0$, $y \cdot p \cdot x \leq x \cdot y$, $x \cdot p \cdot z \leq z \cdot x$ and $y \cdot p \cdot z \leq z \cdot y$. Cohen assumes partial correctness, so we must explicitly express that x and y must terminate: $x \cdot 0 = 0$ and $y \cdot 0 = 0$. Our proof requires 10 particular steps with Isar.

The results in this section show that libraries that support program refinement can be developed quite easily at the algebraic level with Isabelle. Demonic

refinement algebra is part of more powerful calculi which have been described, for instance, in the book of Back and von Wright [8]. Their approach is based on lattice and fixpoint theory. It can easily be obtained by theory expansion from our formalisation of demonic refinement algebra. This is left for future work.

6 A Program Transformation Example

We now consider a classical program transformation example which has first been considered in the partial correctness setting of Kleene algebra with tests. We formalise Kozen's loop transformation theorem in Kleene algebra with tests: *Every sequential while program, appropriately augmented with subprograms of the form* $z \cdot (p \cdot q + (-p) \cdot (-q))$*, can be viewed as a while program with at most one loop under certain* preservation *assumptions* [15]. Hence any while program, suitably augmented with finitely many new *dummy* subprograms, is equivalent to a simple while program of the form x; **while** p **do** y, where x and y do not contain any nested loops.

A key ingredient of Kozen's approach are commutativity conditions of the form $p \cdot x = x \cdot p$. We use preservation conditions instead, which are of the form $p \cdot x = p \cdot x \cdot p$ and $(-p) \cdot x = (-p) \cdot x \cdot (-p)$. In Kleene algebra with tests, these two conditions are equivalent. However we prove the transformation theorem in the weaker setting of *pre-Conway algebras*, where the former imply the latter, but not vice versa (according to nitpick). Pre-Conway algebras are defined as

class *pre-conway* = *pre-dioid-one-zerol* + *dagger-op* +
  assumes *dagger-denest*: $(x + y)^\dagger = (x^\dagger \cdot y)^\dagger \cdot x^\dagger$
  and *dagger-prod-unfold*: $(x \cdot y)^\dagger = 1 + x \cdot (y \cdot x)^\dagger \cdot y$
  and *dagger-simr*: $z \cdot x \leq y \cdot z \longrightarrow z \cdot x^\dagger \leq y^\dagger \cdot z$

As the first line shows, they are based on *pre-dioids* with only a left-annihilating zero [4]. In these structures, the left distributivity law $x \cdot (y + z) = x \cdot y + x \cdot z$ is weakened to sub-distributivity $x \cdot y + x \cdot z \leq x \cdot (y + z)$ which is equivalent to isotonicity $x \leq y \longrightarrow z \cdot x \leq z \cdot y$. Furthermore, the right annihilation law $x \cdot 0 = 0$ is absent. To avoid confusion we use the operator † instead of *. The denest and product-unfold axioms are part of Conway's *classical axioms* for regular algebra [10], but several other axioms, including the idempotency axiom $x^{\dagger\dagger} = x^\dagger$, are absent. In particular, Conway's classical axioms are based on a full dioid. In fact, the dioid-based version plus dagger idempotence is equivalent to the axioms of right Kleene algebra; and complete with respect to the equational theory of regular expressions (see [12] for an overview).

In preparation to the proof of the loop transformation theorem we have verified a number of laws about the dagger in pre-Conway algebra, for instance isotonicity of dagger, $x \leq y \longrightarrow x^\dagger \leq y^\dagger$, a slide rule, $x \cdot (y \cdot x)^\dagger = (x \cdot y)^\dagger \cdot x$, unfold laws for the dagger, $x^\dagger = 1 + x \cdot x^\dagger$ and $x^\dagger = 1 + x^\dagger \cdot x$, along with some preservation properties, such as that $p \cdot x \cdot p = p \cdot x$ implies $p \cdot x^\dagger = (p \cdot x)^\dagger \cdot p$ and $p \cdot (p \cdot x + (-p) \cdot y)^\dagger = (p \cdot x)^\dagger \cdot p$.

The proof itself is by structural induction on while programs. This can be formalised in Isabelle by defining a grammar for programs and imposing the quotient of pre-Conway algebra identities, using Isabelle's quotient package. We only discuss the individual cases of this inductive argument. For each program construct, an inner loop is moved to the outside of a program and these program transformations are verified in pre-Conway algebra with tests. Programs can be augmented by dummy subprograms under preservation assumptions. We follow Kozen's case analysis, but proofs for individual cases are different due to our more general assumptions and the weaker axioms of pre-Conway algebras. To save space we write xy instead of $x \cdot y$ and $\overline{x}$ instead of $-x$. Following Kozen, we take the sequential composition operator to be of lower precedence than the other program constructs.

For conditionals, Kozen shows that the following programs are equivalent:

$$pq + \overline{pq};\ \textbf{if } p \textbf{ then } (x_1;\ \textbf{while } r_1 \textbf{ do } y_1) \textbf{ else } (x_2;\ \textbf{while } r_2 \textbf{ do } y_2),$$
$$pq + \overline{pq};\ \textbf{if } q \textbf{ then } x_1 \textbf{ else } x_2;\ \textbf{while } qr_1 + \overline{q}r_2 \textbf{ do } (\textbf{if } q \textbf{ then } y_1 \textbf{ else } y_2).$$

Translated into pre-Conway algebra we must prove that

$$(pq + \overline{pq})(px_1(r_1y_1)^\dagger\overline{r_1} + \overline{p}x_2(r_2y_2)^\dagger\overline{r_2}) = (pq + \overline{pq})(qx_1 + \overline{q}x_2)((qr_1 + \overline{q}r_2)(qy_1 + \overline{q}y_2))^\dagger\overline{qr_1 + \overline{q}r_2}.$$

This consists of two phases. First, the two terms are simplified by right distributivity, yielding two subterms each. Second, we proved this by verifying the following two equations between these subterms, using preservation:

$$pqx_1(r_1y_1)^\dagger\overline{r_1} = pqx_1(qr_1y_1 + \overline{q}r_2y_2)^\dagger(q\overline{r_1} + \overline{qr_2})$$
$$p\overline{q}x_2(r_2y_2)^\dagger\overline{r_2} = p\overline{q}x_2(qr_1y_1 + \overline{q}r_2y_2)^\dagger(q\overline{r_1} + \overline{qr_2})$$

For nested loops, Kozen proves the following two programs equivalent:

$$\textbf{while } p \textbf{ do } (x;\ \textbf{while } q \textbf{ do } y)$$
$$\textbf{if } p \textbf{ then } (x;\ \textbf{while } p + q \textbf{ do } (\textbf{if } q \textbf{ then } y \textbf{ else } x))$$

The corresponding proof in pre-Conway algebra was fully automatic.

$$(px(qy)^\dagger\overline{q})^\dagger\overline{q} = px((p + q)(qy + \overline{q}x))^\dagger(\overline{p + q} + \overline{p})$$

The case of sequential composition has two subcases. The first one—called *postcomputation*—composes a while loop with a loop-free program:

$$(\textbf{while } p \textbf{ do } x); y$$
$$\textbf{if } \overline{p} \textbf{ then } y \textbf{ else } (\textbf{while } p \textbf{ do } (x;\ \textbf{if } \overline{p} \textbf{ then } y))$$

The corresponding identity in Conway algebra is

$$(px)^\dagger\overline{p}y = \overline{p}y + p(px(\overline{p}y + p))^\dagger\overline{p}.$$

Due to the weaker setting, our proof differs from Kozen's.

$$\begin{aligned} p(px(\overline{p}y+p))^\dagger\overline{p} &= p\overline{p}+ppx((\overline{p}y+p)px)^\dagger(\overline{p}y+p)\overline{p} \\ &= px(\overline{p}ypx+px)^\dagger\overline{p}y\overline{p} \\ &= px(\overline{p}y0+px)^\dagger\overline{p}y \\ &= px(px)^\dagger(\overline{p}y0)^\dagger\overline{p}y \\ &= px(px)^\dagger\overline{p}y(0\overline{p}y)^\dagger \\ &= px(px)^\dagger\overline{p}y. \end{aligned}$$

The first step uses the product unfold law. The second step uses right distributivity and boolean algebra. The third step uses the preservation assumption $\overline{p}y = \overline{p}y\overline{p}$. The forth step uses denesting and right annihilation. The fifth step uses the sliding rule. The last step uses right annihilation and the rule $0^\dagger = 1$, which can be derived from the left unfold law. Finally, adding the term $\overline{p}y$ to both sides and applying unfold yields the desired identity.

The second subcase is the composition of two while loops, which leads to the equivalence of

$$\begin{aligned} &\textbf{while } p \textbf{ do } x;\ \textbf{while } q \textbf{ do } y \\ &\textbf{if } \overline{p} \textbf{ then } (\textbf{while } q \textbf{ do } y) \textbf{ else } (\textbf{while } p \textbf{ do } (x;\ \textbf{if } \overline{p} \textbf{ then } (\textbf{while } q \textbf{ do } y))) \end{aligned}$$

and the identity $(px)^\dagger\overline{p}(qy)^\dagger\overline{q} = \overline{p}(qy)^\dagger\overline{q} + p(px(\overline{p}(qy)^\dagger\overline{q}+p))^\dagger\overline{p}$.

Its proof has two steps. We first prove that $(qy)^\dagger\overline{q}$ preserves p, that is, $p(qy)^\dagger\overline{q} = p(qy)^\dagger\overline{q}p$ and $\overline{q}(qy)^\dagger\overline{q} = \overline{p}(qy)^\dagger\overline{qp}$. Then we prove the identity by applying the previous subcase. This finishes the case analysis.

We have formally shown that every Kleene algebra with tests is a pre-Conway algebra where we interpret † as *.

sublocale *kat* $\subseteq$ *pre-conway star* ⟨*proof*⟩

Thus our proof generalises Kozen's result; and Isabelle makes our theorem automatically available in Kleene algebra with tests. We have also shown that every demonic refinement algebra is a pre-Conway algebra when interpreting † as $^\infty$.

sublocale *dra-tests* $\subseteq$ *pre-conway strong-iteration* ⟨*proof*⟩

Hence our result holds in demonic refinement algebra as well; our proof generalises a previous result by Solin [21].

Finally, Rabehaja and Sanders [20] have further generalised the loop refinement theorem to a probabilistic demonic refinement algebra in which the star and the isolation axiom are absent and the left distributivity axiom is weakened to general left sub-distributivity and to a special left distributivity axiom $p\cdot(x+y) = p\cdot x+p\cdot y$ for tests p. We have adapted our proof so that it covers all three cases. We do not display this most generic result here since probabilistic variants are not the subject of this article. Our Isabelle file contains all relevant

details. Note that left distributivity does not hold in pre-Conway algebras and that the product unfold axiom and simulation axiom cannot be derived from Rabehaja and Sanders' axioms. The decision whether the Conway-style axiom set is appropriate for probabilistc reasoning depends on probabilistic semantics.

In pre-Conway algebras, the dagger axioms are too weak to distinguish between finite and potentially infinite iteration. Conway's axiom $x^{\dagger\dagger} = x^\dagger$, which we have dropped, holds of *, but not of $^\infty$, since $x^{\infty\infty} = \top$. Conway has analysed the relevance of this axiom for regular algebras and remarked that it is equivalent to $1^\dagger = 1$. In demonic refinement algebra, however, $1^\infty = \top$.

7 Relational and Predicate Transformer Semantics

This section presents the formalisation of the two most important models of Kleene algebra with tests and demonic refinement algebra: the relational model for the first and the predicate transformer model for the second. We restrict our attention to the one-sorted formalisation.

It is well known that, for each set A, the structure $(2^{A\times A}, \cup, ;, \emptyset, Id, ^*)$ forms a Kleene algebra; the *full relation Kleene algebra* over A. Here, $\cup$ corresponds to $+$, relational composition ; to $\cdot$, $\emptyset$ to 0, the identity relation Id to 1 and the reflexive transitive closure operation to *. In addition, every subalgebra of a full relation Kleene algebra forms a *relation Kleene algebra*. In the one-sorted approach to the relational model, tests are subidentities and, for each relation x, $n\ x$ is the complement of x intersected with the identity relation: $n\ x = Id \cap (-x)$. In Isabelle we have formalised the fact that binary relations form Kleene algebras with tests by an interpretation statement:

interpretation *rel-kat*: *kat*
"op ∪" "op O" "Id" "op ⊆" "op ⊂" "rtrancl" "λx. Id ∩ (−x)"
⟨*proof*⟩

The proof is fully automatic because binary relations have already been shown to form Kleene algebras [4], hence only the axioms for n need to be checked. Moreover, Isabelle's libraries for binary relations are very well developed.

The formalisation of Kleene algebra in Isabelle contains additional models, including formal languages and regular languages, sets of paths in digraphs, sets of traces and matrices. For languages there are only two tests: the empty language and the empty word language. Linking these structures with Kleene algebra with tests is therefore uninteresting. The other models have a richer test structures. Interpretation statements with respect to Kleene algebra with tests seem straightforward. This is left for future work.

The intended model of demonic refinement algebras is formed by conjunctive predicate transformers [23]. Abstractly speaking these are functions $f : B \to B$ over boolean algebras that distribute over arbitrary meets. Boolean algebras with such functions are also known as *boolean algebras with operators* [14]. We have formalised the isomorphic case where B is a field of sets and functions are

strict and additive. In this model, multiplication is function composition and 1 is the identity function; the other dioid operations are

definition $f + g \equiv \lambda\sigma.\ f\ \sigma \cup g\ \sigma$

definition $0 \equiv \lambda\sigma.\ \{\}$

definition $f \leq g \equiv \forall\sigma.\ f\ \sigma \subseteq g\ \sigma$

The iterations * and $^\infty$ correspond to least and greatest fixpoints of the function $\lambda\sigma.\ 1 + \rho \cdot \sigma$. To characterise the boolean subalgebra, we have defined the adjoint of a function f, following Jónsson and Tarski, as $adjoint\ f \equiv (\lambda\sigma.\ -\ f\ (-\sigma))$. We could then define the operation n in this model as

definition $n\ f \equiv (adjoint\ f\ \cdot\ 0) + 1$

Finally, we have created an Isabelle type for the set of strict additive functions—or boolean operators—and proved that, along with the operators defined above, these functions form a demonic refinement algebra with tests.

typedef $'a\ bool\text{-}op = \{f::'a\ set \Rightarrow 'a\ set.\ (\forall\, g\ h.\ f{\cdot}(g + h) = f{\cdot}g + f{\cdot}h \wedge 0{\cdot}f = 0)\}$

instantiation $bool\text{-}op :: (type)\ dioid\text{-}tests\text{-}zerol\ \langle proof\rangle$

instantiation $bool\text{-}op :: (dioid\text{-}tests\text{-}zerol)\ dra\text{-}tests\ \langle proof\rangle$

A dual statement for multiplicative functions or conjunctive predicate transformers could be obtained similarly. The characterisation of more general function spaces can also be achieved along these lines. We have not pursued this any further since Preoteasa [19] has already formalised an isotone predicate transformer model for demonic refinement algebra. Hence our main contribution lies in the formalisation of the function n. An integration of Preoteasa's model into the Kleene algebra hierarchy is certainly desirable for applications.

8 A Prototypical Verification Tool

We have already explained that Kleene algebras with tests provide an algebraic semantics of while programs in a partial correctness setting. It is also well known [16] that validity of Hoare triples $\vdash \{\!|p|\!\}x\{\!|q|\!\}$ can be encoded as $p \cdot x \cdot (-q) = 0$. This formula states that there are no successful executions of program x from states in p into the complement of q. In other words, if x is executed from precondition p, then its output will satisfy postcondition q upon termination. We have formalised validity of Hoare triples in Kleene algebras with tests. We have also derived all inference rules of propositional Hoare logic without the assignment rule. The derivations were fully automatic.

We now demonstrate how the relational model can be used to derive assignment rules in Isabelle and how the algebraic layer can be extended to a

simple, formally verified tool prototype for program verification and correctness. This semantic approach is in contrast to a previous axiomatic treatment of assignment [5] with schematic Kleene algebra with tests [1]. Within this tool, Kleene algebra with tests also allows us to automatically generate verification conditions which completely eliminate the control structure of programs. By our formal linkage of the relation model with abstract Kleene algebra with tests, we can of course use all abstract theorems in this particular model. Although this is not needed for verification, it is important for program transformation.

In the standard relational semantics of imperative programs, a command is a relation between states and a state is a function from variables to values. We provide a prototypical implementation of states as functions from strings to natural numbers and have defined an Isabelle type for this:

type-synonym *state* = *string* ⇒ *nat*

We have also defined assignment commands as functions from variable names, update functions and states. They return a new state in which the value of the variable has been updated. This is defined in Isabelle as follows, where *lift-fn* lifts the assignment function into the relational model.

definition *lift-fn f* ≡ *Abs-relation* {(*x*, *f x*) | *x*. *True*}

definition *assign-fn x f σ* ≡ (λ*y*. *if x* = *y then f σ else σ y*)

definition *x* := *e* ≡ *lift-fn* (*assign-fn x e*)

Subsets of the identity relation represent tests.

definition *assert P* ≡ *Abs-relation* (*Id-on P*)

This set-up allows us to derive assignment axioms, for instance,

$$P[x|e] \subseteq Q \longrightarrow \{\!|\mathit{assert}\ P|\!\}\ x := e\ \{\!|\mathit{assert}\ Q|\!\},$$

where $P[x|e]$ is the set of states in P in which the variable x has value e.

For convenience we have added a notion of loop invariant for while loops,

$$\textbf{while}\ p\ \textbf{inv}\ i\ \textbf{do}\ x = (p \cdot x)^* \cdot (-p).$$

Invariants are tests or assertions. They are used for generating verification conditions according to the rule

$$(p \leq i) \wedge (i \cdot (-b) \leq q) \wedge \{\!|i \cdot b|\!\} x \{\!|i|\!\} \longrightarrow \{\!|p|\!\}\ \textbf{while}\ b\ \textbf{inv}\ i\ \textbf{do}\ x\ \{\!|q|\!\},$$

which can be derived easily from the original Hoare rule for the while loop.

Finally, we have adapted the simple proof tactic *hoare-auto* from [5] for generating verification conditions in Isabelle. It applies Isabelle's simplifiers together with the rules of Hoare logic. This works in practice since Hoare logic provides

precisely one rule per programming construct. Resolving verification conditions then depends on Isabelle's libraries for the underlying data domains; algebra is no longer needed at this level. We verify Euclid's algorithm as an illustration.

```
lemma euclids-algorithm:
  {|{σ. σ ''x'' = x ∧ σ ''y'' = y}|}   -- states σ where ''x'' = x  and  ''y'' = y
    while {σ. σ ''y'' ≠ 0}             -- while the state has ''y'' ≠ 0
    inv {σ. gcd (σ ''x'') (σ ''y'') = gcd x y}
    do (
      ''z'' := ''y''; ''y'' := ''x'' mod ''y''; ''x'' := ''z''
    )
  {|{σ. σ ''x'' = gcd x y}|}           -- states σ where ''x'' = gcd x y
by hoare-auto (metis gcd-red-nat)
```

In this simple case, *hoare-auto* presents only $gcd\, x\, y = gcd\, y\, (x\ mod\ y)$ as a single verification condition; the other ones have been discharged by simplification. Invoking Sledgehammer discharges this condition automatically, using the fact *gcd-red-nat* which been drawn from Isabelle's library for natural numbers.

This simple prototype of a verification tool yields a general template for algebra-based program analysis systems. It can readily be adapted and extended for complex applications. Refinement and transformation tools using the predicate transformer semantics can be built along the same lines.

9 Conclusion

We have extended a reference formalisation for variants of Kleene algebras in Isabelle by two algebras that are important for program verification and correctness applications: Kleene algebras with tests and demonic refinement algebras. We provide more than 10 algebraic structures, hundreds of theorems and two important models. We have demonstrated the applicability of the implementation by two main examples, and have shown how trustworthy tools for program construction and verification can be implemented from such algebras. A coherent integration of algebraic methods into program analysis tools has thereby been achieved. The associated Isabelle theories in the Archive of Formal Proofs [3] serve as a reference for extensions and further applications.

Main applications of our formalisation lie in the development of tools for program verification and correctness. Our technique for integrating the control flow into the algebraic layer is generic. We have already extended it to arbitrary data types beyond natural numbers and verified additional algorithms. An integration of data flow into predicate transformer semantics and the extension of our tool to refinement or program transformation are topics for future work. Finally, we have applied our approach in the context of shared variable concurrency verification [2], with similar algebras, but trace-based semantics.

Acknowledgements. The authors are grateful to Jordan Milner for a preparatory implementation and to Peter Jipsen for joint work on the one-sorted test axiomatisation. They also acknowledge funding from CNPq and EPSRC.

References

[1] A. Angus and D. Kozen. Kleene algebra with tests and program schematology. Technical Report TR2001-1844, Computer Science Department, Cornell University, July 2001.

[2] A. Armstrong, V. B. F. Gomes, and G. Struth. Algebraic principles for rely-guarantee style concurrency verification tools. *CoRR*, abs/1312.1225, 2013.

[3] A. Armstrong, V. B. F. Gomes, and G. Struth. Kleene algebras with tests and demonic refinement algebras. *Archive of Formal Proofs*, 2014.

[4] A. Armstrong, G. Struth, and T. Weber. Kleene algebra. *Archive of Formal Proofs*, 2013.

[5] A. Armstrong, G. Struth, and T. Weber. Program analysis and verification based on Kleene algebra in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP*, volume 7998 of *LNCS*, pages 197–212. Springer, 2013.

[6] R.-J. Back. A method for refining atomicity in parallel algorithms. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE*, volume 366 of *LNCS*, pages 199–216. Springer, 1989.

[7] R.-J. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM TOPLAS*, 10(4):513–554, 1988.

[8] R-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[9] E. Cohen. Separation and reduction. In R. C. Backhouse and J. N. Oliveira, editors, *MPC*, volume 1837 of *LNCS*, pages 45–59. Springer, 2000.

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

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

[12] S. Foster and G. Struth. Automated analysis of regular algebra. In B. Gramlich, D. Miller, and U. Sattler, editors, *IJCAR*, volume 7364 of *LNCS*, pages 271–285. Springer, 2012.

[13] P. Höfner, G. Struth, and G. Sutcliffe. Automated verification of refinement laws. *Ann. Mathematics and Artificial Intelligence*, 55(1-2):35–62, 2009.

[14] B. Jónsson and A. Tarski. Boolean algebras with operators, part 1. *American Journal of Mathematics*, 73(4):891–939, 1951.

[15] D. Kozen. Kleene algebra with tests. *ACM TOPLAS*, 19(3):427–443, 1997.

[16] D. Kozen. On Hoare logic and Kleene algebra with tests. *ACM TOCL*, 1(1):60–76, 2000.

[17] E. G. Manes and D. B. Benson. The inverse semigroup of a sum-ordered semiring. *Semigroup Forum*, 31(1):129–152, 1985.

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

[19] V. Preoteasa. Algebra of monotonic boolean transformers. In A. S. Simão and C. Morgan, editors, *SBMF*, volume 7021 of *LNCS*, pages 140–155. Springer, 2011.

[20] T. M. Rabehaja and J. W. Sanders. Refinement algebra with explicit probabilism. In W.-N. Chin and S. Qin, editors, *TASE*, pages 63–70. IEEE Comp. Soc., 2009.

[21] K. Solin. Normal forms in total correctness for while programs and action systems. *J. Logic and Algebraic Programming*, 80(6):362–375, 2011.

[22] J. von Wright. From Kleene algebra to refinement algebra. In E. A. Boiten and B. Möller, editors, *MPC*, volume 2386 of *LNCS*, pages 233–262. Springer, 2002.

[23] J. von Wright. Towards a refinement algebra. *Science of Computer Programming*, 51(1-2):23–45, 2004.