Program Analysis and Verification based on Kleene Algebra in Isabelle/HOL

Alasdair Armstrong[1], Georg Struth[1], and Tjark Weber[2]

[1] Department of Computer Science, University of Sheffield, UK
{a.armstrong,g.struth}@dcs.shef.ac.uk
[2] Department of Information Technology, Uppsala University, Sweden
tjark.weber@it.uu.se

Abstract. Schematic Kleene algebra with tests (SKAT) supports the equational verification of flowchart scheme equivalence and captures simple while-programs with assignment statements. We formalise SKAT in Isabelle/HOL, using the quotient type package to reason equationally in this algebra. We apply this formalisation to a complex flowchart transformation proof from the literature. We extend SKAT with assertion statements and derive the inference rules of Hoare logic. We apply this extension in simple program verification examples and the derivation of additional Hoare-style rules. This shows that algebra can provide an abstract semantic layer from which different program analysis and verification tasks can be implemented in a simple lightweight way.

1 Introduction

The relevance of Kleene algebras for program development and verification has been highlighted for more than a decade. Kleene algebras provide operations for non-deterministic choice, sequential composition and finite iteration in computing systems as well as constructs for skip and abort. When a suitable boolean algebra for tests and assertions is embedded, the resulting Kleene algebras with tests (KAT) [18] can express simple while-programs and validity of Hoare triples. Extensions of Kleene algebras support Hoare-style program verification—the rules of Hoare logic except assignment can be derived—and provide notions of equivalence and refinement for program construction and transformation. Reasoning in Kleene algebras is based on first-order equational logic. It is therefore relatively simple, concise and well suited for automation [15,12,13,3]. The lightweight program semantics that Kleene algebras provide can further be specialised in various ways through their models, which include binary relations, program traces, paths in transition systems and (guarded string) languages [4].

The relevance of Kleene algebras has further been underpinned by applications, for instance, in compiler optimisation [19], program construction [5], transformation and termination [9], static analysis [11] or concurrency control [7]; but few have used theorem provers or integrated fine-grained reasoning about assignments or assertions [1,5,14]. The precise role and relevance of Kleene algebras

in a formal environment for program development and verification has not yet been explored. Our paper provides a first step in this direction.

We have implemented a comprehensive library for KAT in Isabelle/HOL [25], using explicit carrier sets for modelling the interaction between actions and tests in programs. For reasoning about assignments and assertions, we have added first-order syntax and axioms to KAT, following Angus and Kozen's approach to schematic Kleene algebras with tests (SKAT) [2]. Program syntax is defined as syntactic sugar on SKAT expressions; axiomatic algebraic reasoning about these expressions is implemented by using Isabelle's quotient package [16,17].

We have applied this simple algebraic verification environment by formalising a complex flowchart equivalence proof in SKAT due to Angus and Kozen. It is an algebraic account of a previous diagrammatic proof by Manna [21]. In their approach, flowchart schemes are translated into SKAT expressions. We have converted the manual proof in SKAT essentially one-to-one into readable Isabelle code. This significantly shortens a previous formalisation with a customised interactive SKAT-prover [1]. This compression demonstrates the power of Isabelle's proof methods and integrated theorem provers.

To illustrate the flexibility of our approach we have extended our SKAT implementation by assertions for Hoare-style partial program correctness proofs. To obtain a predicate transformer semantics for forward reasoning à la Gordon [8] we have formalised the action of programs as SKAT terms which act on a Boolean algebra of predicates or assertions via a scalar product in a Kleene module [10,20]. We have instantiated this abstract algebra of assertions to the standard powerset algebra over program states realised as maps from variables to values. We have encoded validity of Hoare triples and automatically derived the rules of Hoare logic—including assignment—in this setting. We have also provided syntactic sugar for a simple while-language with assertions (pre/post-conditions and invariants) similar to existing Hoare logics in Isabelle [24,26].

We have tested this enhanced environment by automatically verifying some simple algorithms and by automatically deriving some additional Hoare-style inference rules that would be admissible in Hoare logic. Verification is supported by a verification condition generator that reduces program verification tasks to the usual proof obligations for elementary program actions.

The complete Isabelle code for this paper can be found online.[3]

Our study points out two main benefits of using (Kleene) algebra in program development and verification. First, it provides a uniform lightweight semantic layer from which syntax for specifications and programs can be defined, domain-specific inference rules be derived and fine-grained models be explored with exceptional ease. In Isabelle this is seamlessly supported by type classes and locales and by excellent proof automation. Second, it yields a powerful proof engine for concrete analysis tasks, in particular when transforming programs or developing them from specifications. Despite this, the automation of our flowchart example remains somewhat underwhelming; such examples provide interesting benchmarks for further improving proof automation.

[3] http://www.dcs.shef.ac.uk/~alasdair/skat

2 Kleene Algebra with Tests

Kleene algebras with tests (KAT) are at the basis of our implementation. They provide simple encodings of while-programs and Hoare logic (without assignment) and support equational reasoning about program transformations and equivalence. This section gives a short introduction from a programming perspective; more details can be found in the literature [18].

A *semiring* is a structure $(S, +, \cdot, 0, 1)$ such that $(S, +, 0)$ is a commutative semigroup, $(S, \cdot, 1)$ is a monoid (not necessarily commutative), multiplication distributes over addition from the left and right, and 0 is an annihilator ($0 \cdot p = 0 = p \cdot 0$). S is *idempotent* (a *dioid*) if $p + p = p$. In that case, the reduct $(S, +, 0)$ is a semilattice, hence $p \leq q \longleftrightarrow p + q = q$ defines a partial order with least element 0. For programming, imagine that S models the actions of a system; addition is non-deterministic choice, multiplication is sequential composition, 1 is skip and 0 is abort. The next step is to add a notion of finite iteration.

A *Kleene algebra* is a dioid expanded with a star operation that satisfies the unfold axioms $1 + pp^* \leq p^*$ and $1 + p^*p \leq p^*$, and the induction axioms $r + pq \leq q \longrightarrow p^*r \leq q$ and $r + qp \leq q \longrightarrow rp^* \leq q$. This defines p^* as the simultaneous least (pre)fixpoint of the functions $\lambda q.1 + pq$ and $\lambda q.1 + qp$.

Program tests and assertions can be added by embedding a boolean algebra of tests between 0 and 1. A *Kleene algebra with tests* (KAT) is a structure $(K, B, +, \cdot, ^\star, 0, 1, ^-)$ where $(K, +, \cdot, ^*, 0, 1)$ is a Kleene algebra and $(B, +, \cdot, ^-, 0, 1)$ a Boolean subalgebra of K. The operations are overloaded with $+$ as join, $\cdot$ as meet, 0 as the minimal element and 1 the maximal element of B. Complementation $^-$ is only defined on B. We write p, q, r for arbitrary elements of K and a, b, c for tests in B. Conditionals and loops can now be expressed:

$$\mathit{IF}\ b\ \mathit{THEN}\ p\ \mathit{ELSE}\ q\ =\ bp + \overline{b}q, \qquad \mathit{WHILE}\ b\ \mathit{DO}\ p\ \mathit{WEND}\ =\ (bp)^*\overline{b}.$$

Tests play a double role as assertions to encode (the validity of) Hoare triples:

$$\{\!|b|\!\}\, p\, \{\!|c|\!\} \iff bp\overline{c} = 0.$$

Multiplying a program p by a test b at the left or right means restricting its input or output by the condition b. Thus the term $bp\overline{c}$ states that program p is restricted to precondition b in its input and to the negated postcondition c in its output. Accordingly, $bp\overline{c} = 0$ means that p cannot execute from b without establishing c. This faithfully captures the meaning of the Hoare triple $\{\!|b|\!\}\, p\, \{\!|c|\!\}$. It is well known that algebraic relatives of all rules of Hoare logic except assignment can be derived in KAT, and that binary relations under union, relational composition, the unit and the empty relation, and the reflexive transitive closure operation form a KAT. Its Boolean subalgebra of tests is formed by all elements between the empty and the diagonal relation. Binary relations yield, of course, a standard semantics for sequential programs.

A reference Isabelle implementation of Kleene algebras and their models is available in the Archive of Formal Proofs [4]. To capture the subalgebra relationship of B and K we have implemented an alternative with carrier sets and expanded this to KAT. Due to lack of space we cannot present further details.

3 Schematic KAT and Flowchart Schemes

To apply KAT in program development and verification, formal treatment of assignments and program states is required. Axioms for assignments have been added, for instance, in *schematic Kleene algebra with tests* (SKAT) [2]. This extension of KAT is targeted at modelling the transformation of flowchart schemes. A classical reference for flowchart schemes, scheme equivalence, and transformation is Manna's book *Mathematical Theory of Computation* [21]. Our formalisation of SKAT in Isabelle is discussed in this section; our formalisation of a complex flowchart equivalence proof [21,2] is presented in Section 5. We describe the conceptual development of SKAT together with its formalisation in Isabelle.

A *ranked alphabet* or signature Σ consists of a family of function symbols f, g, ... and relation symbols P, Q, ... together with an arity function mapping symbols to $\mathbb{N}$. There is always a null function symbol with arity 0. In Isabelle, we have implemented ranked alphabets as a type class. Variables are represented by natural numbers. Terms over Σ are defined as a polymorphic Isabelle datatype.

```
datatype 'a trm = App 'a "'a trm list" | Var nat
```

We omit arity checks to avoid polluting proofs with side conditions. In practice, verifications will fail if arities are violated. Variables and Σ-terms form assignment statements; together with predicate symbols they form tests in SKAT. Predicate expressions (atomic formulae) are also implemented as a datatype.

```
datatype 'a pred = Pred 'a "'a trm list"
```

Evaluation of terms, predicates and tests relies on an interpretation function. It maps function and relation symbols to functions and relations. It is used to define a notion of flowchart equivalence [2,21] with respect to all interpretations. It is also needed to formalise Hoare logic in Section 6 by interpreting Σ-expressions in semantic domains. In Isabelle, it is based on the following pair of functions.

```
record ('a, 'b) interp =
  interp-fun :: 'a ⇒ 'b list ⇒ 'b
  interp-rel :: 'a ⇒ 'b relation
```

We can now include Σ-expressions into SKAT expressions, which model flowchart schemes.

```
datatype 'a skat-expr =
    SKAssign nat "'a trm"
  | SKPlus "'a skat-expr" "'a skat-expr" (infixl "⊕" 70)
  | SKMult "'a skat-expr" "'a skat-expr" (infixl "⊙" 80)
  | SKStar "'a skat-expr"
  | SKBool "'a pred bexpr"
  | SKOne
  | SKZero
```

In this datatype, *SKAssign* is the assignment constructor; it takes a variable and a Σ-term as arguments. The other constructors capture the programming constructs of sequential composition, conditionals and while loops within KAT. The type *'a pred bexpr* represents Boolean combinations of predicates, which form the tests in SKAT. The connection between the SKAT syntax and Manna's flowchart schemes is discussed in [2], but we do not formalise it.

Having formalised the SKAT syntax we can now define a notion of flowchart equivalence by using Isabelle's quotient types. First we define the obvious congruence on SKAT terms that includes the KAT axioms and the SKAT assignment axioms

$$\begin{array}{lcll} x := s; y := t & = & y := t[x/s]; x := s & (y \notin FV(s)), \\ x := s; y := t & = & x := s; y := t[x/s] & (x \notin FV(s)), \\ x := s; x := t & = & x := t[x/s], & \\ a[x/t]; x := t & = & x := t; a. & \end{array}$$

In the following inductive definition we only show the equivalence axioms, a single Kleene algebra axiom and an assignment axiom explicitly. Additional recursive functions for free variables and substitutions support the assignment axioms.

inductive *skat-cong* :: *('a::ranked-alphabet) skat-expr* ⇒ *'a skat-expr* ⇒ *bool*
(**infix** ≈ *55*) **where**
refl [*intro*]: $p \approx p$
| *sym* [*sym*]: $p \approx q \Longrightarrow q \approx p$
| *trans* [*trans*]: $p \approx q \Longrightarrow q \approx r \Longrightarrow p \approx r$
...
| *mult-assoc*: $(p \odot q) \odot r \approx p \odot (q \odot r)$
...
| *assign1*: $[\![x \neq y;\ y \notin FV\ s]\!] \Longrightarrow$
SKAssign x s ⊙ *SKAssign y t* ≈ *SKAssign y* $(t[x/s])$ ⊙ *SKAssign x s*
...

Isabelle's quotient package [17] now allows us to formally take the quotient of SKAT expressions with respect to *skat-cong*. The SKAT axioms then become available for reasoning about SKAT expressions.

quotient-type *'a skat* = *('a::ranked-alphabet) skat-expr / skat-cong*

Using this notion of equivalence on SKAT expressions we can define additional syntactic sugar by lifting constructors to SKAT operations, for instance,

lift-definition *skat-plus* :: *('a::ranked-alphabet) skat* ⇒ *'a skat* ⇒ *'a skat*
(**infixl** + *70*) **is** *SKPlus*

We have used Isabelle's transfer tactic to provide nice programming syntax and lift definitions from the congruence. For instance,

lemma *skat-assign1*:
$[\![x \neq y;\ y \notin FV\ s]\!] \Longrightarrow (x := s \cdot y := t) = (y := t[x/s] \cdot x := s)$

An interpretation statement formally shows in Isabelle that the algebra thus constructed forms a KAT.

definition *tests* :: *('a::ranked-alphabet) skat ord* **where**
tests = (|*carrier* = *test-set*, *le* = (λ*p q*. *skat-plus p q* = *q*)|)

definition *free-kat* :: *('a::ranked-alphabet) skat test-algebra* **where**
free-kat = (|*carrier* = *UNIV*, *plus* = *skat-plus*, *mult* = *skat-mult*, *one* = *skat-one*,
zero = *skat-zero*, *star* = *skat-star*, *test-algebra.test* = *tests*|)

interpretation *skt*: *kat free-kat*

Proving this statement required some work. First, is uses our comprehensive implementation of Kleene algebra with tests (and with carrier sets) in Isabelle. Second, we needed to show that the quotient algebra constructed satisfies the KAT axioms, including those of Boolean algebra for the subalgebra of tests. A main complication comes from the fact that Boolean complementation is defined as a partial operation, that is, on tests only; thus it cannot be directly lifted from the congruence. We have defined it indirectly using Isabelle's indefinite description operator. After this interpretation proof, most statements shown for KAT are automatically available in the quotient algebra. The unfortunate exception is again the partially defined negation symbol, which is not fully captured by the interpretation statement. Here, KAT theorems need to be duplicated by hand.

When defining a quotient type, Isabelle automatically generates two coercion functions. The *abs-skat* function maps elements of type *'a skat-expr* to elements of the quotient algebra type *'a skat*, while the *rep-skat* function maps in the converse direction. Both these functions are again based on Isabelle's definite description operator, which can be unwieldy. However, as our types are inductively defined, we can as well use the following equivalent, and computationally more appealing, recursive function instead of *abs-skat*, which supports simple proofs by induction.

primrec *abs* :: *('a::ranked-alphabet) skat-expr* ⇒ *'a skat* (⌊-⌋ [*111*] *110*) **where**
abs (*SKAssign x t*) = *x* := *t*
| *abs* (*SKPlus p q*) = *abs p* + *abs q*
| *abs* (*SKMult p q*) = *abs p* · *abs q*
| *abs* (*SKBool a*) = *test-abs a*
| *abs SKOne* = **1**
| *abs SKZero* = **0**
| *abs* (*SKStar p*) = (*abs p*)⋆

Mathematically, *abs* (or ⌊−⌋) is a homomorphism. It is useful for programming various tactics.

4 Formalising a Metatheorem

We have formalised a metatheorem due to Angus and Kozen (Lemma 4.4 in [2]) that can be instantiated, for instance, to check commutativity conditions, eliminate redundant variables or rename variables in flowchart transformation proofs. We instantiate this theorem mainly to develop tactics that support proof automation in the flowchart example of the next section.

theorem *metatheorem*:
  assumes *kat-homomorphism f*
  and *kat-homomorphism g*
  and $\bigwedge a.\ a \in \mathit{atoms}\ p \Longrightarrow f\ a \cdot q = q \cdot g\ a$
  shows $f\ p \cdot q = q \cdot g\ p$

We proceed by induction on p, expanding Angus and Kozen's proof. The predicate *kat-homomorphism* in the theorem states that f and g are KAT morphisms. This notion is defined in Isabelle as a locale in the obvious way. The functions f and g map from SKAT terms into the SKAT quotient algebra, hence they have the same type as *abs*. The atoms function returns all the atomic subexpressions of a SKAT term, i.e. all the assignments and atomic tests.

Angus and Kozen have observed that if q commutes with all atomic subexpressions of p, then q commutes with p. This is a simple instantiation of the metatheorem. It can be obtained in Isabelle as follows:

lemmas *skat-comm* = *metatheorem*[*OF abs-hom abs-hom*]

This instantiates f and g using the fact that *abs* is a KAT morphism.

Lemma 4.5 in [2] states that if a variable x is not read in an expression p, then setting it to null will eliminate it from p.

lemma *eliminate-variables*:
  assumes $x \notin \mathit{reads}\ p$
  shows $\lfloor p \rfloor \cdot x := \mathit{null} = \lfloor \mathit{eliminate}\ x\ p \rfloor \cdot x := \mathit{null}$

In the statement of this lemma, *reads p* is a recursive function that returns all the variables on the right-hand side of all assignments within p, and the function *eliminate x p* removes all assignments to x in p.

We have used the metatheorem and its instances to develop tactics that check for commutativity and eliminate variables. These tactics take expressions of the quotient algebra and coerce them into the term algebra to perform these syntactic manipulations. All the machinery for these coercions, such as *abs*, is thereby hidden from the user. A simple application example is given by the following lemma.

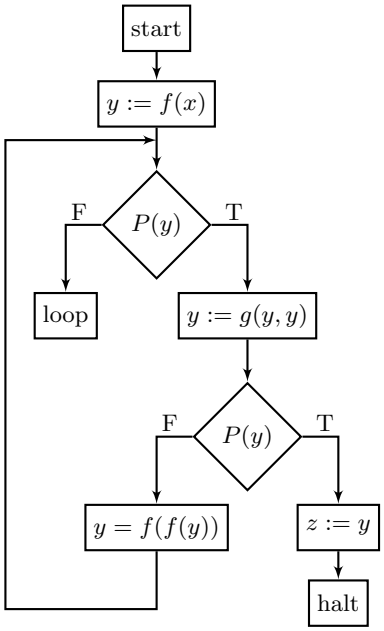
lemma *comm-ex: (1 := Var 2; 3 := Var 4) = (3 := Var 4; 1 := Var 2)*
  by *skat-comm*

5 Verification of Flowchart Equivalence

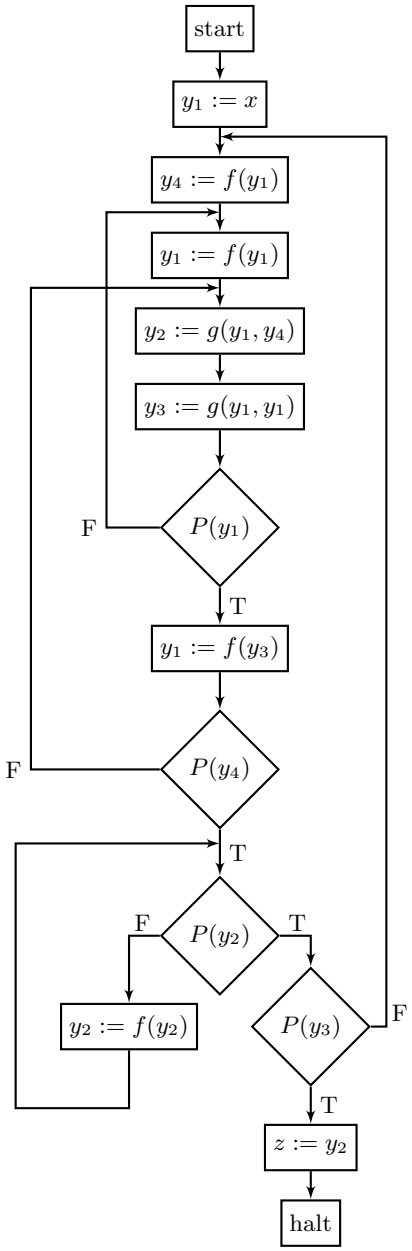
We have applied our SKAT implementation to verify a well known flowchart equivalence example in Isabelle. It is attributed by Manna to Paterson [21]. The flowcharts can be found at page 16f. in Angus and Kozen's paper [2] or pages 254 and 258 in Manna's book [21]; they are reproduced here in Figure 1. Manna's proof essentially uses diagrammatic reasoning, whereas Angus and Kozen's proof is equational. We reconstruct the algebraic proof at the same level of granularity in Isabelle. The two flowcharts, translated into SKAT by Angus and Kozen, are as follows.

```
definition scheme1 ≡ seq
 [ 1 := vx, 4 := f (Var 1), 1 := f (Var 1)
 , 2 := g (Var 1) (Var 4), 3 := g (Var 1) (Var 1)
 , loop
   [ !(P (Var 1)), 1 := f (Var 1)
   , 2 := g (Var 1) (Var 4), 3 := g (Var 1) (Var 1)
   ]
 , P (Var 1), 1 := f (Var 3)
 , loop
   [ !(P (Var 4)) + seq
     [ P (Var 4)
     , (!(P (Var 2)); 2 := f (Var 2))⋆
     , P (Var 2), ! (P (Var 3))
     , 4 := f (Var 1), 1 := f (Var 1)
     ]
   , 2 := g (Var 1) (Var 4), 3 := g (Var 1) (Var 1)
   , loop
     [ !(P (Var 1)), 1 := f (Var 1)
     , 2 := g (Var 1) (Var 4), 3 := g (Var 1) (Var 1)
     ]
   , P (Var 1), 1 := f (Var 3)
   ]
 , P (Var 4)
 , (!(P (Var 2)) · 2 := f (Var 2))⋆
 , P (Var 2), P (Var 3), 0 := Var 2, halt
 ]

definition scheme2 ≡ seq
 [ 2 := f vx, P (Var 2)
 , 2 := g (Var 2) (Var 2)
 , loop
   [ !(P (Var 2))
   , 2 := f (f (Var 2))
   , P (Var 2)
   , 2 := g (Var 2) (Var 2)
   ]
 , P (Var 2), 0 := Var 2, halt
 ]
```



Scheme S_{6E} [21, p. 258]



Scheme S_{6A} [21, p. 254]

Fig. 1. Two equivalent flowchart schemes

In the code, lists delimited by brackets indicate blocks of sequential code; loop expressions indicate the star of a block of code that follows. The *seq* function converts a block of code into a SKAT expression. The *halt* command sets all non output variables used in the scheme to *null*. To make algebraic reasoning more efficient, we follow Angus and Kozen in introducing definitions that abbreviate atomic commands, in particular assignments, and tests. The flowchart equivalence problem can then be expressed more succinctly and abstractly in KAT, as all assignment statements, which are dealt with by SKAT, have been abstracted.

$$
\begin{aligned}
&\mathit{seq}\ [x1,p41,p11,q214,q311,\mathit{loop}\ [!a1,p11,q214,q311],a1,p13\\
&\quad ,\mathit{loop}\ [!a4\ +\ \mathit{seq}\ [a4,(!a2{\cdot}p22)^{\star},a2,!a3,p41,p11]\\
&\quad ,q214,q311,\mathit{loop}\ [!a1,p11,q214,q311],a1,p13]\\
&\quad ,a4,(!a2{\cdot}p22)^{\star},a2,a3,z2,\mathit{halt}]\\
=\\
&\mathit{seq}\ [s2,a2,q222,(\mathit{seq}\ [!a2,r22,a2,q222])^{\star},a2,z2,\mathit{halt}]
\end{aligned}
$$

The proof that rewrites these KAT expressions, however, needs to descend to SKAT in order to derive commutativity conditions between expressions that depend on variables and Σ-terms. These conditions are then lifted to KAT. The condition expressed in Lemma *comm-ex* from Section 4, for instance, reduces to the KAT identity $pq = qp$ when abbreviating $1 := Var\ 2$ as p, and $3 := Var\ 4$ as q. In our proof we infer these commutativity conditions in a lazy fashion. This follows Angus and Kozen's proof essentially line by line.

We heavily depend on our underlying KAT library, which contains about 100 lemmas for dealing with the Kleene star and combined reasoning about the interaction between actions and tests. Typical properties are $(p + q)^* = p^*(q \cdot p^*)^*$, $(pq)^*p = p(qp)^*$ or $bp = pc \Longleftrightarrow bp!c = !bpc$. We have also refined the tactics mentioned in the previous section to be able to efficiently manipulate the large SKAT expressions that occur in the proof. Most of these implement commutations in lists of expressions modulo commutativity conditions on atomic expressions which are inferred from SKAT terms on the fly.

The size of our proof as a LaTeX document is about 12 pages, twice as many as in Angus and Kozen's manual proof, but this is essentially due to aligning their horizontal equational proofs in a vertical way. A previous proof in a special-purpose SKAT prover required 41 pages [1]. This impressively demonstrates the power of Isabelle's proof automation. Previous experience in theorem proving with algebra shows that the level of proof automation in algebra is often very high [15,13,12]. In this regard, our present proof experience is slightly underwhelming, as custom tactics and low-level proof techniques were needed for our step-by-step proof reconstruction. A higher degree of automation seems difficult to achieve, and a complete automation of the scheme equivalence proof currently out of reach. The main reason is that the flowchart terms in KAT are much longer, and combinatorially more complex, than those in typical textbook proofs. Decision procedures for variants of Kleene algebras, which currently only exist in Coq [6], might overcome this difficulty.

6 Hoare Logic

It is well known that Hoare logic—except the assignment rule—can be encoded in KAT as well as in other variants of Kleene algebra such as modal Kleene algebras [22] and Kleene modules [10]. The latter are algebraic relatives of propositional dynamic logic. A combination of these algebras with the assignment rule and their application in program verification has so far not been attempted.

We have implemented a novel approach in which SKAT and Kleene modules are combined. This allows us to separate tests conceptually from the pre- and post-conditions of programs.

A *Kleene module* [20] is a structure $(K, L, :)$ where K is a Kleene algebra, L a join-semilattice with least element $\bot$ and $:$ a mapping of type $L \times K \to L$ where

$$\begin{aligned} P : (p + q) &= P : p \sqcup P : q, & (P \sqcup Q) : p &= P : p \sqcup Q : p, \\ P : (p \cdot q) &= P : p : q, & (P \sqcup Q) : p \leq Q &\longrightarrow P : p^* \leq Q, \\ P : 0 &= \bot, & P : 1 &= P. \end{aligned}$$

In this context, L models the space of states, propositions or assertions of a program, K its actions, and the scalar product maps a proposition and an action to a new proposition. We henceforth assume that L is a Boolean algebra with maximal element $\top$ and use a KAT instead of a Kleene algebra as the first component of the module. The interaction between assertions, as modelled by the Boolean algebra L, and tests, as modelled by the Boolean algebra B, is captured by the new axiom

$$P : a = P \sqcap (\top : a).$$

The scalar product $\top : a$ coerces the test a into an assertion ($\top$ does not restrict it); the scalar product $P : a$ is therefore equal to a conjunction between the assertion P and the test a.

We have used Isabelle's locales to implement modules over KAT. Hoare triples can then be defined as usual.

definition *hoare-triple* :: $'b \Rightarrow 'a \Rightarrow 'b \Rightarrow$ *bool* ($\{\!|$-$|\!\}$ - $\{\!|$-$|\!\}$ [*54*,*54*,*54*] *53*) **where**
$\{\!|P|\!\}\ p\ \{\!|Q|\!\} \equiv P :: p \sqsubseteq_L Q$

As : is a reserved symbol in Isabelle, we use :: for the scalar product. The index L refers to the Boolean algebra of assertions and the order $\sqsubseteq_L$ is the order on this Boolean algebra. As is well known, the Hoare rules excluding assignment can now be derived as theorems in these modules more or less automatically. Applying the resulting Hoare-style calculus—which is purely equational—for program verification requires us to provide more fine-grained syntax for assertions and refinement statements and adding some form of assignment axiom.

We obtain this first-order syntax once more by specialising KAT to SKAT, and by interpreting the SKAT expressions in the Boolean algebra of propositions or states. As usual, program states are represented as functions from variables

to values. Assertions correspond to sets of states. Hence the Boolean algebra L is instantiated as a powerset algebra over states. Similar implementations are already available in theorem provers such as Isabelle, HOL and Coq [24,26,8,23], but they have not been implemented as simple instantiations of more general algebraic structures. Assignment statements are translated in Gordon style [8] into forward predicate transformers which map assertions (preconditions) to assertions (postconditions).

This is, of course, compatible with the module-based approach. To implement the scalar product of our KAT module, we begin by writing an evaluation function which, given an interpretation and a SKAT expression, returns the forward predicate transformer for that expression.

fun *eval-skat-expr* ::
  (*'a*::*ranked-alphabet*, *'b*) *interp* ⇒ *'a skat-expr* ⇒ *'b mems* ⇒ *'b mems*
  where
  eval-skat-expr D (*SKAssign x t*) *Δ* = *assigns D x t Δ*
| *eval-skat-expr D* (*SKBool a*) *Δ* = *filter-set* (*eval-bexpr D a*) *Δ*
| *eval-skat-expr D* (*p* ⊙ *q*) *Δ* = *eval-skat-expr D q* (*eval-skat-expr D p Δ*)
| *eval-skat-expr D* (*p* ⊕ *q*) *Δ* = *eval-skat-expr D p Δ* ∪ *eval-skat-expr D q Δ*
| *eval-skat-expr D* (*SKStar p*) *Δ* = (⋃ *n*. *iter n* (*eval-skat-expr D p*) *Δ*)
| *eval-skat-expr D SKOne Δ* = *Δ*
| *eval-skat-expr D SKZero Δ* = {}

We can now prove that if two SKAT expressions are equivalent according to the congruence defined in Section 3, then they represent the same predicate transformer. The proof is by induction. This property allows us to lift the *eval-skat-expr* function to the quotient algebra.

theorem *skat-cong-eval*:
  skat-cong p q ⟹ ∀ *Δ*. *eval-skat-expr D p Δ* = *eval-skat-expr D q Δ*

lift-definition *eval* ::
  (*'a*::*ranked-alphabet*, *'b*) *interp* ⇒ *'a skat* ⇒ *'b mems* ⇒ *'b mems*
  is *eval-skat-expr*

Using this lifting, we can reason algebraically in instances of SKAT that have been generated by the evaluation function. This enables us to derive an assignment rule for forward reasoning in Hoare logic from the SKAT axioms.

lemma *hoare-assignment*: $P[x/t] \subseteq Q \Longrightarrow \{\!|P|\!\}\ x := t\ \{\!|Q|\!\}$

We could equally derive a forward assignment rule $P\{\!|x := s|\!\}P[x/s]$, but this seems less useful in practice.

To facilitate automated reasoning we have added a notion of loop invariant as syntactic sugar for while loops. Invariants are assertions used by the tactic that generates verification conditions.

$$WHILE\ b\ INVARIANT\ i\ DO\ p\ WEND = (bp)^*\overline{b}.$$

We have also derived a refined while rule which uses the loop invariant.

lemma *hoare-while-inv*:
assumes *b-test*: $b \in carrier\ tests$
and *Pi*: $P \subseteq i$ **and** *iQ*: $i \cap (UNIV :: !b) \subseteq Q$
and *inv-loop*: $\{\!|i \cap (UNIV :: b)|\!\}\ p\ \{\!|i|\!\}$
shows $\{\!|P|\!\}\ WHILE\ b\ INVARIANT\ i\ DO\ p\ WEND\ \{\!|Q|\!\}$

This particular rule has been instantiated to the powerset algebra of states, but it could as well have been defined abstractly.

Isabelle already provides a package for Hoare logic [26]. Since there is one Hoare rule per programming construct, it uses a tactic to blast away the control structure of programs. We have implemented a similar tactic for our SKAT-based implementation, called *hoare-auto*.

7 Verification Examples

We have applied our variant of Hoare logic to prove the partial correctness of some simple algorithms. Instead of applying each rule manually we use our tactic *hoare-auto* to make their verification with sledgehammer almost fully automatic. More complex examples would certainly require more user interaction or more sophisticated tactics to discharge the generated proof obligations.

lemma *euclids-algorithm*:
$\{\!|\{mem.\ mem\ 0 = x \wedge mem\ 1 = y\}|\!\}$
$WHILE\ !(pred\ (EQ\ (Var\ 1)\ (NAT\ 0)))$
$INVARIANT\ \{mem.\ gcd\ (mem\ 0)\ (mem\ 1) = gcd\ x\ y\}$
DO
$2 := Var\ 1;$
$1 := MOD\ (Var\ 0)\ (Var\ 1);$
$0 := Var\ 2$
$WEND$
$\{\!|\{mem.\ mem\ 0 = gcd\ x\ y\}|\!\}$
by *hoare-auto* (*metis gcd-red-nat*)

lemma *factorial*:
$\{\!|\{mem.\ mem\ 0 = x\}|\!\}$
$1 := NAT\ 1;$
$(WHILE\ !(pred\ (EQ\ (Var\ 0)\ (NAT\ 0)))$
$INVARIANT\ \{mem.\ fact\ x = mem\ 1 * fact\ (mem\ 0)\}$
DO
$1 := MULT\ (Var\ 1)\ (Var\ 0);\ 0 := MINUS\ (Var\ 0)\ (NAT\ 1)$
$WEND)$
$\{\!|\{mem.\ mem\ 1 = fact\ x\}|\!\}$
by *hoare-auto* (*metis fact-reduce-nat*)

Finally, our algebraic approach is expressive enough for deriving further program transformation or refinement rules, which would only be admissible in

Hoare logic. As an example we provide proofs of two simple Hoare-style inference rules. Program refinement or transformation rules could be derived in a similar way.

lemma *derived-rule1*:
assumes $\{P1,P2,Q1,Q2\} \subseteq$ *carrier A* **and** $p \in$ *carrier K*
and $\{\!|P1|\!\}\ p\ \{\!|Q1|\!\}$ **and** $\{\!|P2|\!\}\ p\ \{\!|Q2|\!\}$
shows $\{\!|P1 \sqcap P2|\!\}\ p\ \{\!|Q1 \sqcap Q2|\!\}$
using *assms*
apply (*auto simp add*: *hoare-triple-def assms*, *subst A.bin-glb-var*)
by (*metis A.absorb1 A.bin-lub-var A.meet-closed A.meet-comm mod-closed mod-join*)+

lemma *derived-rule2*:
assumes $\{P,Q,R\} \subseteq$ *carrier A* **and** $p \in$ *carrier K* **and** $P :: p = (\top :: p) \sqcap P$
and $\{\!|Q|\!\}\ p\ \{\!|R|\!\}$
shows $\{\!|P \sqcap Q|\!\}\ p\ \{\!|P \sqcap R|\!\}$
by (*insert assms*) (*smt derived-rule1 derived-rule2 insert-subset*)

Only the derivation of the first rule is not fully automatic. The side condition $P :: p = (\top :: p) \sqcap P$ expresses the fact that if assertion P holds before execution of program p, which is the left-hand side of the equation, then it also holds after p is executed. The expression $\top :: p$ represents the assertion that holds after p is executed without any input restriction.

These examples demonstrate the benefits of the algebraic approach in defining syntax, deriving domain-specific inference rules and linking with more refined models and semantics of programs with exceptional ease. While, in the context of verification, these tasks belong more or less to the metalevel, they are part of actual correctness proofs in program construction, transformation or refinement. We believe that this will be the most important domain for future applications.

8 Conclusion

We have implemented schematic Kleene algebra with tests in Isabelle/HOL, and used it to formalise a complex flowchart equivalence proof by Angus and Kozen. Our proof is significantly shorter than a previous formalisation in a custom theorem prover for Kleene algebra with tests. Our proof follows Angus and Kozen's manual proof almost exactly and translates it essentially line-by-line into Isabelle, despite some weaknesses in proof automation which sometimes forced us to reason at quite a low level. We have also extended SKAT to support the verification of simple algorithms in a Hoare-logic style. Our approach provides a seamless bridge between our abstract algebraic structures and concrete programs. We have tested our approach on a few simple verification examples. Beyond that, we have derived additional Hoare-style rules and tactics for proof automation abstractly in the algebraic setting. These can be instantiated to different semantics and application domains. In the context of verification the main aplications of algebra seem to be at this meta-level. The situation is different

when developing programs from specifications or proving program equivalence, as the flowchart scheme tranformation shows. In this context, algebra can play an essential role in concrete proofs.

Our Isabelle implementation sheds some light on the role of Kleene algebra for program development and verification. Given libraries for the basic algebraic structures and important models, we could prototype tools for flowchart equivalence and Hoare-style verification proofs with little effort and great flexibility. Moving, e.g., from a partial to a total correctness environment would require minor changes to the algebra (and of course a termination checker). We doubt that a bottom-up semantic approach would be equally simple and flexible. Isabelle turned out to be very well suited for our study. Hierarchies of algebras and their models could be implemented using type classes and locales, verification tasks were well supported by tactics and automated theorem provers. The automation of textbook algebraic proofs is usually very high. Algebraic proof obligations generated from verification conditions, however, turned out to be more complex, cf. our flowchart example. Such proofs can provide valuable benchmarks for developers of theorem provers, decision procedures and domain specific solvers.

References

1. K. Aboul-Hosn and D. Kozen. KAT-ML: an interactive theorem prover for Kleene algebra with tests. *Journal of Applied Non-Classical Logics*, 16(1–2):9–34, 2006.
2. A. Angus and D. Kozen. Kleene algebra with tests and program schematology. Technical Report TR2001-1844, Computer Science Department, Cornell University, July 2001.
3. A. Armstrong and G. Struth. Automated reasoning in higher-order regular algebra. In W. Kahl and T. G. Griffin, editors, *RAMiCS*, volume 7560 of *LNCS*, pages 66–81. Springer, 2012.
4. A. Armstrong, G. Struth, and T. Weber. Kleene algebra. *Archive of Formal Proofs*, 2013. http://afp.sf.net/entries/Kleene_Algebra.shtml, Formal proof development.
5. R. Berghammer and G. Struth. On automated program construction and verification. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *MPC 2010*, volume 6120 of *LNCS*, pages 22–41. Springer, 2010.
6. T. Braibant and D. Pous. Deciding Kleene algebras in Coq. *Logical Methods in Computer Science*, 8(1), 2012.
7. E. Cohen. Separation and reduction. In R. C. Backhouse and J. Nuno Oliveira, editors, *MPC 2000*, volume 1837 of *LNCS*, pages 45–59. Springer, 2000.
8. H. Collavizza and M. Gordon. Forward with Hoare. In A. W. Roscoe, C. B. Jones, and K. R. Wood, editors, *Reflections on the Work of C. A. R. Hoare*, pages 101–121. Springer, 2010.
9. J. Desharnais, B. Möller, and G. Struth. Algebraic notions of termination. *Logical Methods in Computer Science*, 7(1), 2011.
10. T. Ehm, B. Möller, and G. Struth. Kleene modules. In R. Berghammer, B. Möller, and G. Struth, editors, *RelMiCS*, volume 3051 of *LNCS*, pages 112–124. Springer, 2003.
11. T. Fernandes and J. Desharnais. Describing data flow analysis techniques with Kleene algebra. *Sci. Comput. Program.*, 65(2):173–194, 2007.

12. S. Foster and G. Struth. Automated analysis of regular algebra. In B. Gramlich, D. Miller, and U. Sattler, editors, *IJCAR 2012*, volume 7364 of *LNCS*, pages 271–285. Springer, 2012.
13. W. Guttmann, G. Struth, and T. Weber. Automating algebraic methods in Isabelle. In S. Quin and Z. Qiu, editors, *ICFEM 2011*, volume 6991 of *LNCS*, pages 617–632. Springer, 2011.
14. W. Guttmann, G. Struth, and T. Weber. A repository for Tarski-Kleene algebras. In P. Höfner, A. McIver, and G. Struth, editors, *ATE 2011*, volume 760 of *CEUR Workshop Proceedings*, pages 30–39. CEUR-WS.org, 2011.
15. P. Höfner and G. Struth. Automated reasoning in Kleene algebra. In F. Pfenning, editor, *CADE*, volume 4603 of *LNCS*, pages 279–294. Springer, 2007.
16. B. Huffman and O. Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In *Isabelle Users Workshop*, 2012.
17. C. Kaliszyk and C. Urban. Quotients revisited for Isabelle/HOL. In W. C. Chu, W. E. Wong, M. J. Palakal, and C-C. Hung, editors, *SAC*, pages 1639–1644. ACM, 2011.
18. D. Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, 1997.
19. D. Kozen and M.-C. Patron. Certification of compiler optimizations using Kleene algebra with tests. In J. D. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. Moniz Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic*, volume 1861 of *LNCS*, pages 568–582. Springer, 2000.
20. H. Leiß. Kleene modules and linear languages. *J. Log. Algebr. Program.*, 66(2):185–194, 2006.
21. Z. Manna. *Mathematical theory of computation*. McGraw-Hill, 1974.
22. B. Möller and G. Struth. Algebras of modal operators and partial correctness. *Theor. Comput. Sci.*, 351(2):221–239, 2006.
23. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In J. Hook and P. Thiemann, editors, *ICFP*, pages 229–240. ACM, 2008.
24. T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics. *Formal Asp. Comput.*, 10(2):171–186, 1998.
25. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
26. N. Schirmer. *Verification of sequential imperative programs in Isabelle-HOL*. PhD thesis, Technische Universität München, 2006.