Kleene Algebra with Tests

DEXTER KOZEN
Cornell University

We introduce Kleene algebra with tests, an equational system for manipulating programs. We give a purely equational proof, using Kleene algebra with tests and commutativity conditions, of the following classical result: every **while** program can be simulated by a **while** program with at most one **while** loop. The proof illustrates the use of Kleene algebra with tests and commutativity conditions in program equivalence proofs.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*structured programming*; D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; D.3.3 [**Software Engineering**]: Language Constructs and Features—*control structures*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions*; *invariants*; *logics of programs*; *mechanical verification*; *pre- and postconditions*; *specification techniques*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*algebraic approaches to semantics*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*control primitives*; I.1.1 [**Algebraic Manipulation**]: Expressions and Their Representations—*simplification of expressions*; I.1.3 [**Algebraic Manipulation**]: Languages and Systems—*special-purpose algebraic systems*; I.2.2 [**Algebraic Manipulation**]: Automatic Programming—*program modification*; *program synthesis*; *program transformation*; *program verification*

General Terms: Design, Languages, Theory, Verification

Additional Key Words and Phrases: Dynamic logic, Kleene algebra, specification

1. INTRODUCTION

Kleene algebras are algebraic structures with operators $+$, $\cdot$, *, 0, and 1 satisfying certain axioms. They arise in various guises in many contexts: relational algebra [Ng 1984; Tarski 1941], semantics and logics of programs [Kozen 1981; Pratt 1988], automata and formal language theory [Kuich 1987; Kuich and Salomaa 1986], and the design and analysis of algorithms [Aho et al. 1975; Iwano and Steiglitz 1990; Kozen 1991]. Many authors have contributed to the development of Kleene algebra [Anderaa 1965; Archangelsky 1992; Backhouse 1975; Bloom and Ésik 1993; Boffa 1990; Cohen 1994a; Conway 1971; Gorshkov 1989; Kleene 1956; Kozen 1981; Kozen 1990; Kozen 1994; Krob 1991; Kuich and Salomaa 1986; Pratt 1990; Redko

The support of the National Science Foundation under grant CCR-9317320 is gratefully acknowledged.
Author's address: Computer Science Department, Cornell University, Ithaca, NY 14853-7501; email: kozen@cs.cornell.edu.

1964; Sakarovitch 1987; Salomaa 1966.]

In semantics and logics of programs, Kleene algebra forms an essential component of Propositional Dynamic Logic (PDL) [Fischer and Ladner 1979], in which it is mixed with Boolean algebra and modal logic to give a theoretically appealing and practical system for reasoning about computation at the propositional level.

Syntactically, PDL is a two-sorted logic consisting of *programs* and *propositions*, defined by mutual induction. A test $\varphi?$ can be formed from any proposition φ; intuitively, $\varphi?$ acts as a guard that succeeds with no side effects in states satisfying φ and fails or aborts in states not satisfying φ. Semantically, programs are modeled as binary relations on a set of states, and $\varphi?$ is interpreted as the subset of the identity relation consisting of all pairs (s, s) such that φ is true in state s.

From a practical point of view, many simple program manipulations, such as loop unwinding and basic safety analysis, do not require the full power of PDL, but can be carried out in a purely equational subsystem using the axioms of Kleene algebra. However, tests are an essential ingredient, since they are needed to model conventional programming constructs such as conditionals and **while** loops. We define in Section 2 a variant of Kleene algebra, called *Kleene algebra with tests*, for reasoning equationally with these constructs.

Cohen has studied Kleene algebra in the presence of extra Boolean and commutativity conditions. He has given several practical examples of the use of Kleene algebra in program verification, such as lazy caching [Cohen 1994b] and concurrency control [Cohen 1994c]. He has also shown that Kleene algebra with extra conditions of the form $p = 0$ remains decidable [Cohen 1994a]. It can be shown using a result of Berstel [1979] (see also Gibbons and Rytter [1986] and Kozen [1996]) that *-continuous Kleene algebra in the presence of extra commutativity conditions of the form $pq = qp$, even for atomic p and q, is undecidable.

In Section 3 we give a complete equational proof of a classical folk theorem [Harel 1980; Mirkowska 1972] which states that every **while** program can be simulated by another **while** program with at most one **while** loop. The approach we take is that of Mirkowska [1972], who gives a set of local transformations that allow every **while** program to be transformed systematically to one with at most one **while** loop. For each such transformation, we give a purely equational proof of correctness. This result illustrates the use of Kleene algebra with tests and commutativity conditions in program equivalence proofs.

In Section 4.1, we observe that the universal Horn theory of the *-continuous Kleene algebras is not recursively enumerable, therefore not finitely axiomatizable. This follows from a construction of Berstel [1979] (see also Gibbons and Rytter [1986]) and Lemma 7.1 of Kozen [1991]. This resolves an open question of Kozen [1994].

2. KLEENE ALGEBRA

As defined in Kozen [1994], a *Kleene algebra* is an algebraic structure

$$(K,\ +,\ \cdot,\ ^*,\ 0,\ 1)$$

satisfying (1)–(15) below. As usual, we omit the operator $\cdot$ from expressions, writing pq for $p \cdot q$. The order of precedence of the operators is $^* > \cdot > +$; thus $p + qr^*$

should be parsed as $p + (q(r^*))$. The unary operator $^+$ is defined by $q^+ = qq^*$.

$$\begin{aligned}
p + (q + r) &= (p + q) + r && (1)\\
p + q &= q + p && (2)\\
p + 0 &= p && (3)\\
p + p &= p && (4)\\
p(qr) &= (pq)r && (5)\\
1p &= p && (6)\\
p1 &= p && (7)\\
p(q + r) &= pq + pr && (8)\\
(p + q)r &= pr + qr && (9)\\
0p &= 0 && (10)\\
p0 &= 0 && (11)\\
1 + pp^* &= p^* && (12)\\
1 + p^*p &= p^* && (13)\\
q + pr \le r &\rightarrow p^*q \le r && (14)\\
q + rp \le r &\rightarrow qp^* \le r && (15)
\end{aligned}$$

where $\le$ refers to the natural partial order on K:

$$p \le q \leftrightarrow p + q = q.$$

Instead of (14) and (15), we might take the equivalent axioms

$$\begin{aligned}
pr \le r &\rightarrow p^*r \le r && (16)\\
rp \le r &\rightarrow rp^* \le r. && (17)
\end{aligned}$$

Axioms (1)–(11) say that the structure is an idempotent semiring under $+$, $\cdot$, 0, and 1, and the remaining axioms (12)–(17) say essentially that * behaves like the Kleene star operator of formal language theory or the reflexive transitive closure operator of relational algebra. See Kozen [1994] for an introduction.

A Kleene algebra is said to be *-*continuous* if it satisfies the infinitary condition

$$pq^*r = \sup_{n\ge 0} pq^n r \qquad (18)$$

where

$$\begin{aligned}
q^0 &= 1\\
q^{n+1} &= qq^n
\end{aligned}$$

and where the supremum is with respect to the natural order $\le$. We can think of (18) as a conjunction of the infinitely many axioms $pq^n r \le pq^*r$, $n \ge 0$, and the infinitary Horn formula

$$\bigwedge_{n\ge 0} pq^n r \le s \rightarrow pq^*r \le s.$$

In the presence of the other axioms, the *-continuity condition (18) implies (14)–(17) and is strictly stronger in the sense that there exist Kleene algebras that are not *-continuous [Kozen 1990].

All true identities between regular expressions, interpreted as regular sets of strings, are derivable from the axioms of Kleene algebra [Kozen 1994]. In the author's experience, two of the most useful such identities for simplifying expressions are

$$\begin{aligned} p(qp)^* &= (pq)^*p & (19)\\ p^*(qp^*)^* &= (p+q)^*, & (20)\end{aligned}$$

which we call the *sliding* and the *denesting rules*, respectively. For example, to derive the identity $(p^*q)^* = 1 + (p+q)^*q$, we might reason equationally as follows:

$$\begin{aligned} (p^*q)^* &= 1 + p^*q(p^*q)^* && \text{by (12)}\\ &= 1 + p^*(qp^*)^*q && \text{by (19)}\\ &= 1 + (p+q)^*q && \text{by (20).}\end{aligned}$$

2.1 Kleene Algebra with Tests

To accommodate tests, we introduce the following variant of Kleene algebra. A *Kleene algebra with tests* is a two-sorted algebra

$$(K,\ B,\ +,\ \cdot,\ ^*,\ 0,\ 1,\ ^-)$$

where $B \subseteq K$, and $^-$ is a unary operator defined only on B such that

$$(K,\ +,\ \cdot,\ ^*,\ 0,\ 1)$$

is a Kleene algebra and

$$(B,\ +,\ \cdot,\ ^-,\ 0,\ 1)$$

is a Boolean algebra. The elements of B are called *tests*. We reserve the letters p, q, r, s, t, u, v for arbitrary elements of K and a, b, c, d, e for tests. In PDL, a test would be written $b?$, but since we are using different symbols for tests we can omit the ?.

The sequential composition operator $\cdot$ acts as conjunction when applied to tests, and the choice operator $+$ acts as disjunction. Intuitively, a test bc succeeds iff both b and c succeed, and $b + c$ succeeds iff either b or c succeeds.

It follows immediately from the definition that $b \le 1$ for all $b \in B$. It is tempting to define tests in an arbitrary Kleene algebra to be the set $\{p \in K \mid p \le 1\}$. This is the approach taken by Cohen [1994a]. This approach is more restrictive than ours, and we feel that it is less desirable for two important reasons:

(1) Although it makes sense in algebras of binary relations [Ng 1984; Tarski 1941], it does not work in all Kleene algebras. For example, consider the (min,+) Kleene algebra of the theory of algorithms (see Kozen [1991]), consisting of the set $\{x \in \mathbf{R} \mid x \ge 0\} \cup \{\infty\}$ with operations min, $+$, $x \mapsto 0$, ∞, and 0 for the Kleene algebra operations $+$, $\cdot$, *, 0, and 1, respectively. In this algebra, $p \le 1$ for all p, but the idempotence law $pp = p$ fails, so the set $\{p \in K \mid p \le 1\}$ does not form a Boolean algebra. In our approach, every Kleene algebra extends

trivially to a Kleene algebra with tests by taking the two-element Boolean algebra $\{0, 1\}$. Of course, there are more interesting models as well.

(2) Even in algebras of binary relations, it forces us to consider all elements $p \le 1$ as tests, including conditions that in practice would not normally be considered testable. For example, there may be programs p whose input/output relations have no side effects—i.e., $p \le 1$—but the associated test succeeds iff p halts, which in general is undecidable. We intend tests to be viewed as simple predicates that are easily recognizable as such and that are immediately decidable in a given state (and whose complements are therefore also immediately decidable). Having an explicit Boolean subalgebra allows this.

2.2 While Programs

For the results of Section 3, we work with a Pascal-like programming language with sequential composition $p\,;q$, a conditional test **if** b **then** p **else** q, and a looping construct **while** b **do** p. Programs built inductively from atomic programs and tests using these constructs are called *while programs*. We take the sequential composition operator to be of lower precedence than the conditional test or **while** loop, parenthesizing with **begin...end** where necessary; thus

while b **do** $p\,;q$

should be parsed as

begin while b **do** p **end** $;q$

and not

while b **do begin** $p\,;q$ **end**.

We occasionally omit the **else** clause of a conditional test. This can be considered an abbreviation for a conditional test with the dummy **else** clause 1 (true).

These constructs are modeled in Kleene algebra with tests as follows:

$$\begin{aligned} p\,;q &= pq \\ \textbf{if } b \textbf{ then } p \textbf{ else } q &= bp + \overline{b}q \\ \textbf{if } b \textbf{ then } p &= bp + \overline{b} \\ \textbf{while } b \textbf{ do } p &= (bp)^*\overline{b}. \end{aligned}$$

See Kozen and Tiuryn [1990] for a discussion of the relation-theoretic semantics of **while** programs and tests and a semantic justification of these definitions.

2.3 Commutativity Conditions

We will also be reasoning in the presence of *commutativity conditions* of the form $bp = pb$, where p is an arbitrary element of the Kleene algebra and where b is a test. The practical significance of these conditions will become apparent in Section 3. Intuitively, the execution of program p does not affect the value of b. It stands to reason that if p does not affect b, then neither should it affect $\overline{b}$. This is indeed the case:

LEMMA 2.3.1. *In any Kleene algebra with tests, the following are equivalent:*

(1) $pb = bp$
(2) $p\overline{b} = \overline{b}p$
(3) $bp\overline{b} + \overline{b}pb = 0$.

PROOF. By symmetry, it suffices to show the equivalence of (1) and (3). Assuming (1), we have

$$bp\overline{b} + \overline{b}pb \;=\; pb\overline{b} + \overline{b}bp \;=\; p0 + 0p \;=\; 0.$$

Conversely, assuming (3), we have $bp\overline{b} = \overline{b}pb = 0$. Then

$$\begin{aligned} pb &= (b+\overline{b})pb = bpb + \overline{b}pb = bpb + 0 = bpb, \\ bp &= bp(b+\overline{b}) = bpb + bp\overline{b} = bpb + 0 = bpb; \end{aligned}$$

therefore $pb = bp$. □

Of course, any pair of tests commute, i.e., $bc = cb$; this is an axiom of Boolean algebra.

We conclude this section with a pair of useful results that are fairly evident from an intuitive point of view, but nevertheless require formal justification.

LEMMA 2.3.2. *In any Kleene algebra with tests, if $bq = qb$, then*

$$bq^* \;=\; (bq)^*b \;=\; q^*b \;=\; b(qb)^*.$$

Note that it is *not* the case that $bq^* = (bq)^*$: when $b = 0$, the left-hand side is 0 and the right-hand side is 1.

PROOF. We prove the three inequalities

$$bq^* \;\le\; (bq)^*b \;\le\; q^*b \;\le\; bq^*;$$

the equivalence of $b(qb)^*$ with these expressions follows from (19). For the first inequality, it suffices by axiom (15) to show that $b + (bq)^*bq \le (bq)^*b$. By Boolean algebra and the commutativity assumption, we have $bq = bbq = bqb$; therefore

$$b + (bq)^*bq \;=\; b + (bq)^*bqb \;=\; (1 + (bq)^*bq)b \;=\; (bq)^*b.$$

The second inequality follows from $b \le 1$ and the monotonicity of the Kleene algebra operators.

For the last inequality, it suffices by (14) to show $b + qbq^* \le bq^*$:

$$b + qbq^* \;=\; b + bqq^* \;=\; b(1 + qq^*) \;=\; bq^*.$$

Note that in this last argument, we did not use the fact that b was a test. □

THEOREM 2.3.3. *In any Kleene algebra, if p is generated by a set of elements all of which commute with q, then p commutes with q.*

PROOF. Let p be an expression in the language of Kleene algebra, and assume that all atomic subexpressions of p commute with q. The proof is by induction on the structure of p. The basis and all inductive cases except for subexpressions of the form r^* are straightforward. For the inductive case $p = r^*$, we have by the induction hypothesis that $qr = rq$, and we need to argue that $qr^* = r^*q$. The inequality in one direction is given by the argument in the last paragraph in the proof of Lemma 2.3.2, which uses (14), and in the other direction by a symmetric argument using (15). □

3. A FOLK THEOREM

In this section we give a purely equational proof, using Kleene algebra with tests and commutativity conditions, of a classical result: *every* **while** *program can be simulated by a* **while** *program with at most one* **while** *loop, provided extra Boolean variables are allowed.* This theorem is the subject of a treatise on folk theorems by Harel [1980], who notes that it is commonly but erroneously attributed to Böhm and Jacopini [1966] and who argues with some justification that it was known to Kleene. The version as stated here is originally due to Mirkowska [1972], who gives a set of local transformations that allow every **while** program to be transformed systematically to one with at most one **while** loop. We consider a similar set of local transformations and give a purely equational proof of correctness for each. This result illustrates the use of Kleene algebra with tests and commutativity conditions in program equivalence proofs.

It seems to be a commonly held belief that this result has no purely schematic (i.e., propositional, uninterpreted) proof [Harel 1980]. The proofs of Hirose and Oya [1972] and Mirkowska [1972], as reported in Harel [1980], use extra variables to remember certain values at certain points in the program, either program counter values or the values of tests. Since having to remember these values seems unavoidable, one might infer that the only recourse is to introduce extra variables, along with an explicit assignment mechanism for assigning values to them. Thus, as the argument goes, proofs of this theorem cannot be purely propositional.

We disagree with this conclusion. The only purpose of these extra variables is to *preserve values across computations.* In our treatment, we only need to preserve the values of certain tests b over certain computations p.

We can handle this purely equationally as follows. Suppose we need to preserve the value of b across p. We introduce a new test c and commutativity condition $cp = pc$, which intuitively says that the computation of p does not affect the value of c. Then we insert in an appropriate place the program $s\,;bc + \overline{b}\overline{c}$, where s is a new atomic program. Intuitively, we regard s as assigning the value of b to some new Boolean variable that is tested in c, although there is no explicit mechanism for doing this; s is just an uninterpreted atomic program symbol. However, the guard $bc + \overline{b}\overline{c}$ (equivalently, $b \leftrightarrow c$; in the language of **while** programs, **if** b **then** c **else** $\overline{c}$) insures that b and c have the same Boolean value just after execution of s.

3.1 An Example

To illustrate this technique, consider the simple program

```
if b then begin p ; q end
     else  begin p ; r end                                    (21)
```

If the value of b were preserved by p, then we could rewrite this program more simply as

```
p ; if b then q else r                                        (22)
```

Formally, the assumption that the value of b is preserved by p takes the form of the commutativity condition $bp = pb$. By Lemma 2.3.1, we also have $\overline{b}p = p\overline{b}$. Expressed in the language of Kleene algebra, the equivalence of (21) and (22) becomes the equation

$$bpq + \overline{b}pr \;=\; p(bq + \overline{b}r).$$

This identity can be established by simple equational reasoning:

$$\begin{aligned} p(bq + \overline{b}r) &= pbq + p\overline{b}r && \text{by (8)} \\ &= bpq + \overline{b}pr && \text{by the commutativity assumptions.} \end{aligned}$$

But what if b is not preserved by p? This situation seems to call for a Boolean variable to remember the value of b across p and an explicit assignment mechanism to set the value of the variable. However, our approach is much less drastic. We introduce a new atomic test c and atomic program s along with the commutativity condition $pc = cp$, intuitively modeling the idea that c tests a variable that is not modified by p, and insert the program $s\,;bc + \overline{b}\overline{c}$.

We can now give a purely equational proof of the equivalence of the two programs

```
$s\,; bc + \overline{b}\overline{c}\,;$
**if** $b$ **then begin** $p\,; q$ **end**
     **else  begin** $p\,; r$ **end**
```

and

```
$s\,; bc + \overline{b}\overline{c}\,;$
$p\,;$ **if** $c$ **then** $q$ **else** $r$
```

using the axioms of Kleene algebra with tests and the commutativity condition $pc = cp$. In fact, we can even do away with the atomic program s: if the two programs are equivalent without the precomputation s, then they are certainly equivalent with it.

Removing the leading s and expressing the two programs in the language of Kleene algebra, the equivalence problem becomes

$$(bc + \overline{b}\overline{c})(bpq + \overline{b}pr) \;=\; (bc + \overline{b}\overline{c})p(cq + \overline{c}r). \tag{23}$$

Using the distributive laws (8) and (9) and the laws of Boolean algebra, we can simplify the left-hand side of (23) as follows:

$$\begin{aligned} (bc + \overline{b}\overline{c})(bpq + \overline{b}pr) &= bcbpq + \overline{b}\overline{c}bpq + bc\overline{b}pr + \overline{b}\overline{c}\overline{b}pr \\ &= bcpq + \overline{b}\overline{c}pr. \end{aligned}$$

The right-hand side of (23) simplifies in a similar fashion to the same expression:

$$\begin{aligned} (bc + \overline{b}\overline{c})p(cq + \overline{c}r) &= bcpcq + \overline{b}\overline{c}pcq + bcp\overline{c}r + \overline{b}\overline{c}p\overline{c}r \\ &= bccpq + \overline{b}\overline{c}cpq + bc\overline{c}pr + \overline{b}\overline{c}\overline{c}pr \\ &= bcpq + \overline{b}\overline{c}pr. \end{aligned}$$

Here the commutativity assumption is used in the second step.

3.2 A Normal Form Theorem

A program is in *normal form* if it is of the form

$$p\,;\ \textbf{while}\ b\ \textbf{do}\ q \tag{24}$$

where p and q are **while** free. The subprogram p is called the *precomputation* of the normal form.

In our approach, the folk theorem takes the following form:

THEOREM 3.2.1. *Every* **while** *program, suitably augmented with finitely many new subprograms of the form $s\,; bc+\overline{b}\overline{c}$, is equivalent to a* **while** *program in normal form, reasoning in Kleene algebra with tests under certain commutativity assumptions $cp = pc$.*

Theorem 3.2.1 is proved by induction on the structure of the program. Each programming construct accounts for one case in the inductive proof. For each case, we give a transformation that moves an inner **while** loop to the outside and an equational proof of its correctness. The inductive construction is performed from the inside out, i.e., smaller subprograms first.

In the statement of Theorem 3.2.1, we have been deliberately vague about where to insert subprograms $s\,; bc+\overline{b}\overline{c}$ and what commutativity assumptions $cp = pc$ need to be assumed. This will all be made explicit, but it is better done in the context of the construction below.

3.3 Conditional

We first show how to deal with a conditional containing programs in normal form in its **then** and **else** clauses. Consider the program

```
if $b$ then begin $p_1$ ; while $d_1$ do $q_1$ end
     else  begin $p_2$ ; while $d_2$ do $q_2$ end.
```

We introduce a new atomic program s and test c and assume that c commutes with p_1, p_2, q_1, and q_2.

Under these assumptions, we show that the programs

```
$s$ ; $bc + \overline{b}\overline{c}$ ;
if $b$ then begin $p_1$ ; while $d_1$ do $q_1$ end                (25)
     else  begin $p_2$ ; while $d_2$ do $q_2$ end
```

and

```
$s$ ; $bc + \overline{b}\overline{c}$ ;
if $c$ then $p_1$ else $p_2$ ;
while $cd_1 + \overline{c}d_2$ do                                  (26)
   if $c$ then $q_1$ else $q_2$
```

are equivalent. Note that if the two programs in the **then** and **else** clauses of (25) are in normal form, then (26) is in normal form. As argued in Section 3.1, we can do away with the precomputation s.

Removing s and rewriting (25) in the language of Kleene algebra, we obtain

$$(bc + \overline{b}\overline{c})(bp_1(d_1q_1)^*\overline{d}_1 + \overline{b}p_2(d_2q_2)^*\overline{d}_2)$$

which reduces by distributivity and Boolean algebra to

$$bcp_1(d_1q_1)^*\overline{d}_1 + \overline{b}\overline{c}p_2(d_2q_2)^*\overline{d}_2. \tag{27}$$

Similarly, (26) becomes

$$(bc + \overline{b}\overline{c})(cp_1 + \overline{c}p_2)((cd_1 + \overline{c}d_2)(cq_1 + \overline{c}q_2))^*\overline{cd_1 + \overline{c}d_2}. \tag{28}$$

The subexpression $\overline{cd_1 + \overline{c}d_2}$ of (28) is equivalent by propositional reasoning to $c\overline{d}_1 + \overline{c}\overline{d}_2$. Here we have used the familiar propositional equivalence

$$cd_1 + \overline{c}d_2 \;=\; (\overline{c} + d_1)(c + d_2)$$

and a De Morgan law. The other subexpressions of (28) can be simplified using distributivity and Boolean algebra to obtain

$$(bcp_1 + \overline{b}\overline{c}p_2)(cd_1q_1 + \overline{c}d_2q_2)^*(c\overline{d}_1 + \overline{c}\overline{d}_2). \tag{29}$$

Using distributivity, this can be broken up into the sum of four terms:

$$\begin{aligned} & bcp_1(cd_1q_1 + \overline{c}d_2q_2)^*c\overline{d}_1 & (30)\\ +\; & bcp_1(cd_1q_1 + \overline{c}d_2q_2)^*\overline{c}\overline{d}_2 & (31)\\ +\; & \overline{b}\overline{c}p_2(cd_1q_1 + \overline{c}d_2q_2)^*c\overline{d}_1 & (32)\\ +\; & \overline{b}\overline{c}p_2(cd_1q_1 + \overline{c}d_2q_2)^*\overline{c}\overline{d}_2. & (33)\end{aligned}$$

Under the commutativity assumptions, Lemma 2.3.2 implies that (31) and (32) vanish; and for the remaining two terms (30) and (33), we have

$$\begin{aligned} bcp_1(cd_1q_1 + \overline{c}d_2q_2)^*c\overline{d}_1 &= bcp_1(ccd_1q_1 + c\overline{c}d_2q_2)^*\overline{d}_1\\ &= bcp_1(d_1q_1)^*\overline{d}_1\\ \overline{b}\overline{c}p_2(cd_1q_1 + \overline{c}d_2q_2)^*\overline{c}\overline{d}_2 &= \overline{b}\overline{c}p_2(\overline{c}cd_1q_1 + \overline{c}\overline{c}d_2q_2)^*\overline{d}_2\\ &= \overline{b}\overline{c}p_2(d_2q_2)^*\overline{d}_2. \end{aligned}$$

The sum of these two terms is exactly (27).

3.4 Nested Loops

We next consider the case that is perhaps the most interesting: denesting two nested **while** loops. This construction is particularly remarkable in that no commutativity conditions (thus no extra variables) are needed; compare the corresponding transformations of Hirose and Oya [1972] and Mirkowska [1972], as reported in Harel [1980], which do use extra variables.

We show that the program

```
while $b$ do begin
              $p$ ;
              while $c$ do $q$                    (34)
           end
```

is equivalent to the program

```
if $b$ then begin
            $p$ ;
            while $b + c$ do
              if $c$ then $q$ else $p$
          end.                                                    (35)
```

This construction transforms a pair of nested **while** loops to a single **while** loop inside a conditional test. No commutativity conditions are assumed in the proof.

After this transformation, the **while** loop can be taken outside the conditional using the transformation of Section 3.3 (this part does require a commutativity condition). A dummy normal form such as 1**;** **while** 0 **do** 1 can be supplied for the missing **else** clause. Note that if the program inside the **begin...end** block of (34) is in normal form, then the resulting program will be in normal form.

Not surprisingly, the key property used in the proof is the denesting property (20), which equates a regular expression of *-depth two with another of *-depth one.

Translating to the language of Kleene algebra, (34) becomes

$$(bp(cq)^*\overline{c})^*\overline{b}, \tag{36}$$

and (35) becomes

$$bp((b+c)(cq+\overline{c}p))^*\overline{b+c}+\overline{b}. \tag{37}$$

The $\overline{b}$ in (37) is for the nonexistent **else** clause of the outermost conditional of (35). The starred subexpression in (37) can be simplified using the basic laws of Kleene and Boolean algebra:

$$\begin{aligned}(b+c)(cq+\overline{c}p) &= bcq+b\overline{c}p+ccq+c\overline{c}p\\ &= bcq+cq+\overline{c}bp\\ &= (b+1)cq+\overline{c}bp\\ &= cq+\overline{c}bp,\end{aligned}$$

and $\overline{b+c}$ is equivalent to $\overline{b}\overline{c}$ by a De Morgan Law. Making these transformations to (37), we obtain

$$bp(cq+\overline{c}bp)^*\overline{b}\overline{c}+\overline{b}. \tag{38}$$

Unwinding the outer loop in (36) using (12) and distributing $\overline{b}$ over the resulting sum, we obtain

$$\overline{b}+bp(cq)^*\overline{c}(bp(cq)^*\overline{c})^*\overline{b}.$$

Applying the sliding rule (19) to this expression gives

$$\overline{b}+bp(cq)^*(\overline{c}bp(cq)^*)^*\overline{c}\overline{b}.$$

Now removing $\overline{b}$, bp, and $\overline{b}\overline{c}$ from this expression and (38), it suffices to show

$$(cq+\overline{c}bp)^* \;=\; (cq)^*(\overline{c}bp(cq)^*)^*.$$

But this is just an instance of the denesting rule (20).

3.5 Eliminating Postcomputations

We wish to show that a postcomputation occurring after a **while** loop can be absorbed into the **while** loop. Consider a program of the form

$$\textbf{while } b \textbf{ do } p\,;q. \tag{39}$$

We can assume without loss of generality that b, the test of the **while** loop, commutes with the postcomputation q. If not, introduce a new test c that commutes with q along with an atomic program s that intuitively sets the value of c to b, and insert $s\,;bc+\overline{b}\overline{c}$ before the loop and in the loop after the body. We then claim that the two programs

$$s\,;bc+\overline{b}\overline{c}\,;\textbf{while } b \textbf{ do begin } p\,;s\,;bc+\overline{b}\overline{c} \textbf{ end}\,;q \tag{40}$$

$$s\,;bc+\overline{b}\overline{c}\,;\textbf{while } c \textbf{ do begin } p\,;s\,;bc+\overline{b}\overline{c} \textbf{ end}\,;q \tag{41}$$

are equivalent. This allows us to replace (40) with (41), for which the commutativity assumption holds.

For purposes of arguing that (40) and (41) are equivalent, we can omit all occurrences of s. The leading occurrences can be omitted as argued in Section 3.1, and the occurrences inside the **while** loops can be assumed to be part of p. The two trailing occurrences of q can be omitted as well. After making these modifications and writing the programs in the language of Kleene algebra, we need to show

$$(bc+\overline{b}\overline{c})(bp(bc+\overline{b}\overline{c}))^*\overline{b} \;=\; (bc+\overline{b}\overline{c})(cp(bc+\overline{b}\overline{c}))^*\overline{c}.$$

Using the sliding rule (19) on both sides, this becomes

$$((bc+\overline{b}\overline{c})bp)^*(bc+\overline{b}\overline{c})\overline{b} \;=\; ((bc+\overline{b}\overline{c})cp)^*(bc+\overline{b}\overline{c})\overline{c}.$$

By distributivity and Boolean algebra, both sides reduce easily to $(bcp)^*\overline{b}\overline{c}$.

Under the assumption that b and q commute, we show that (39) is equivalent to the program

```
if $\overline{b}$ then $q$
    else  while $b$ do begin
                        $p$ ;
                        if $\overline{b}$ then $q$                    (42)
                      end.
```

Note that if p and q are **while** free, then (42) consists of a program in normal form inside a conditional, which can be transformed to normal form using the construction of Section 3.3.

Written in the language of Kleene algebra, (39) becomes

$$(bp)^*\overline{b}q, \tag{43}$$

and (42) becomes

$$\overline{b}q+b(bp(\overline{b}q+b))^*\overline{b}. \tag{44}$$

Unwinding one iteration of (43) using (12) and distributing $\overline{b}q$ over the resulting sum, we obtain

$$\overline{b}q+bp(bp)^*\overline{b}q.$$

Eliminating the term $\overline{b}q$ from both sides, it suffices to prove

$$bp(bp)^*\overline{b}q \;=\; b(bp(\overline{b}q+b))^*\overline{b}.$$

Starting from the right-hand side, we have

$$\begin{aligned}
b(bp(\overline{b}q+b))^*\overline{b} &= b\overline{b} + bbp(\overline{b}q+b)(bp(\overline{b}q+b))^*\overline{b} && \text{by (12)}\\
&= bp(\overline{b}q+b)(bp(\overline{b}q+b))^*\overline{b} && \text{Boolean algebra}\\
&= bp((\overline{b}q+b)bp)^*(\overline{b}q+b)\overline{b} && \text{by sliding (19)}\\
&= bp((\overline{b}qb+b)p)^*\overline{b}q\overline{b} && \text{distributivity}\\
&= bp(bp)^*\overline{b}q && \text{commutativity assumption.}
\end{aligned}$$

3.6 Composition

The composition of two programs in normal form

```
    $p_1$ ;
    while $b_1$ do $q_1$ ;
    $p_2$ ;                                          (45)
    while $b_2$ do $q_2$
```

can be transformed to a single program in normal form. We have actually already done all the work needed to handle this case. First, we use the result of Section 3.5 to absorb the **while**-free program p_2 into the first **while** loop. We can also ignore p_1, since it can be absorbed into the precomputation of the resulting normal form when we are done. It therefore suffices to show how to transform a program

```
    while $b$ do $p$ ;
    while $c$ do $q$                                 (46)
```

to normal form, where p and q are **while** free.

As argued in Section 3.5, we can assume without loss of generality that the test b commutes with the program q by introducing a new test if necessary. Since b also commutes with c by Boolean algebra, by Theorem 2.3.3 we have that b commutes with the entire second **while** loop. This allows us to use the transformation of Section 3.5, absorbing the second **while** loop into the first. The resulting program looks like

```
    if $\overline{b}$ then while $c$ do $q$
        else  while $b$ do begin
                            $p$ ;                     (47)
                            if $\overline{b}$ then while $c$ do $q$
                          end.
```

Here we have just substituted **while** c **do** q for q in (42). At this point we can apply the transformation of Section 3.3 to the inner subprogram

```
    if $\overline{b}$ then while $c$ do $q$
```

using a dummy normal form for the omitted **else** clause, giving two nested loops in the **else** clause of (47); then the transformation of Section 3.4 to the **else** clause of (47); finally, the transformation of Section 3.3 to the entire resulting program, yielding a program in normal form.

The transformations of Sections 3.3–3.6 give a systematic method for moving **while** loops outside of any other programming construct. By applying these transformations inductively from the innermost loops outward, we can transform any program into a program in normal form.

None of these arguments needed an explicit assignment mechanism to Boolean variables, but only a dummy program s which does something unspecified (and which more often than not can be omitted in the actual proof) and a guard of the form $bc + \overline{b}\overline{c}$ that says that b and c somehow obtained the same Boolean value. Of course, in a real implementation, s might assign the value of the test b to some new Boolean variable that is tested in c, but this does not play a role in equational proofs. The point is that it is not significant exactly how Boolean values are preserved across computations, but rather that they *can be* preserved; and for the purposes of formal verification, this fact is completely captured by a commutativity assumption. Thus we are justified in our claim that we have given a purely equational proof of Theorem 3.2.1.

4. RELATED RESULTS

4.1 Decidability and Undecidability

How difficult is reasoning in Kleene algebra under commutativity assumptions $pq = qp$? In other words, how hard is it to determine whether the universal Horn formula $E \rightarrow s = t$ holds in all Kleene algebras or in all *-continuous Kleene algebras, where E is a finite set of equations of the form $pq = qp$?

It turns out that the problem is undecidable in general for *-continuous Kleene algebras. However, for conditions of the form $pb = bp$ where b is a test, the problem is *PSPACE*-complete for both Kleene algebras with tests and *-continuous Kleene algebras with tests; thus it is no more difficult than reasoning without extra conditions.

The undecidability result for general commutativity assumptions $pq = qp$ in *-continuous Kleene algebras follows from an undecidability result of Berstel [1979] (see also Gibbons and Rytter [1986]) on regular sets over partially commutative monoids and a result relating *-continuous Kleene algebras and regular sets [Kozen 1991, Lemma 7.1, p. 35]. A more direct proof has also been given by Cohen (private communication, 1994); see Kozen [1996]. This reduction is not valid in the absence of *-continuity for reasons explained below in Section 4.2.

The decidability result for conditions of the form $pb = bp$ follows from the fact that these conditions are equivalent to conditions of the form $p = 0$ (Lemma 2.3.1). Cohen [1994a] shows that Kleene algebra with conditions $p = 0$ reduces efficiently to Kleene algebra without conditions. In Kozen and Smith [1996], this result is extended to Kleene algebra with tests. Thus reasoning in the presence of commutativity conditions of the form used in Section 3 is no harder than reasoning without extra conditions.

4.2 Finite Axiomatizability

Berstel's result establishes more than just undecidability. The constructions of Berstel [1979] (see also Gibbons and Rytter [1986]) and Cohen (see Kozen [1996]) employ reductions from co-r.e. complete sets, namely the emptiness problem for

Turing machines and the complement of Post's correspondence problem, respectively. This implies that the universal Horn theory of the *-continuous Kleene algebras is not recursively enumerable, therefore not finitely axiomatizable. In fact, the theory is Π_1^1-complete [Kozen 1997]. The universal Horn theory of the Kleene algebras is finitely axiomatizable (the axiomatization is given in Section 2); thus the two theories do not coincide. This answers a question of Kozen [1994].

4.3 Completeness and Complexity

In Kozen and Smith [1996], we show that the equational theory of *-continuous Kleene algebras with tests is complete over relational models and that it admits free language-theoretic models consisting of sets of "guarded strings." These models are analogous to the regular sets of strings over a finite alphabet for Kleene algebra without tests. Using a technique based on Kozen [1994], it is also shown in Kozen and Smith [1996] that the equational theories of Kleene algebras with tests and *-continuous Kleene algebras with tests coincide. These results are analogous to the completeness result of Kozen [1994] for Kleene algebras without tests.

These results imply that the equational theory of Kleene algebras with tests reduces to PDL and is therefore decidable in at most exponential time. We have shown by different methods that the problem is actually *PSPACE*-complete [Cohen et al. 1996], thus no harder than the equational theory of Kleene algebras, which is known to be *PSPACE*-complete [Stockmeyer and Meyer 1973].

ACKNOWLEDGEMENTS

Thanks to Ernie Cohen, Neal Glew, David Gries, David Harel, Greg Morrisett, Vaughan Pratt, Fred B. Schneider, Frederick Smith, Dave Walker, and the anonymous referees for valuable comments. I am indebted to Ernie Cohen for his kind permission to include a previously unpublished result in an earlier version of this article [Kozen 1996, Theorem 4].

REFERENCES

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. 1975. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass.

Anderaa, S. 1965. On the algebra of regular expressions. *Appl. Math.*, 1–18.

Archangelsky, K. V. 1992. A new finite complete solvable quasiequational calculus for algebra of regular languages. Manuscript, Kiev State University.

Backhouse, R. C. 1975. Closure algorithms and the star-height problem of regular languages. Ph.D. thesis, Imperial College, London, U.K.

Berstel, J. 1979. *Transductions and Context-free Languages.* Teubner, Stuttgart, Germany.

Bloom, S. L. and Ésik, Z. 1993. Equational axioms for regular sets. *Math. Struct. Comput. Sci. 3*, 1–24.

Boffa, M. 1990. Une remarque sur les systèmes complets d'identités rationnelles. *Informatique Théoretique et Applications/Theoretical Informatics and Applications 24,* 4, 419–423.

Böhm, C. and Jacopini, G. 1966. Flow diagrams, Turing machines, and languages with only two formation rules. *Commun. ACM 9,* 5 (May), 366–371.

Cohen, E. 1994a. Hypotheses in Kleene algebra. Available from ftp://ftp.bellcore.com/pub/ernie/research/homepage.html

Cohen, E. 1994b. Lazy caching. Available from ftp://ftp.bellcore.com/pub/ernie/research/homepage.html

Cohen, E. 1994c. Using Kleene algebra to reason about concurrency control. Available from ftp://ftp.bellcore.com/pub/ernie/research/homepage.html

Cohen, E., Kozen, D. C., and Smith, F. 1996. The complexity of Kleene algebra with tests. Tech. Rep. TR96-1598, Cornell University. July.

Conway, J. H. 1971. *Regular Algebra and Finite Machines.* Chapman and Hall, London, U.K.

Fischer, M. J. and Ladner, R. E. 1979. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci. 18,* 2, 194–211.

Gibbons, A. and Rytter, W. 1986. On the decidability of some problems about rational subsets of free partially commutative monoids. *Theor. Comput. Sci. 48*, 329–337.

Gorshkov, P. V. 1989. Rational data structures and their applications. *Cybernetics 25,* 6 (Nov.–Dec.), 760–765.

Harel, D. 1980. On folk theorems. *Commun. ACM 23,* 7 (July), 379–389.

Hirose, K. and Oya, M. 1972. General theory of flowcharts. *Comment. Math. Univ. St. Pauli 21,* 2, 55–71.

Iwano, K. and Steiglitz, K. 1990. A semiring on convex polygons and zero-sum cycle problems. *SIAM J. Comput. 19,* 5, 883–901.

Kleene, S. C. 1956. Representation of events in nerve nets and finite automata. In *Automata Studies*, C. E. Shannon and J. McCarthy, Eds. Princeton University Press, Princeton, N.J., 3–41.

Kozen, D. C. 1981. On induction vs. *-continuity. In *Proc. Workshop on Logic of Programs*, Kozen, Ed. Lecture Notes in Computer Science, vol. 131. Springer-Verlag, New York, 167–176.

Kozen, D. C. 1990. On Kleene algebras and closed semirings. In *Proc. Math. Found. Comput. Sci.*, Rovan, Ed. Lecture Notes in Computer Science, vol. 452. Springer-Verlag, Banska-Bystrica, Slovakia, 26–47.

Kozen, D. C. 1991. *The Design and Analysis of Algorithms.* Springer-Verlag, New York.

Kozen, D. C. 1994. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput. 110,* 2 (May), 366–390.

Kozen, D. C. 1996. Kleene algebra with tests and commutativity conditions. In *Proc. Second Int. Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, T. Margaria and B. Steffen, Eds. Lecture Notes in Computer Science, vol. 1055. Springer-Verlag, Passau, Germany, 14–33.

Kozen, D. C. 1997. On the complexity of reasoning in Kleene algebra. In *Proc. 12th Symp. Logic in Comput. Sci.* IEEE, Los Alamitos, Ca. To appear. Cornell Tech. Report 97-1624, March 1997.

Kozen, D. C. and Smith, F. 1996. Kleene algebra with tests: Completeness and decidability. In *Proc. Conf. Computer Science Logic (CSL'96).* Lecture Notes in Computer Science. Springer-Verlag, New York. To appear. Cornell University Tech. Report TR96-1582, April, 1996.

Kozen, D. C. and Tiuryn, J. 1990. Logics of programs. In *Handbook of Theoretical Computer Science*, van Leeuwen, Ed. Vol. B. North Holland, Amsterdam, 789–840.

Krob, D. 1991. A complete system of B-rational identities. *Theor. Comput. Sci. 89,* 2 (October), 207–343.

Kuich, W. 1987. The Kleene and Parikh theorem in complete semirings. In *Proc. 14th Colloq. Automata, Languages, and Programming*, T. Ottmann, Ed. Lecture Notes in Computer Science, vol. 267. EATCS, Springer-Verlag, New York, 212–225.

Kuich, W. and Salomaa, A. 1986. *Semirings, Automata, and Languages.* Springer-Verlag, Berlin.

Mirkowska, G. 1972. Algorithmic logic and its applications. Ph.D. thesis, University of Warsaw. in Polish.

Ng, K. C. 1984. Relation algebras with transitive closure. Ph.D. thesis, University of California, Berkeley.

Pratt, V. 1988. Dynamic algebras as a well-behaved fragment of relation algebras. In *Proc. Conf. on Algebra and Computer Science*, D. Pigozzi, Ed. Lecture Notes in Computer Science, vol. 425. Springer-Verlag, Ames, Iowa, 77–110.

PRATT, V. 1990. Action logic and pure induction. In *Proc. Logics in AI: European Workshop JELIA '90*, J. van Eijck, Ed. Lecture Notes in Computer Science, vol. 478. Springer-Verlag, New York, 97–120.

REDKO, V. N. 1964. On defining relations for the algebra of regular events. *Ukrain. Mat. Z. 16*, 120–126. In Russian.

SAKAROVITCH, J. 1987. Kleene's theorem revisited: A formal path from Kleene to Chomsky. In *Trends, Techniques, and Problems in Theoretical Computer Science*, A. Kelemenova and J. Keleman, Eds. Lecture Notes in Computer Science, vol. 281. Springer-Verlag, New York, 39–50.

SALOMAA, A. 1966. Two complete axiom systems for the algebra of regular events. *J. Assoc. Comput. Mach. 13,* 1 (January), 158–169.

STOCKMEYER, L. J. AND MEYER, A. R. 1973. Word problems requiring exponential time. In *Proc. 5th Symp. Theory of Computing.* ACM, ACM, New York, 1–9.

TARSKI, A. 1941. On the calculus of relations. *J. Symb. Logic 6,* 3, 65–106.

Received May 1996; revised November 1996; accepted December 1996