Kleene Algebra with Tests

In semantics and logics of programs, Kleene algebra forms an essential component of Propositional Dynamic Logic (PDL) [8], in which it is mixed with Boolean algebra and modal logic to give a simple yet powerful system for modeling and 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 here a variant of Kleene algebra, called *Kleene algebra with tests* (KAT), for reasoning equationally with these constructs. KAT was introduced in [14, 15].

KAT has been applied successfully in a number of low-level verification tasks involving communication protocols, basic safety analysis, concurrency control, and local compiler optimizations [5, 6, 17, 1]. A useful feature of KAT in this regard is its ability to accommodate certain basic equational assumptions regarding the interaction of atomic instructions and tests. This feature makes KAT ideal for reasoning about the correctness of low-level code transformations.

In this lecture we introduce the system and illustrate its use by giving a complete equational proof of a classical folk theorem [10, 20] 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 [20], 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 can give a purely equational proof of correctness.

Definition of Kleene Algebra with Tests

A *Kleene algebra with tests* is a two-sorted algebra

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

where $^-$ is a unary operator defined only on B, such that

- $B \subseteq K$,
- $(K,\ +,\ \cdot,\ ^*,\ 0,\ 1)$ is a Kleene algebra, and
- $(B,\ +,\ \cdot,\ ^-,\ 0,\ 1)$ is a Boolean algebra.

The elements of B are called *tests*. We reserve the letters $p, q, r, \ldots$ for arbitrary elements of K and $a, b, c, \ldots$ for tests. In PDL, a test would be written b?, but since we are using different symbols for tests we can omit the ?.

This deceptively simple definition actually carries a lot of information in a concise package. Note the overloading of the operators $+, \cdot, 0, 1$, each of which plays two roles: applied to arbitrary elements of K, they refer to nondeterministic choice, composition, fail, and skip, respectively; and applied to tests, they take on the additional meaning of Boolean disjunction, conjunction, falsity, and truth, respectively. These two usages do not conflict—for example, sequential testing of b and c is the same as testing their conjunction—and their coexistence admits considerable economy of expression.

It follows immediately from the definition that $b \leq 1$ for all $b \in B$ (this is an axiom of Boolean algebra). It is tempting to define tests in an arbitrary Kleene algebra to be the set $\{p \in K \mid p \leq 1\}$, and this is the approach taken by [4]. This approach makes some sense in algebras of binary relations [21, 23], but it does not work in all Kleene algebras. For example, it rules out the tropical Kleene algebra described in Lecture **??**. In this algebra, $p \leq 1$ for all p, but the idempotence law $pp = p$ fails, so the set $\{p \in K \mid p \leq 1\}$ 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.

However, there is a more serious issue. Even in algebras of binary relations, this approach forces us to consider all elements $p \leq 1$ 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—that is, $p \leq 1$—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.

While Programs

We will 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 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:

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

Conditions

We will also be reasoning in the presence of extra assumptions. Recall that Kleene algebra and KAT are Horn theories, and these extra assumptions take the form of equational premises to the left of the implication symbol in a Horn formala.

There are several common forms of premises that arise in applications. The most common form is a commutativity condition $pq = qp$ or $pb = bp$. A commutativity condition between two atomic programs p and q expresses the fact that the two operations represented by p and q do not interfere with each other and may be done in either order. For example, if p and q are the assignments $x := 3$ and $y := 4$ respectively, then they do not affect each other. A commutativity condition $pb = bp$, where p is a program and b is a test, indicates that the execution of p does not affect the truth value of b. For example, if p is the assignment $x := 3$ and b is the test $y = 4$, then execution of p has no effect on the truth value of b.

Another common assumption comes in the form of a pair of equations $p = pb$ and $bp = b$. The first equation indicates that the execution of p causes b to become true, thus testing b after executing p is always redundant. The second equation indicates that if the sole purpose of p is to make b true, then there is no need to execute p if b is already true. For example, if p is the assignment $x := 3$ and b is the test $x = 3$, then these conditions hold.

This is useful for example when we want to eliminate a redundant assignment. We can use the first equation $p = pb$ to generate the condition b in a KAT expression, then use commutativity conditions to move b around until it comes up against another p, then use the second equation $bp = b$ to eliminate the second occurrence of p, then move the b back again and eliminate it by applying $p = pb$ in the opposite direction.

It stands to reason that if p does not affect b, then neither should it affect $\overline{b}$. This is indeed the case. Thus $pb = bp$ should be equivalent to $p\overline{b} = \overline{b}p$. More generally,

Lemma 0.1 *In any Kleene algebra with tests, the following are equivalent:*

(1) $bp = pc$

(2) $\overline{b}p = p\overline{c}$

(3) $bp\overline{c} + \overline{b}pc = 0$.

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

$$bp\overline{c} + \overline{b}pc \quad = \quad pc\overline{c} + \overline{b}bp \quad = \quad p0 + 0p \quad = \quad 0.$$

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

$$bp \quad = \quad bp(c + \overline{c}) \quad = \quad bpc + bp\overline{c} \quad = \quad bpc + 0 \quad = \quad bpc + \overline{b}pc \quad = \quad (b + \overline{b})pc \quad = \quad pc.$$

□

Of course, any pair of tests commute; that is, $bc = cb$. This is an axiom of Boolean algebra.

A Folk Theorem

One can give a purely equational proof, using KAT and commutativity conditions, of a classical result: *every* **while** *program can be simulated by a* **while** *program with at most one* **while** *loop.* This theorem is the subject of a treatise on folk theorems by Harel [10], who notes that it is commonly but erroneously attributed to Böhm and Jacopini [3] and who argues with some justification that it was known to Kleene. The version as stated here is originally due to Mirkowska [20], 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 is a commonly held belief that this result has no purely schematic (that is, propositional, uninterpreted) proof [10]. The proofs of [11, 20], as reported in [10], 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.

However, in KAT, if we need to preserve the value of a test b across an action p with which b is not guaranteed to commute, we can 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** $\overline{c}$) ensures that b and c have the same Boolean value just after execution of s.

To illustrate this technique, consider the simple program

$$\begin{array}{l} \textbf{if } b \textbf{ then } \{p\,;\, q\} \\ \qquad \textbf{else } \ \{p\,;\, r\} \end{array} \tag{1}$$

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

$$p\,;\, \textbf{if } b \textbf{ then } q \textbf{ else } r \tag{2}$$

Formally, the assumption that the value of b is preserved by p takes the form of the commutativity condition $bp = pb$. By Lemma 0.1, we also have $\overline{b}p = p\overline{b}$. Expressed in the language of Kleene algebra, the equivalence of (1) and (2) becomes the equation

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

This identity can be established by simple equational reasoning:

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

But what if b is not preserved by p? Here we can 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

$$\begin{array}{l} s\,;\, bc + \overline{b}\overline{c}\,; \\ \textbf{if } b \textbf{ then } \{p\,;\, q\} \\ \qquad \textbf{else } \ \{p\,;\, r\} \end{array}$$

and

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

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

$$(bc + \overline{b}\,\overline{c})(bpq + \overline{b}pr) \quad = \quad (bc + \overline{b}\,\overline{c})p(cq + \overline{c}r). \tag{3}$$

Using the distributive laws and the laws of Boolean algebra, we can simplify the left-hand side of (3) 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 (3) 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.

Normal Form

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

$$p\,;\ \textbf{while}\ b\ \textbf{do}\ q \tag{4}$$

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

Now we can show that 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 $cp = pc$.

This can be proved by induction on the structure of the program. Each programming construct accounts for one case in the inductive proof. For each case, we can 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; that is, smaller subprograms first.

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

$$\begin{aligned}&\textbf{if}\ b\ \textbf{then}\ \{p_1\,;\ \textbf{while}\ d_1\ \textbf{do}\ q_1\} \\ &\qquad\ \textbf{else}\ \ \{p_2\,;\ \textbf{while}\ d_2\ \textbf{do}\ q_2\}.\end{aligned}$$

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

$$\begin{array}{l} s\ ;\ bc+\overline{b}\overline{c}\ ; \\ \textbf{if } b \textbf{ then } \{p_1\ ;\ \textbf{while } d_1 \textbf{ do } q_1\} \\ \qquad \textbf{else } \ \{p_2\ ;\ \textbf{while } d_2 \textbf{ do } q_2\} \end{array} \tag{5}$$

and

$$\begin{array}{l} s\ ;\ bc+\overline{b}\overline{c}\ ; \\ \textbf{if } c \textbf{ then } p_1 \textbf{ else } p_2\ ; \\ \textbf{while } cd_1+\overline{c}d_2 \textbf{ do} \\ \quad \textbf{if } c \textbf{ then } q_1 \textbf{ else } q_2 \end{array} \tag{6}$$

are equivalent. Note that if the two programs in the **then** and **else** clauses of (5) are in normal form, then (6) is in normal form. As previously mentioned, we can do away with the precomputation s.

Removing s and rewriting (5) 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

$$bcp_1(d_1q_1)^*\overline{d}_1+\overline{b}\overline{c}p_2(d_2q_2)^*\overline{d}_2. \tag{7}$$

Similarly, (6) becomes

$$(bc+\overline{b}\overline{c})(cp_1+\overline{c}p_2)((cd_1+\overline{c}d_2)(cq_1+\overline{c}q_2))^*\overline{cd_1+\overline{c}d_2}. \tag{8}$$

The subexpression $\overline{cd_1+\overline{c}d_2}$ of (8) is equivalent by propositional reasoning to $c\overline{d}_1+\overline{c}\overline{d}_2$. Here we have used the familiar propositional equivalence

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

and a De Morgan law. The other subexpressions of (8) can be simplified using distributivity and Boolean algebra to obtain

$$(bcp_1+\overline{b}\overline{c}p_2)(cd_1q_1+\overline{c}d_2q_2)^*(c\overline{d}_1+\overline{c}\overline{d}_2). \tag{9}$$

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

$$\begin{aligned} & bcp_1(cd_1q_1+\overline{c}d_2q_2)^*c\overline{d}_1 && (10)\\ +\;& bcp_1(cd_1q_1+\overline{c}d_2q_2)^*\overline{c}\overline{d}_2 && (11)\\ +\;& \overline{b}\overline{c}p_2(cd_1q_1+\overline{c}d_2q_2)^*c\overline{d}_1 && (12)\\ +\;& \overline{b}\overline{c}p_2(cd_1q_1+\overline{c}d_2q_2)^*\overline{c}\overline{d}_2. && (13) \end{aligned}$$

Under the commutativity assumptions, (11) and (12) vanish; and for the remaining two terms (10) and (13), 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 (7).

Nested Loops

We next consider two nested **while** loops. This construction is particularly interesting in that no commutativity conditions (thus no extra variables) are needed, in contrast to the corresponding transformations of [11, 20], as reported in [10].

It can be shown that the program

```
while b do {
   p ;
   while c do q
}
```

is equivalent to the program

```
if b then {
   p ;
   while b + c do
      if c then q else p
}
```

or in the language of Kleene algebra,

$$(bp(cq)^*\overline{c})^*\overline{b} \quad = \quad bp((b+c)(cq+\overline{c}p))^*\overline{b+c} + \overline{b}. \tag{14}$$

The $\overline{b}$ on the right-hand side of (14) is for the nonexistent **else** clause of the outermost conditional of the second program.

This construction transforms a pair of nested **while** loops to a single **while** loop inside a conditional test. After this transformation, the **while** loop can be taken outside the conditional using the previous transformation (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.

We leave the proof of this equivalence as an exercise. Not surprisingly, the key property used in the proof is the denesting property $(p^*q)^*p^* = (p+q)^*$, which equates a regular expression of *-depth two with another of *-depth one.

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

$$\{\textbf{while } b \textbf{ do } p\} \, ; \, q. \tag{15}$$

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

$$s \, ; \, bc + \overline{b}\overline{c} \, ; \, \textbf{while } b \textbf{ do } \{p \, ; \, s \, ; \, bc + \overline{b}\overline{c}\} \, ; \, q \tag{16}$$

$$s \, ; \, bc + \overline{b}\overline{c} \, ; \, \textbf{while } c \textbf{ do } \{p \, ; \, s \, ; \, bc + \overline{b}\overline{c}\} \, ; \, q \tag{17}$$

are equivalent. This allows us to replace (16) with (17), for which the commutativity assumption holds.

For purposes of arguing that (16) and (17) are equivalent, we can omit all occurrences of s. The leading occurrences can be omitted as argued above, 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 KAT, 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 $p(qp)^* = (pq)^*p$ 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 can show that (15) is equivalent to the program

$$\begin{array}{l}
\textbf{if } \overline{b} \\
\quad \textbf{then } q \\
\quad \textbf{else while } b \textbf{ do } \{ \\
\quad\quad p \, ; \\
\quad\quad \textbf{if } \overline{b} \textbf{ then } q \\
\quad \}
\end{array} \tag{18}$$

Note that if p and q are **while** free, then (18) consists of a program in normal form inside a conditional, which can be transformed to normal form using the transformation above.

We leave this equivalence as an exercise.

Composition

The composition of two programs in normal form

$$
\begin{array}{l}
p_1\,; \\
\textbf{while } b_1 \textbf{ do } q_1\,; \\
p_2\,; \\
\textbf{while } b_2 \textbf{ do } q_2
\end{array}
\tag{19}
$$

can be transformed to a single program in normal form. First, we use the result of the previous section 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

$$
\begin{array}{l}
\textbf{while } b \textbf{ do } p\,; \\
\textbf{while } c \textbf{ do } q
\end{array}
\tag{20}
$$

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

As already argued, 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 a homework exercise we have that b commutes with the entire second **while** loop. This allows us to use the construction of the previous section to absorb the second **while** loop into the first. The resulting program looks like

$$
\begin{array}{l}
\textbf{if } \overline{b} \\
\quad \textbf{then while } c \textbf{ do } q \\
\quad \textbf{else } \{ \\
\quad\quad \textbf{while } b \textbf{ do } \{ \\
\quad\quad\quad p\,; \\
\quad\quad\quad \textbf{if } \overline{b} \textbf{ then while } c \textbf{ do } q \\
\quad\quad \} \\
\quad \}
\end{array}
\tag{21}
$$

Here we have just substituted **while** c **do** q for q in (18). At this point we can apply the conditional test transformation to the inner subprogram

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

using a dummy normal form for the omitted **else** clause, giving two nested loops in the **else** clause of (21); then denest the loops in the **else** clause of (21); finally, apply the conditional test transformation to the entire resulting program, yielding a program in normal form.

The transformations described above 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.

References

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

[2] Jean Berstel. *Transductions and Context-free Languages*. Teubner, Stuttgart, Germany, 1979.

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

[4] Ernie Cohen. Hypotheses in Kleene algebra. Available as ftp://ftp.telcordia.com/pub/ernie/research/homepage.html, April 1994.

[5] Ernie Cohen. Lazy caching. Available as ftp://ftp.telcordia.com/pub/ernie/research/homepage.html, 1994.

[6] Ernie Cohen. Using Kleene algebra to reason about concurrency control. Available as ftp://ftp.telcordia.com/pub/ernie/research/homepage.html, 1994.

[7] Ernie Cohen, Dexter Kozen, and Frederick Smith. The complexity of Kleene algebra with tests. Technical Report 96-1598, Computer Science Department, Cornell University, July 1996.

[8] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.

[9] Alan Gibbons and Wojciech Rytter. On the decidability of some problems about rational subsets of free partially commutative monoids. *Theoretical Computer Science*, 48:329–337, 1986.

[10] David Harel. On folk theorems. *Comm. Assoc. Comput. Mach.*, 23(7):379–389, July 1980.

[11] K. Hirose and M. Oya. General theory of flowcharts. *Comment. Math. Univ. St. Pauli*, 21(2):55–71, 1972.

[12] Dexter Kozen. *The Design and Analysis of Algorithms.* Springer-Verlag, New York, 1991.

[13] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.

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

[15] Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[16] Dexter Kozen. On the complexity of reasoning in Kleene algebra. In *Proc. 12th Symp. Logic in Comput. Sci.*, pages 195–202, Los Alamitos, Ca., June 1997. IEEE.

[17] Dexter Kozen and Maria-Cristina Patron. Certification of compiler optimizations using Kleene algebra with tests. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luis Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Proc. 1st Int. Conf. Computational Logic (CL2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 568–582, London, July 2000. Springer-Verlag.

[18] Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In D. van Dalen and M. Bezem, editors, *Proc. 10th Int. Workshop Computer Science Logic (CSL'96)*, volume 1258 of *Lecture Notes in Computer Science*, pages 244–259, Utrecht, The Netherlands, September 1996. Springer-Verlag.

[19] Dexter Kozen and Jerzy Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 789–840. North Holland, Amsterdam, 1990.

[20] G. Mirkowska. *Algorithmic Logic and its Applications.* PhD thesis, University of Warsaw, 1972. in Polish.

[21] K. C. Ng. *Relation Algebras with Transitive Closure.* PhD thesis, University of California, Berkeley, 1984.

[22] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *Proc. 5th Symp. Theory of Computing*, pages 1–9, New York, 1973. ACM.

[23] A. Tarski. On the calculus of relations. *J. Symb. Logic*, 6:3:73–89, 1941.