Kleene Algebra — Lecture 1

Tobias Kappé

January 2022

1 Housekeeping

Welcome to this project course on Kleene Algebra! If you're reading this, you're looking at the first set of lecture notes. These serve as an aid to both my verbal lectures, and as a reference for you to consult in your own time.

Let's get some housekeeping out of the way before we start. First of all, the best way to reach me is to send me an email. If you have any questions regarding course material or organization, please contact me there. Lecture notes and supplemental material are distributed through the website:

`https://staff.fnwi.uva.nl/t.w.j.kappe/teaching/ka`

Lectures consist of two parts, with a 15-minute break. This course is worth 6EC, so you can expect a full-time workload. Between lectures, you will work on homework assignments. Most of these assignments will require you to come up with some rigorous mathematical proof of a given statement, to allow you to get a more thorough understanding of the material. Some questions will be aimed at getting you to reflect on the larger themes of the course.

By the third week, you will have chosen a particular topic for further study, related to the course material. I will provide a list of topics for you to choose from, but if you like, you may also pursue something else that sparks your interest, in consultation with me. You will use this week to study your topic by reading some original scientific publications, for which I will provide links. You will have the opportunity to meet with me and get help if you are stuck, or we can discuss the material in depth, and see if I can suggest further reading.

In the final week, you will give a 30-minute lecture, explaining the chosen topic to your peers. The goal is for you to provide a basic understanding of the source material. The topic of this lecture will be discussed in advance with me, and I may provide some specific points that I would like you to touch upon.

Grading will be on a fail/pass basis, based on completeness and quality of answers to homework exercises, final presentations, and lecture attendance.

The objective for this course is twofold. First, we want to convince you that equational reasoning about programs in particular provides a productive perspective on programs in particular, and logical systems in general. Furthermore, in the second half of this course you will read some scientific writing that was cutting-edge at the time. It may be presented using unfamiliar notation, or with a different motivation; we want you to be able to consume this writing, understand it, and reproduce the technical contents in a suitable format.

2 Framework

You know how to manipulate mathematical expressions according to certain laws. For instance, multiplication *distributes* over addition — i.e., if x, y and z are numbers, then $x \cdot (y + z)$ is *equivalent* to $x \cdot y + x \cdot z$. Nevertheless, these expressions are not *equal*; for instance, the former expression includes just one multiplication, whereas the latter has two.

You may feel that algebraic expressions like this are bit mundane — didn't you already conquer this topic when you graduated from secondary school? Why on earth are we discussing this again? The answer lies not in the specific application, but rather in the *technique*: the idea that we can reason about equivalence of expressions by manipulating their syntax, using a few basic *axioms*. These ideas go back all the way to antiquity, when al-Khwarizmi worked out his method for finding the roots of quadratic polynomials, and have borne fruit in the form of further mathematical results up to the present day.

But there is no reason to limit equational reasoning to expressions pertaining to numbers. Indeed, computer programs can also be thought of as expressions in a certain syntax. For instance, the following two programs are equivalent:

```
while a and b do
  e;
while a do
  f;
  while a and b do
    e;
```

```
while a do
  if b then
    e;
  else
    f;
```

Part of the purpose of this course in general, and the first two lectures in particular, is to convince you that you can reason algebraically about programs such as these two. The composition operators for programs adhere to some rules that you already know, but others will be new and require some exercise.

Once you are comfortable with equational reasoning about programs, new questions arise. For instance, is there a set of axioms that allow us to prove *all* equations that are true about programs in general? How can we automate our reasoning about program equivalence? Can we solve equations that contain "unknown" programs, thereby obtaining a program that satisfies certain properties? We will address all of these questions in due time.

3 Syntax

Let's look at an extremely simple "programming language", which will form the basis of the systems that we will encounter in this course: the *rational expressions*. Rational expressions are built using a finite set of actions $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Each action represents a primitive operation that can be performed by our program; for example, `a` could mean "make the red LED blink once" or "verify that the next input letter is `a`". These actions are our building blocks: they can typically not be expressed as compositions of simpler actions.
We can build our programs from primitive programs:

- The program $e \cdot f$ first runs e, *and then* f.

- The program $e + f$ non-deterministically runs e *or* f.
- The program e^* *repeats* e some number of times, possibly zero.
- The constant program 0 has *no valid behavior*.
- The constant program 1 *skips* — i.e., it does nothing whatsoever.

Definition 1.1. Rational expressions (over Σ) are generated by the grammar:

$$e, f ::= 0 \mid 1 \mid \mathtt{a} \in \Sigma \mid e + f \mid e \cdot f \mid e^*$$

We will write $\mathbb{E}(\Sigma)$ for the set of expressions over Σ, and usually elide Σ.

You may have seen something very similar to this syntax if you used the *regular expressions* matching library available to your favorite programming language. The connection with regular expressions will be made explicit later this week.

You might wonder: why would I ever want to non-deterministically combine two programs, or run my program some indeterminate number of times? This programming language seems a bit wacky! The answer is two-fold:

1. It is useful to compare your program to another program that represents a set of desirable behaviors. That is to say, you want to check whether your program exhibits only behaviors that are good, or at least not bad. When specifying this behavior as a program, it is often useful to use non-determinism to "lump together" the behaviors that you deem permissible.
2. If, on the other hand, you want to specify your exact program, you need to tame the set of behaviors represented somehow. This can be done by carefully choosing the semantics of your primitive program (more on that in a moment), or by extending the syntax to include "tests" (next lecture).

4 An example program

Let's look at a small example of how you can program in this language. Suppose you are given a natural number n, and want to find out the *integer square root* of n — in other words, the largest number i such that $i^2 \leq n$. In a traditional programming language, you could write something like the following

```
$i \leftarrow 0$;
while $(i + 1)^2 \leq n$ do
  | $i \leftarrow i + 1$;
```

In our programming language, you could write this algorithm as follows:

$$\mathtt{init} \cdot (\mathtt{guard} \cdot \mathtt{incr})^* \cdot \mathtt{validate}$$

Here, `init`, `guard`, `incr` and `validate` are all primitive programs, where:

- `init` sets the variable i to zero, and leaves the other variables unaffected.
- `guard` checks if $(i + 1)^2 \leq n$. If this is the case, it does nothing; otherwise, it aborts program execution (i.e., kills this non-deterministic branch).

- `incr` increments the value of i, and leaves the other variables unaffected.
- `validate` is the opposite of `guard`: if $(i+1)^2 > n$, it does nothing, and it aborts the program in all other cases.

The checks `guard` and `validate` may seem weird — why would we want to deliberately abort the program? The idea here is to remove *invalid executions.*

For instance, suppose that, at the beginning of the program, $n = 0$. We first execute `init`, so $i = 0$ as well. There are now two possibilities:

- If we execute `guard` $\cdot$ `incr` some positive number of times, then the next thing we should do is run `guard`. But because $(i+1)^2 > n$, this aborts the program! This possiblity is therefore discarded.
- Otherwise, we can execute `guard` $\cdot$ `incr` *zero* times. In that case, the program executes `validate`, but survives, reaching the end of our program.

In total, there is only one way of executing the program for this input: skip the loop on `guard` $\cdot$ `incr` entirely.

5 Semantics

We have our programming language syntax, but we have not yet defined what programs in the language mean. One way to do this, is to assign a semantics in terms of a *relation* on machine states, representing the possible effects that a program may have on the machine state. To do this, we first need to fix the meaning of the primitive programs, which we do as follows.

Definition 1.2. An *interpretation* of Σ is a pair $\langle S, \sigma \rangle$, where S is a set (of *states*) and σ is a function from Σ to relations on S, i.e., $\sigma : \Sigma \to 2^{S \times S}$. We often denote an interpretation simply by σ, with S as the set of states.

When we fix an interpretation, we can give an interpretation of a program.

Definition 1.3. Given an interpretation $\sigma : \Sigma \to 2^{S \times S}$, we define the σ-semantics $[\![-]\!]_\sigma : \mathbb{E} \to 2^{S \times S}$ inductively, as follows:

$$[\![0]\!]_\sigma = \emptyset \qquad [\![1]\!]_\sigma = \mathsf{id}_S \qquad [\![\mathsf{a}]\!]_\sigma = \sigma(\mathsf{a})$$

$$[\![e+f]\!]_\sigma = [\![e]\!]_\sigma \cup [\![f]\!]_\sigma \qquad [\![e \cdot f]\!]_\sigma = [\![e]\!]_\sigma \circ [\![f]\!]_\sigma \qquad [\![e^*]\!]_\sigma = [\![e]\!]_\sigma^*$$

In the above, we write id_S for the identity relation on S, i.e.,

$$\mathsf{id}_S = \{\langle s, s \rangle : s \in S\}$$

We also write $R_1 \circ R_2$ for the relational composition of $R_1, R_2 \subseteq S \times S$, i.e.,

$$R_1 \circ R_2 = \{\langle s_1, s_3 \rangle : \exists s_2.\ s_1 \ R_1 \ s_2 \ R_2 \ s_3\}$$

Finally, we write R^* for the reflexive-transitive closure of R, or alternatively

$$R^* = \bigcup_{n \in \mathbb{N}} R^n \qquad \text{where} \quad R^0 = \mathsf{id}_S \quad \text{and} \quad R^{n+1} = R \circ R^n$$

Returning to our example program, we can give the following interpretation to the primitive programs. First, we need to fix our machine states. In this case, the program has two variables i and n, both of which are natural numbers. This means that the state of the machine is entirely described by the current values for i and n. Thus, we choose for S the set of functions $s : \{i, n\} \to \mathbb{N}$.

Next, we can fix our interpretation of the primitive programs, as follows:

$$\begin{aligned} \sigma(\texttt{init}) &= \{\langle s, s[0/i]\rangle : s \in S\} \\ \sigma(\texttt{guard}) &= \{\langle s, s\rangle : (s(i)+1)^2 \leq s(n)\} \\ \sigma(\texttt{incr}) &= \{\langle s, s[s(i)+1/i]\rangle : s \in S\} \\ \sigma(\texttt{validate}) &= \{\langle s, s\rangle : (s(i)+1)^2 > s(n)\} \end{aligned}$$

Here, we write $s[k/v]$ for the function that is the same as s, except that we replace the value of v with k. More precisely, $s[k/v] : \{i, n\} \to \mathbb{N}$ is given by

$$s[k/v](v') = \begin{cases} k & v = v' \\ s(v') & \text{otherwise} \end{cases}$$

6 Reasoning

Given two programs e and f, can we work out whether they have the same semantics? This is not an easy question at all! For one, as we have seen, the semantics is dependent on the interpretation given to primitive symbols. It then stands to reason that equivalence also depends on the interpretation. What's more, even if we fix an interpretation, we may still be in trouble when the semantic domain is infinite. For instance, under the example program and interpretation we saw before, it is not too hard to show that $[\![\texttt{init} \cdot (\texttt{guard} \cdot \texttt{incr})^* \cdot \texttt{validate}]\!]_\sigma$ is an infinite relation, and so we cannot compute it explicitly.

It turns out that we can still do *some* reasoning about programs if we abstract from the interpretation, and focus on the composition operators, whose definition does not (directly) depend on the interpretation. For example, let $e, f \in \mathbb{E}$. Now the program $e + f$ has the same semantics as $f + e$, *regardless of the interpretation* σ. After all, we can calculate the following:

$$[\![e + f]\!]_\sigma = [\![e]\!]_\sigma \cup [\![f]\!]_\sigma = [\![f]\!]_\sigma \cup [\![e]\!]_\sigma = [\![f + e]\!]_\sigma$$

This brings us to the following rules for equivalence of programs.

Definition 1.4 (Kleene Algebra)**.** We define $\equiv$ as the smallest congruence on $\mathbb{E}$ satisfying the following rules, for all programs $e, f, g \in \mathbb{E}$:

$$e + 0 \equiv e \qquad e + e \equiv e \qquad e + f \equiv f + e \qquad e + (f + g) \equiv (f + g) + h$$

$$e \cdot (f \cdot g) \equiv (e \cdot f) \cdot g \qquad e \cdot (f + g) \equiv e \cdot f + e \cdot h \qquad (e + f) \cdot g \equiv e \cdot g + f \cdot g$$

$$e \cdot 1 \equiv e \equiv 1 \cdot e \qquad e \cdot 0 \equiv 0 \equiv 0 \cdot e \qquad 1 + e \cdot e^* \equiv e^* \equiv 1 + e^* \cdot e$$

$$e + f \cdot g \leqq g \implies f^* \cdot e \leqq g \qquad e + f \cdot g \leqq f \implies e \cdot g^* \leqq f$$

Here, we use $e \leqq f$ as a shorthand for $e + f \equiv f$.

Most of these rules are fairly easy to understand intuitively. For instance, the rule $e + e \equiv e$ says that a non-deterministic choice between executing e or e is really no choice at all — the $+$ operator is *idempotent*. Similarly, choosing between e and $f + g$ is the same as choosing between $e + f$ and g, as witnessed by the rule $e + (f + g) \equiv (e + f) + g$ — the operator $+$ is *associative*.

To check that these rules are sound, we must validate that expressions related by $\equiv$ indeed have the same semantics. This turns out to be the case.

Lemma 1.5. *Suppose $e \equiv f$, and let σ be an interpretation. Then $[\![e]\!]_\sigma = [\![f]\!]_\sigma$.*

Proof sketch. We proceed by induction on $\equiv$ as a relation. In the base, we must check all of the rules on the first three lines. For instance, we have already validated that $[\![e + f]\!]_\sigma = [\![f + e]\!]_\sigma$, so the rule $e + f \equiv f + e$ is covered. Each of these cases is fairly straightforward; some of them are left as homework.

For the inductive step, there are two things we must check.

- Since $\equiv$ is a congruence, we must check that if $e_0 \equiv e_1$ and $f_0 \equiv f_1$, then $e_0 + e_1 \equiv f_0 + f_1$, and similarly for other operators. This is fairly straightforward: by induction, we have $[\![e_0]\!]_\sigma = [\![e_1]\!]_\sigma$ and $[\![f_0]\!]_\sigma = [\![f_1]\!]_\sigma$. Thus, $[\![e_0 + e_1]\!]_\sigma = [\![e_0]\!]_\sigma \cup [\![e_1]\!]_\sigma = [\![f_0]\!]_\sigma \cup [\![f_1]\!]_\sigma = [\![f_0 + f_1]\!]_\sigma$.

- We must also validate the last two rules. Let's look at the first one; the other can be treated similarly. Here, we have that $f^* \cdot e \leqq g$ because $e + f \cdot g \leqq g$. By induction, we then know that $[\![e]\!]_\sigma \cup [\![f]\!]_\sigma \circ [\![g]\!]_\sigma = [\![e + f \cdot g]\!]_\sigma \subseteq [\![g]\!]_\sigma$. We must show that $[\![f]\!]_\sigma^* \circ [\![e]\!]_\sigma = [\![f^* \cdot e]\!]_\sigma \subseteq [\![g]\!]_\sigma$. This amounts to showing that $[\![f]\!]_\sigma^n \circ [\![e]\!]_\sigma \subseteq [\![g]\!]_\sigma$ for all $n \in \mathbb{N}$.

  We proceed by induction on n. In the base, where $n = 0$, we have
  $$[\![f]\!]_\sigma^0 \circ [\![e]\!]_\sigma = \mathsf{id}_S \circ [\![e]\!]_\sigma = [\![e]\!]_\sigma \subseteq [\![g]\!]_\sigma.$$
  For the inductive step, assume that the claim holds for n. We calculate:
  $$[\![f]\!]_\sigma^{n+1} \circ [\![e]\!]_\sigma = [\![f]\!]_\sigma \circ [\![f]\!]_\sigma^n \circ [\![e]\!]_\sigma \subseteq [\![f]\!]_\sigma \circ [\![g]\!]_\sigma \subseteq [\![g]\!]_\sigma$$
  This completes the proof. □

Let's look at an example of how we can use these rules, particularly the last two rules, to show a non-trivial equivalence. Let $\mathtt{a}, \mathtt{b} \in \Sigma$, and consider the programs $\mathtt{a} \cdot (\mathtt{b} \cdot \mathtt{a})^*$ and $(\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a}$. We can intuitively understand that these programs do the same thing: they execute `a` and `b` in alternation, starting and ending with an `a`. To show equivalence, we need the following lemma.

Lemma 1.6. *If $e \leqq f$ and $f \leqq e$, then $e \equiv f$.*

Proof. Note that $e \leqq f$ and $f \leqq e$ are simply shorthand for $e + f \equiv f$ and $f + e \equiv e$, respectively. We can then derive that $e \equiv f + e \equiv e + f \equiv f$. □

We can now prove the desired equivalence.

Lemma 1.7. $\mathtt{a} \cdot (\mathtt{b} \cdot \mathtt{a})^* \equiv (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a}$

Proof. It suffices to show that $\mathtt{a} \cdot (\mathtt{b} \cdot \mathtt{a})^* \leqq (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a}$ and $(\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a} \leqq \mathtt{a} \cdot (\mathtt{b} \cdot \mathtt{a})^*$. To show that $\mathtt{a} \cdot (\mathtt{b} \cdot \mathtt{a})^* \leqq (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a}$, we can apply the last rule:
$$e + f \cdot g \leqq f \implies e \cdot g^* \leqq f$$

In our case, this means that we need to validate the premise:

$$\mathtt{a} + (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{a} \leqq (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a} \tag{1}$$

Let's look at the program on the left-hand side. We can derive

$$\begin{aligned}
\mathtt{a} + (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{a} &\equiv 1 \cdot \mathtt{a} + (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{a} && (1 \cdot e \equiv e) \\
&\equiv (1 + (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a} \cdot \mathtt{b}) \cdot \mathtt{a} && (e \cdot g + f \cdot g \equiv (e + f) \cdot g) \\
&\equiv (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a} && (1 + e^* \cdot e \equiv e)
\end{aligned}$$

We can then derive that

$$\begin{aligned}
\mathtt{a} + (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{a} + (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a} &\equiv (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a} + (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a} && \text{(above derivation)} \\
&\equiv (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a} && (e + e \equiv e)
\end{aligned}$$

which is precisely sufficient to conclude (1), and hence $\mathtt{a} \cdot (\mathtt{b} \cdot \mathtt{a})^* \leqq (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a}$. The proof that $(\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a} \leqq \mathtt{a} \cdot (\mathtt{b} \cdot \mathtt{a})^*$ is very similar, and is left as homework. □

7 Homework

1. In the example, you've seen how you can use programs like `guard` and `validate` to abort non-deterministic branches of program execution, and create something like a **while-do** construct using the star operator.

   You can use the same technique to encode a **if-then-else** construct, using the non-deterministic choice operator + and suitably chosen programs that abort execution under certain conditions.

   Suppose you want to encode the following piece of code as a rational expression, where M, L, R and y are valued as natural numbers:

   ```
   if $M^2 \leq y$ then
   | $L \leftarrow M$;
   else
   | $R \leftarrow M$;
   ```

   We can encode this snippet as $\mathtt{lte} \cdot \mathtt{writeL} + \mathtt{gt} \cdot \mathtt{writeR}$, where

   - `lte` and `gt` are programs that check whether $M^2 \leq y$ and $M^2 > y$ respectively, and abort the program if this is not the case.
   - `writeL` and `writeR` write M to L or to R, respectively.

   Your task is two-fold:

   (a) Give a semantic domain S that encodes the state of this four-variable program. *Hint: look back at how we encoded two-variable state.*

   (b) Give an interpretation σ to each primitive program, in S.

2. The snippet you saw above is actually part of a larger program, which finds the integer square root of the input y by means of binary search:

```
$L \leftarrow 0$;
$R \leftarrow y + 1$;
while $L < R - 1$ do
    $M \leftarrow \lfloor \frac{L+R}{2} \rfloor$;
    if $M^2 \leq y$ then
        $L \leftarrow M$;
    else
        $R \leftarrow M$;
```

(a) Come up with suitable primitive programs and their description to encode this program. You may reuse the primitive programs from the last exercise (no need to repeat their description).

(b) Give a rational expression to encode the program as a whole. You may reuse the rational expression for the **if-then-else** construct.

(c) Give an interpretation σ to the new primitive programs that you thought of, using the same semantic domain S as in the last exercise.

3. Recall that $e \leqq f$ is shorthand for $e + f \equiv f$. Let's examine some useful properties of this shorthand. In the following, let $e, f, g \in \mathbb{E}$.

   (a) Show that $e \leqq e$ always holds.

   (b) Show that if $e \leqq f$ and $f \leqq g$, then $e \leqq g$.

   (c) Show that if $e \leqq f$, then $e + g \leqq f + g$.

   (d) Show that if $e \leqq f$, then $e \cdot g \leqq f \cdot g$.

   (e) *(Optional)* Show that if $e \leqq f$, then $e^* \leqq f^*$.

4. In Lemma 1.7, we showed that $\mathtt{a} \cdot (\mathtt{b} \cdot \mathtt{a})^* \leqq (\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a}$. Argue the other direction, i.e., that $(\mathtt{a} \cdot \mathtt{b})^* \cdot \mathtt{a} \leqq \mathtt{a} \cdot (\mathtt{b} \cdot \mathtt{a})^*$. *Hint: the structure of the proof is very similar to the direction already shown above!*

5. *(Optional)* Let $e, f \in \mathbb{E}$. Prove that $(e + f)^* \equiv e^* \cdot (f \cdot e^*)^*$.

8 Bibliographical notes

Just to make this abundantly clear: none of the material discussed in these notes was first discovered by me. What follows is a brief and non-exhaustive set of pointers to literature relevant to the ideas we looked at today.

A good argument making the case for comparing programs through algebraic reasoning comes from Hoare and collaborators [HHH$^+$87]. The operators of Kleene Algebra were proposed by Kleene [Kle56]. Before that, Tarski and his students studied the algebraic properties of the relational model we have seen, using operators specific to relations as well [Tar41].

There exists a wild variety of axioms for Kleene Algebra; possibilities include the ones proposed by Salomaa [Sal66], Conway [Con71], Krob [Kro90], Boffa [Bof90], and Kozen [Koz94]. The axioms presented in this lecture are due to Kozen [Koz94]. The idea of encoding traditional program structures using rational expressions can also be traced back to Kozen [Koz96].

References

[Bof90] Maurice Boffa. Une remarque sur les systèmes complets d'identités rationnelles. *ITA*, 24:419–428, 1990.

[Con71] John Horton Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, Ltd., London, 1971.

[HHH$^+$87] Tony Hoare, Ian J. Hayes, Jifeng He, Carroll Morgan, A. W. Roscoe, Jeff W. Sanders, Ib Holm Sørensen, J. Michael Spivey, and Bernard Sufrin. Laws of programming. *Commun. ACM*, 30(8):672–686, 1987. `doi:10.1145/27651.27653`.

[Kle56] Stephen C. Kleene. Representation of events in nerve nets and finite automata. In Claude E. Shannon and John McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, 1956.

[Koz94] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Inf. Comput.*, 110(2):366–390, 1994. `doi:10.1006/inco.1994.1037`.

[Koz96] Dexter Kozen. Kleene algebra with tests and commutativity conditions. In *TACAS*, pages 14–33, 1996. `doi:10.1007/3-540-61042-1_35`.

[Kro90] Daniel Krob. A complete system of b-rational identities. In *ICALP*, pages 60–73, 1990. `doi:10.1007/BFb0032022`.

[Sal66] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, 1966. `doi:10.1145/321312.321326`.

[Tar41] Alfred Tarski. On the calculus of relations. *J. Symb. Log.*, 6(3):73–89, 1941. `doi:10.2307/2268577`.