Kleene Algebra — Lecture 2

Tobias Kappé

January 2022

1 Today's lecture

In the last lecture, we saw the syntax of rational expressions, and we gave one possible semantics that was based on interpreting the primitive programs as relations. We also saw that we can reason about equivalence of these expressions using a set of rules, and that we can prove some non-trivial equivalences, like

$$e \cdot (f \cdot e)^* \equiv (e \cdot f)^* \cdot e \qquad (e + f)^* \equiv e^* \cdot (f \cdot e^*)^*$$

We also saw that, if we choose our primitive programs well, we can encode constructs like **while-do** and **if-then-else**, using non-deterministic composition operators like $+$ and $*$ to obtain programs that proceed deterministically.

These "filtering" programs are commonly given an interpretation of the form $\{\langle s, s\rangle : P(s)\}$, where P was some sort of predicate. They check if the machine is in a state satisfying P, and if this is the case, continuing as usual; on the other hand, if the current state does *not* satisfy P, no further step is possible.

It turns out that these filtering programs actually admit their own equations: those of *Boolean algebra*. For instance, if the interpretation of $\mathtt{a}, \mathtt{b} \in \Sigma$ is a subidentity relation, then $\mathtt{a} \cdot \mathtt{b}$ has the same semantics as $\mathtt{b} \cdot \mathtt{a}$. However, this equality does not hold in general for programs — that would be very strange!

Today, we will augment our syntax of rational expressions and add such programs on their own terms, as *tests*. This will allow us to add new rules specialized for tests, encode well-known programming constructs more uniformly, and also prove some more familiar general equivalences between programs.

2 Syntax

We are going to add a new layer to the syntax of rational expressions, to obtain *guarded rational expressions*. To do this, we also need to fix a (finite) set of *primitive tests*, which we denote by T. Think of these as the checks that cannot be written as compositions of other checks in your program.

Definition 2.1. *Boolean expressions* (over T) and the *guarded rational expressions* (over Σ and T) are generated by the first and second layer of the grammar:

$$b, c ::= 0 \mid 1 \mid \mathtt{t} \in T \mid b + c \mid b \cdot c \mid \bar{b} \qquad e, f ::= b \mid \mathtt{a} \in \Sigma \mid e + f \mid e \cdot f \mid e^*$$

We will write $\mathbb{B}(T)$ for the set of Boolean expressions over T, and $\mathbb{G}(\Sigma, T)$ for the set of guarded rational expressions over Σ and T; usually, we elide Σ and T.

Observe that our syntax for disjunction (resp. conjunction) on the level of Boolean expressions overlaps with that of choice (resp. concatenation). This might seem cumbersome, but it's actually not a problem. For instance, we will see later on that the semantics of the $b + c$, where $b + c$ is a test, is the same as the semantics of $b + c$ where b and c are tests, composed non-deterministically.

Furthermore, note that we create a program from a Boolean expression b. Intuitively, if b is a Boolean expression, then the *program* b should be read as "check whether b holds, and continue if it does (leaving the state unchanged); otherwise, abort execution". In other words, you can think of such a program as the **assert** statement in some programming languages.

Lastly, note that the constant programs 0 and 1 have moved to the first layer of our syntax. As Boolean expressions, you should think of them as the constants **false** and **true** respectively. Their interpretations as *programs* matches with the intuition about **assert** statements: you can see the program 0 as **assert false**: the program that unconditionally aborts execution, while **assert true** is the program that unconditionally does nothing, and simply continues.

3 Semantics

Our new syntax needs a new semantics. First, we need to fix an interpretation for the primitive tests, just like we did for the primitive programs. Recall that programs transform state, so we assigned each primitive program a relation on states. Tests, on the other hand, are not meant to change the state at all; rather, their behavior is completely determined by whether or not they hold in a certain state. This motivates the following definition.

Definition 2.2. A *(guarded) interpretation* is a triple $\langle S, \tau, \sigma \rangle$, where S is a set and $\tau : T \to 2^S$ as well as $\sigma : \Sigma \to 2^{S \times S}$ are functions.

You can think of τ as assigning, to each primitive test $\mathtt{t} \in T$, the set of states $\tau(\mathtt{t})$ where $\mathtt{t}$ is true. We can then build on this primitive interpretation to extend it to obtain semantics of Boolean expressions, as follows:

Definition 2.3. For an interpretation with $\tau : T \to 2^S$, we define $(\!|-|\!)_\tau : \mathbb{B} \to S$ by induction on the input expression, as follows:

$$(\!|0|\!)_\tau = \emptyset \qquad (\!|1|\!)_\tau = S \qquad (\!|\mathtt{t}|\!)_\tau = \tau(\mathtt{t})$$

$$(\!|b + c|\!)_\tau = (\!|b|\!)_\tau \cup (\!|c|\!)_\tau \qquad (\!|b \cdot c|\!)_\tau = (\!|b|\!)_\tau \cap (\!|c|\!)_\tau \qquad (\!|\bar{b}|\!)_\tau = S \setminus (\!|b|\!)_\tau$$

Intuitively, $(\!|b|\!)_\tau$ should be thought of as the set of states where the test b is true. For instance, when $b = \mathtt{t}_0 \cdot \mathtt{t}_1$, this comes down to the set of all states where both $\mathtt{t}_0$ and $\mathtt{t}_1$ are true, which is exactly $\tau(\mathtt{t}_0) \cap \tau(\mathtt{t}_1)$.

We can build on this semantics for Boolean expressions to obtain a semantics for guarded rational expressions. As you might expect, this semantics turns the set of states satisfied by a test into a sub-identity relation.

Definition 2.4. For an interpretation with $\tau : T \to 2^S$ and $\sigma : \Sigma \to 2^{S \times S}$, we define the $\langle \sigma, \tau \rangle$-semantics $[\![-]\!]_{\sigma,\tau} : \mathbb{G} \to 2^{S \times S}$ inductively, as follows:

$$[\![b]\!]_{\sigma,\tau} = \{\langle s, s \rangle : s \in (\!|b|\!)_\tau\} \qquad [\![\mathtt{a}]\!]_{\sigma,\tau} = \sigma(\mathtt{a})$$

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

Let $b, c \in \mathbb{B}$. As hinted, there are now two ways of interpreting $b + c$. Fortunately, for us, they coincide, and so we have no ambiguity in our semantics. If you like, you can try proving this as an optional exercise.

Let's consider an example of using guarded rational expressions to encode the integer square root program that we saw during the last lecture:

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

This time, instead of using *two primitive programs* to encode $(i+1)^2 \leq n$ and its negation, we are going to use *a single primitive test*. Let's call it `bound`, and re-use the programs `init` and `incr`. We end up with the following program:

$$\texttt{init} \cdot (\texttt{bound} \cdot \texttt{incr})^* \cdot \overline{\texttt{bound}}$$

Let's use the same semantic domain to interpret this program — i.e., S is the set of functions from $\{i, n\}$ to $\mathbb{N}$. We need to give an interpretation of the primitive test `bound`, and the primitive programs `init` and `incr`, which we do as follows:

$$\tau(\texttt{bound}) = \{s \in S : (s(i)+1)^2 \leq s(n)\}$$

$$\sigma(\texttt{init}) = \{\langle s, s[0/i]\rangle : s \in S\} \qquad \sigma(\texttt{incr}) = \{\langle s, s[s(i)+1/i]\rangle : s \in S\}$$

4 Flow control

By now, you are familiar with techniques that encode deterministic flow-control in rational expressions. Because we have an operator for negation on tests, we can also generalize this pattern to obtain the following shorthands, for Boolean expressions b and guarded rational expressions e and f:

$$\textbf{if } b \textbf{ then } e \textbf{ else } f := b \cdot e + \overline{b} \cdot f \qquad \textbf{while } b \textbf{ do } e := (b \cdot e)^* \cdot \overline{b}$$

For example, our integer square root finding program can be written as

$$\texttt{init} \cdot \textbf{while } \texttt{bound} \textbf{ do } \texttt{incr}$$

This nicer, more readable syntax, will enable us to make some general statements about imperative programs in a moment. Keep in mind, however, that "under the hood", expressions like these encode plain old guarded rational expressions.

Let's go over the intuition behind these encodings. For the **if-then-else** construct, we create a program that non-deterministically chooses between $b \cdot e$ and $\overline{b} \cdot f$. If the current state satisfies b it does *not* satisfy $\overline{b}$, and only the result of the left-hand program will survive — i.e., the one obtained by executing e. Conversely, if the current state satisfies $\overline{b}$, then only the right-hand branch will execute. Thus, this program chooses between executing e or f, based on b.

Furthermore, the **while-do** construct is a program that executes $b \cdot e$ some (non-deterministic) number of times, and finally executes $\overline{b}$. Note however, that the only non-deterministic choice here is between executing $b \cdot e$ one more time, and executing $\overline{b}$ — again, only one branch "survives", which is the one where e is executed as long as b is true, and halts when b is no longer true.

5 Boolean reasoning

Note how, in the syntax above (and in contrast with the last lecture), there was no need to create a separate primitive test or program to encode the negation of the loop guard — we can simply use the negation operator that is native to our syntax. More generally, we can make the internal structure of our tests part of the syntax now, including conjunctions and disjunctions, and *reason about it*.

In order to do that, however, we are going to need a set of rules that allows us to reason about tests. Luckily, it turns out that the laws of *Boolean algebra* augment our reasoning principles in a suitable way, as follows.

Definition 2.5 (Boolean algebra)**.** We define $\equiv_{\mathbb{B}}$ as the smallest congruence on $\mathbb{B}$ satisfying the following rules, for all $b, c, d \in \mathbb{B}$:

$$b + 0 \equiv_{\mathbb{B}} b \qquad b + c \equiv_{\mathbb{B}} c + b \qquad b + \overline{b} \equiv_{\mathbb{B}} 1 \qquad b + (c + d) \equiv_{\mathbb{B}} (b + c) + d$$

$$b \cdot 1 \equiv_{\mathbb{B}} b \qquad b \cdot c \equiv_{\mathbb{B}} c \cdot b \qquad b \cdot \overline{b} \equiv_{\mathbb{B}} 0 \qquad b \cdot (c \cdot d) \equiv_{\mathbb{B}} (b \cdot c) \cdot d$$

$$b + c \cdot d \equiv_{\mathbb{B}} (b + c) \cdot (b + d) \qquad b \cdot (c + d) \equiv_{\mathbb{B}} b \cdot c + b \cdot d$$

Whenever it is clear that we are reasoning about Boolean expressions, we will drop the subscript $\mathbb{B}$ and simply write $\equiv$.

These rules may seem rather terse, but it turns out that they are *complete*: every equality that is true under any interpretation of the primitive tests τ can be proved using these rules. Unfortunately, we do not have the time to fully prove this claim; instead, let's take the rules for a spin, to see how we can use them, and what kind of useful things we can show. Here is one.

Lemma 2.6. *If b and c are tests such that $b + c \equiv_{\mathbb{B}} 1$ and $b \cdot c \equiv_{\mathbb{B}} 0$, then $b \equiv_{\mathbb{B}} \overline{c}$.*

Note now, intuitively, the premises sort of describe b and c as opposites: at least (left premise) and at most (right premise) is true. Let's see the proof.

Proof of Lemma 2.6. We derive as follows:

$$\begin{aligned}
\overline{c} &\equiv \overline{c} + 0 && \text{(unit)} \\
&\equiv \overline{c} + b \cdot c && \text{(premise)} \\
&\equiv (\overline{c} + b) \cdot (\overline{c} + c) && \text{(distributivity)} \\
&\equiv (\overline{c} + b) \cdot 1 && \text{(eliminated middle)} \\
&\equiv (\overline{c} + b) \cdot (\overline{b} + b) && \text{(eliminated middle)} \\
&\equiv \overline{c} \cdot \overline{b} + b && \text{(distributivity)} \\
&\equiv \overline{c} \cdot \overline{b} \cdot 1 + b && \text{(unit)} \\
&\equiv \overline{c} \cdot \overline{b} \cdot (b + c) + b && \text{(premise)} \\
&\equiv \overline{c} \cdot \overline{b} \cdot b + \overline{c} \cdot \overline{b} \cdot c + b && \text{(distributivity)} \\
&\equiv \overline{c} \cdot \overline{b} \cdot b + \overline{b} \cdot \overline{c} \cdot c + b && \text{(commutativity)} \\
&\equiv \overline{c} \cdot 0 + \overline{b} \cdot 0 + b && \text{(absurdity)} \\
&\equiv 0 + 0 + b && \text{(absorption)} \\
&\equiv b && \text{(unit)} \quad \square
\end{aligned}$$

You are most likely already familiar with DeMorgan's laws. Let's prove the algebraic encoding of the first law from our axioms, using Lemma 2.6.

Lemma 2.7 (De Morgan's first law)**.** *If b and c are tests, then $\overline{b+c} \equiv \overline{b} \cdot \overline{c}$.*

Proof. By the previous lemma, it suffices to show the following two equivalences:

$$\overline{b} \cdot \overline{c} + b + c \equiv 1 \qquad\qquad \overline{b} \cdot \overline{c} \cdot (b + c) \equiv 0$$

For the first equivalence, we derive

$$\begin{aligned}
\overline{b} \cdot \overline{c} + b + c &\equiv (\overline{b} + b + c) \cdot (\overline{c} + b + c) && \text{(distributivity)} \\
&\equiv (\overline{b} + b + c) \cdot (b + \overline{c} + c) && \text{(commutativity)} \\
&\equiv (1 + c) \cdot (b + 1) && \text{(eliminated middle)} \\
&\equiv 1 \cdot 1 && \text{(absorption)} \\
&\equiv 1 && \text{(idempotence)}
\end{aligned}$$

For the second equivalence, we derive

$$\begin{aligned}
\overline{b} \cdot \overline{c} \cdot (b + c) &\equiv \overline{b} \cdot \overline{c} \cdot b + \overline{b} \cdot \overline{c} \cdot c && \text{(distributivity)} \\
&\equiv \overline{c} \cdot \overline{b} \cdot b + \overline{b} \cdot \overline{c} \cdot c && \text{(commutativity)} \\
&\equiv \overline{c} \cdot 0 + \overline{b} \cdot 0 && \text{(absurdity)} \\
&\equiv 0 + 0 && \text{(absorption)} \\
&\equiv 0 && \text{(idempotence)} \quad \square
\end{aligned}$$

Before we wrap up our discussion of Boolean equivalence, let's check if these rules are fit for reasoning about Boolean expressions:

Lemma 2.8 (Soundness for Boolean algebra)**.** *Let $b, c \in \mathbb{B}$, and let $\tau : T \to 2^S$ be (part of an) interpretation. If $b \equiv_{\mathbb{B}} c$, then $(\!|b|\!)_\tau = (\!|c|\!)_\tau$.*

Proof sketch. We again proceed by induction on the construction of $\equiv_{\mathbb{B}}$. In the base, we need to validate the generating rules. For instance, for $b + c \equiv_{\mathbb{B}} c + b$:

$$(\!|b + c|\!)_\tau = (\!|b|\!)_\tau \cup (\!|c|\!)_\tau = (\!|c|\!)_\tau \cup (\!|b|\!)_\tau = (\!|c + b|\!)_\tau$$

Each of these is fairly straightforward; you will look at one case in the homework.

For the inductive step, we need to validate the congruence rules. For instance, if $b_0 \cdot c_0 \equiv_{\mathbb{B}} b_1 \cdot c_1$ because $b_0 \equiv_{\mathbb{B}} b_1$ and $c_0 \equiv_{\mathbb{B}} c_1$, then we know that $(\!|b_0|\!)_\tau = (\!|b_1|\!)_\tau$ and $(\!|c_0|\!)_\tau = (\!|c_1|\!)_\tau$ by induction. We can then derive that

$$(\!|b_0 \cdot c_0|\!)_\tau = (\!|b_0|\!)_\tau \cap (\!|c_0|\!)_\tau = (\!|b_1|\!)_\tau \cap (\!|c_1|\!)_\tau = (\!|b_1 \cdot c_1|\!)_\tau$$

The other cases are similar. $\square$

6 Guarded reasoning

Now that we have discussed the rules for Boolean equivalence, we can weave the rules we use at that level together with the rules for rational expressions that we already know. It turns out they combine quite well: we can simply adjoin both sets of rules, making sure that $\equiv_{\mathbb{B}}$ applies only to the Boolean terms.

Definition 2.9 (Kleene Algebra with Tests)**.** We define $\equiv_{\mathbb{G}}$ as the smallest congruence on $\mathbb{G}$ satisfying the following rules for all $b, c \in \mathbb{B}$ and $e, f, g \in \mathbb{G}$:

$$b \equiv_{\mathbb{B}} c \implies b \equiv_{\mathbb{G}} c \qquad e + 0 \equiv_{\mathbb{G}} e \qquad e + e \equiv_{\mathbb{G}} e \qquad e + f \equiv_{\mathbb{G}} f + e$$

$$e + (f + g) \equiv_{\mathbb{G}} (f + g) + h \qquad e \cdot (f \cdot g) \equiv_{\mathbb{G}} (e \cdot f) \cdot g$$

$$e \cdot (f + g) \equiv_{\mathbb{G}} e \cdot f + e \cdot h \qquad (e + f) \cdot g \equiv_{\mathbb{G}} e \cdot g + f \cdot g$$

$$e \cdot 1 \equiv_{\mathbb{G}} e \equiv_{\mathbb{G}} 1 \cdot e \qquad e \cdot 0 \equiv_{\mathbb{G}} 0 \equiv_{\mathbb{G}} 0 \cdot e \qquad 1 + e \cdot e^* \equiv_{\mathbb{G}} e^* \equiv_{\mathbb{G}} 1 + e^* \cdot e$$

$$e + f \cdot g \leqq_{\mathbb{G}} g \implies f^* \cdot e \leqq_{\mathbb{G}} g \qquad e + f \cdot g \leqq_{\mathbb{G}} f \implies e \cdot g^* \leqq_{\mathbb{G}} f$$

Just like before, we use $e \leqq_{\mathbb{G}} f$ as a shorthand for $e + f \equiv_{\mathbb{G}} f$. We will also drop the subscript $\mathbb{G}$ when the expressions being reasoned on are clear from context.

Lemma 2.10 (Soundness for Kleene Algebra with Tests)**.** *Let $e, f \in \mathbb{G}$, and let $\langle S, \tau, \sigma \rangle$ be a guarded interpretation. If $e \equiv_{\mathbb{G}} f$, then $[\![e]\!]_{\sigma,\tau} = [\![f]\!]_{\sigma,\tau}$.*

Let's take our shiny set of rules for a spin to prove some truths about programs that you may already be familiar with from programming.

Lemma 2.11. *Let $e, f \in \mathbb{G}$ as well as $b \in \mathbb{B}$. The following holds:*

$$\textbf{if } b \textbf{ then } e \textbf{ else } f \equiv \textbf{if } \overline{b} \textbf{ then } f \textbf{ else } e$$

Proof. This is a matter of unrolling the syntactic sugar and applying our rules:

$$\begin{aligned}
\textbf{if } b \textbf{ then } e \textbf{ else } f &= b \cdot e + \overline{b} \cdot f && \text{(by definition)} \\
&\equiv \overline{b} \cdot f + b \cdot e && \text{(commutativity)} \\
&\equiv \overline{b} \cdot f + \overline{\overline{b}} \cdot e && \text{(see below)} \\
&= \textbf{if } \overline{b} \textbf{ then } f \textbf{ else } e && \text{(by definition)}
\end{aligned}$$

Here, we used that $\overline{\overline{b}} \equiv b$, something that you will prove in the homework. □

Let's introduce one more shorthand in our notation. For tests b and programs e, we use "**if** b **then** e" for $b \cdot e + \overline{b}$. Note how this expression intuitively matches your understanding of the **if-then** construct: if b holds, then the $b \cdot e$ branch is executed; otherwise, $\overline{b}$ holds, and so the $\overline{b}$ branch is executed (but the state is not changed). Using this shorthand, we can state the following:

Lemma 2.12. *Let $e \in \mathbb{G}$ and $b \in \mathbb{B}$. The following holds:*

$$\textbf{while } b \textbf{ do } e \equiv \textbf{if } b \textbf{ then } (e \cdot \textbf{while } b \textbf{ do } e)$$

Proof. We again unfold the syntactic sugar and apply our rules, as follows:

$$\begin{aligned}
\textbf{while } b \textbf{ do } e &= (b \cdot e)^* \cdot \overline{b} && \text{(by definition)} \\
&\equiv (b \cdot e \cdot (b \cdot e)^* + 1) \cdot \overline{b} && \text{(unrolling)} \\
&\equiv b \cdot e \cdot (b \cdot e)^* \cdot \overline{b} + 1 \cdot \overline{b} && \text{(distributivity)} \\
&\equiv b \cdot e \cdot (b \cdot e)^* \cdot \overline{b} + \overline{b} && \text{(unit)} \\
&= \textbf{if } b \textbf{ then } (e \cdot (b \cdot e)^* \cdot \overline{b}) && \text{(by definition)} \\
&= \textbf{if } b \textbf{ then } (e \cdot \textbf{while } b \textbf{ do } e) && \text{(by definition)}
\end{aligned}$$

□

7 Homework

1. Let's practice Boolean equivalences. In the following, let $b, c \in \mathbb{B}$.

   (a) Use Lemma 2.6 to prove that $\overline{\overline{b}} \equiv b$.

   (b) Prove DeMorgan's second law: $\overline{b \cdot c} = \overline{b} + \overline{c}$.

   (c) Our rules are somewhat redundant, in that some of the rules can be proved from the *other* rules. Show that this is the case for $b + b \equiv b$.

   (d) *(Optional)* Prove that $b + (\overline{b} + c) \equiv 1$ *without using associativity.*

2. On to practice guarded rational equivalence. Let $b, c \in \mathbb{B}$ and $e, f, g \in \mathbb{G}$.

   (a) Prove that the **if-then-else** construct is "skew-associative", i.e.:

   $$\begin{aligned} &\textbf{if } b \textbf{ then } (\textbf{if } c \textbf{ then } e \textbf{ else } f) \textbf{ else } g \\ &\quad \equiv \textbf{if } b \cdot c \textbf{ then } e \textbf{ else } (\textbf{if } b \textbf{ then } f \textbf{ else } g) \end{aligned}$$

   Hint 1: It's easier to start your derivation with the second program.
   Hint 2: You may use the facts proved in the previous exercise.

   (b) Suppose $\textbf{if } b \textbf{ then } e \cdot f \textbf{ else } g \leqq f$. Show that $(\textbf{while } b \textbf{ do } e) \cdot g \leqq f$.
   Hint: You need to use second-to-last rule from Definition 2.9.

3. *Hoare logic* is a formalism for reasoning about the correctness of programs. Its statements are *triples* denoted $\{P\}C\{Q\}$, where P and Q are logical formulas, and C is a program. Such a triple *holds* when, if a machine starts in a state satisfying P, it reaches a state satisfying Q after executing C.

   We can encode Hoare logic using guarded rational expressions: writing

   $$\{b\}e\{c\} \qquad \text{as a shorthand for} \qquad b \cdot e \cdot \overline{c} \equiv 0$$

   The idea is that, if $b \cdot e \cdot \overline{c} \equiv 0$, then there is no valid way to execute $b \cdot e \cdot \overline{c}$, i.e., to assert that b holds, execute e, and then assert that $\overline{c}$ holds. Thus, necessarily, if e runs to completion after asserting b, then c holds.

   In this exercise, you will verify that the rules of inference in Hoare logic are actually compatible with this encoding in guarded rational expressions.

   (a) Hoare logic contains the following rule for sequential composition:

   $$\frac{\{P\}C_1\{Q\} \qquad \{Q\}C_2\{R\}}{\{P\}C_1; C_2\{R\}}$$

   Verify the compatibility of this rule: given $b, c, d \in \mathbb{B}$ and $e, f \in \mathbb{G}$ such that $\{b\}e\{c\}$ and $\{c\}f\{d\}$ hold, prove that $\{b\}e \cdot f\{d\}$ holds.

   (b) *(Optional)* The following rule governs **while-do** constructs.

   $$\frac{\{P \wedge Q\}C\{P\}}{\{P\}\textbf{while } Q \textbf{ do } C\{\neg Q \wedge P\}}$$

   Verify this rule as well: given $b, c \in \mathbb{B}$ and $e \in \mathbb{G}$ such that $\{b \cdot c\}e\{b\}$ holds, prove that $\{b\}\textbf{while } c \textbf{ do } e\{\overline{c} \cdot b\}$ holds.
   Hint 1: first argue that $b \cdot (c \cdot e)^* \leqq (c \cdot e)^* \cdot b$.
   Hint 2: remember that if $e \leqq e'$*, then* $e \cdot f \leqq e' \cdot f$.

8 Bibliographical notes

Kleene Algebra with Tests as such was first introduced by Kozen [Koz96], where the encoding of traditional imperative programs also appears. Relevant follow-up work includes [KS96, Koz97], although we will address those ideas in a later lecture. The connection with Hoare logic was explored by Kozen [Koz00].

Boolean algebra comes from George Boole [Boo47, Boo54]. Whitehead proposed the first axiomatization [Whi98]; his axioms are almost the same as the ones above, except that Whitehead's absorbtion laws are replaced by the idempotence law. A concise set of laws for Boolean algebra is due to Huntington [Hun04]. De Morgan is credited with the well-known inversion laws [DM47], though they appear to have been known since Aristotle, if not before.

References

[Boo47] George Boole. *The Mathematical Analysis of Logic: Being an Essay Towards a Calculus of Deductive Reasoning*. MacMillan, Barclay & MacMillan, London, 1847. URL: `https://www.gutenberg.org/ebooks/36884`.

[Boo54] George Boole. *An investigation of the laws of thought: on which are founded the mathematical theories of logic and probabilities*. Walton and Maberly, London, 1854. URL: `https://www.gutenberg.org/ebooks/15114`.

[DM47] Augustus De Morgan. *Formal logic: or, the calculus of inference, necessary and probable*. Taylor and Walton, London, 1847. URL: `https://archive.org/details/formallogicorthe00demouoft`.

[Hun04] Edward V. Huntington. Sets of independent postulates for the algebra of logic. *Trans. Am. Math. Soc.*, 5(3):288–309, 1904. `doi:10.1090/s0002-9947-1904-1500675-4`.

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

[Koz97] Dexter Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, 1997. `doi:10.1145/256167.256195`.

[Koz00] Dexter Kozen. On Hoare logic and Kleene algebra with tests. *ACM Trans. Comput. Log.*, 1(1):60–76, 2000. `doi:10.1145/343369.343378`.

[KS96] Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In *CSL*, pages 244–259, 1996. `doi:10.1007/3-540-63172-0_43`.

[Whi98] Alfred North Whitehead. *A Treatise on Universal Algebra, with Applications*. Cambridge University Press, Cambridge, 1898. URL: `https://archive.org/details/atreatiseonuniv00goog`.