Foundations and Trends® in Theoretical Computer Science

Kleene Algebra

Suggested Citation: Tobias Kappé, Alexandra Silva and Jana Wagemaker (xxxx), "Kleene Algebra", Foundations and Trends® in Theoretical Computer Science: Vol. xx, No. xx, pp 1–18. DOI: 10.1561/XXXXXXXXX.

now
the essence of knowledge
Boston — Delft

Contents

1 Introduction

Multiplication *distributes* over addition: if x, y and z are numbers, then $x \cdot (y + z)$ is *equivalent* to $x \cdot y + x \cdot z$. Using similar laws, we can reason about equivalence of arithmetic expressions by manipulating their syntax. Kleene Algebra is motivated by the idea that these principles are not limited to numbers — indeed, in the words of Hoare et al. [1]:

> *programs are mathematical expressions [. . .] subject to a set of laws as rich and elegant as any other branch of mathematics, engineering, or natural science.*

By studying these laws, we can gain theoretical insights and practical tools to reason about program correctness. Applications include:

1. When rearranging code to improve readability, we want to be certain that the new program has the same semantics, or at the very least, does not introduce any new behavior.
2. We want to check whether our implementation of an algorithm conforms to its (abstract) specification, or at least refines it.
3. A compiler typically transforms its code, for instance to conform to hardware constraints, and we want to check that these transformations never change the meaning of the program.

To illustrate the first case a bit more, consider the programs below, written in a standard imperative programming language.

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

Here, the terms a and b stand in for any Boolean expressions (tests), while e and f represent general programs. As you can see, this program is somewhat messy: it includes two occurrences of the sub-program e, and no fewer than *three* occurrences of Boolean expression a!

Clearly, this code could benefit from some refactoring. After staring at it for a while, you realise that e runs only when a and b hold, while f is executed when a is true and b is false. Furthermore, if a is false after running e or f, the program will stop; otherwise, it will continue with (some) loop. This eventually leads you to refactor the code as follows:

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

At this point, you may have some intuition of why these two programs are the same, or perhaps you are still skeptical. By the end of this booklet, you will be able to *prove* that these two programs have the same behavior, for any choice of e, f, a and b.

In Chapter 2, we will go over the basic theory of Kleene Algebra. Chapter 3 follows this up with some basic material on *coalgebra*, which we will be using as a framework throughout this booklet. Next, in Chapter 4, we instantiate coalgebra to study automata, and their connection to Kleene Algebra. This is followed by a proof of one of the most important results in Kleene Algebra, namely the *completeness theorem*, in Chapter 5. Chapter 6 extends the preceding material by introducing *Kleene Algebra with Tests*, a variant of Kleene Algebra that

$$\begin{tikzcd}
 & 4.1 \arrow[r] & 4.3 \arrow[dr, bend right] \arrow[drr, bend left] & & & \\
2 \arrow[ur, bend left] \arrow[dr, bend right] & & & 5.2 \arrow[r] & 6 \arrow[r] & 7 \\
 & 3 \arrow[r] & 4 \arrow[r] & 5 \arrow[ur, bend right] & &
\end{tikzcd}$$

Figure 1.1: Possible paths through this booklet.

can be used to reason about programs such as the ones on the previous page. We conclude in Chapter 7 with an overview of the source material and other literature that may be of interest to the reader.

1.1 Source material

The text of this booklet includes excerpts of the authors' PhD theses [2], [3], [4] and lecture notes developed for a master-level course that one author taught at the University of Amsterdam and at ESSLLI 2023. Results and definitions due to others will be annotated with citations.

1.2 Reading guide

There are three possible paths through these notes, drawn in Figure 1.1. If you are teaching a graduate-level course, you could consider following the blue path, and covering all material. For a shorter (possibly undergraduate) course it might be more appropriate to skip the completeness results and the corresponding coalgebraic theory, and instead focus on the automata theory and soundness. This is captured by the red paths.

1.3 Notation

We use $\mathbb{N}$ to denote the set of natural numbers.

Relations A *relation* between two sets A and B is a subset of $A \times B$. We often use infix notation to denote membership of a relation: for a relation $R \subseteq A \times B$, we can write aRb to denote $(a, b) \in R$. If R is a relation on a set A (so a subset of $A \times A$), we call the set

$\mathsf{id}_A = \{(a,a) \mid a \in A\}$ the *identity relation.* If R is a relation between sets A and B and R' a relation between sets B and C, we write $R \circ R'$ for the *relational composition* of R and R', that is: $R \circ R' = \{(a,c) \mid \exists b \in B \text{ such that } (a,b) \in R \wedge (b,c) \in R'\}$.

Equivalence relations A relation $R \subseteq A \times A$ is said to be *reflexive* when for all $a \in A$ we have aRa, *symmetric* when for all $a, b \in A$, aRb implies bRa, and *transitive* when aRb and bRc imply aRc for all $a, b, c \in R$. Given $R \subseteq A \times A$, we write R^* for the *reflexive-transitive closure* of R, which is the smallest reflexive and transitive relation on A that contains R; such a relation is guaranteed to exist. If $R \subseteq A \times A$ is reflexive, symmetric and transitive it is called an *equivalence relation.* Given $a \in A$, we write $[a]$ for the *equivalence class* of A under R (the relation R will be clear from context), which is the set $[a] = \{b \in A \mid aRb\}$. We denote $A/R = \{[a] \mid a \in A\}$ for the *quotient* of A by R, i.e., the set of equivalence classes under R. Furthermore, we write $\equiv_R$ for the *equivalence closure* of R, which is the smallest equivalence relation containing R.

Congruence relations Let R be an equivalence relation defined on a set of expressions A. We call R a *congruence relation* if it is compatible with the operators in the expressions that make up A. For instance, if $+$ is such an operator, and we know that eRf and gRh, then R being a congruence tells us that $e + g \mathrel{R} f + h$.

Singletons and disjoint unions We write 1 for the set containing the symbol $\{*\}$. Given two sets A and B, we write $A + B$ for their *disjoint union.* Formally, this is $\{\langle a, 0\rangle \mid a \in A\} \cup \{\langle b, 1\rangle \mid b \in B\}$; we will elide this detail, and simply treat elements of A or B as elements of $A + B$, and elements of $A + B$ as either elements of A or B (but not both). Overlap between A and B can be avoided by renaming elements.

Functions and powersets A *function* f from sets A to B, denoted $f \colon A \to B$, assigns to each element of A exactly one element of B, written $f(a)$. We write B^A for the set of functions from A to B, and id_A for the *identity* function on A, which assigns each element to itself.

We use 2 as an abbreviation for $\{0, 1\}$. In particular, this means that 2^A, the set of functions from A to $\{0, 1\}$, is in one-to-one correspondence with the powerset of A. We will therefore simply use 2^A to denote the powerset of A, and treat its members as subsets of A or as their characteristic functions depending on the context.

2

Basics

In this chapter, we develop a basic algebraic theory of regular expressions, known as Kleene Algebra. If you have a background in Computer Science, you may have encountered these expressions in undergraduate computer science courses with titles such as “Formal Languages”, “Theory of Computation” or “Foundations of Computer Science”. You may also be familiar with them as a tool for finding patterns of characters in text using tools like GNU Grep or the Perl Compatible Regular Expressions (`pcre`) functionality available in many programming languages today. At their core, regular expressions provide a way to describe *patterns of events*, whether those are validation conditions or packet forwarding behavior in a network. More broadly, we can think of regular expressions as an abstract syntax for imperative programs, which is the main motivation for studying them in this book.

We will start by defining regular expressions, along with two semantics, both of which have an intuitive meaning in connection to programming languages. We then go on to introduce the laws of Kleene Algebra, which can be used to reason about *equivalence* of regular expressions, in the sense that they have the same semantics.

2.1 Syntax and Semantics

Throughout this booklet, we fix a finite set of actions $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \dots\}$. You could think of these actions as representing primitive operations that can be performed by a machine; for example, the action symbol $\mathtt{a}$ could mean "make the red LED blink once" or "verify that the next input letter is $\mathtt{a}$". Usually, we will abstract from the exact meaning of the actions, preferring to focus on when and how often they occur, and in what order. Regular expressions are built from these actions.

Definition 2.1 (Expressions). Regular expressions (over an alphabet A) are generated by the grammar:

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

where $\mathtt{a} \in A$. We will write Exp for the set of regular expressions.

When we write regular expressions, we may elide "$\cdot$" — just like how multiplication is commmonly omitted. We also adopt the convention that * takes precedence over $\cdot$, which in turn takes precedence over $+$. Thus, $(\mathtt{a} \cdot (\mathtt{b}^*)) + \mathtt{c}$ can also be written as $\mathtt{ab}^* + \mathtt{c}$. Parentheses can also be used to disambiguate expressions the same way this is done in other contexts (even though they are not part of the syntax).

In regular expressions, the constant 0 represents a program that crashes immediately, and 1 represents a program that succeeds immediately. The "union" or "non-deterministic choice" operator $+$ is commonly interpreted as taking the union of possible behaviors — i.e., the expression $e + f$ represents all patterns of events described by either e or f. Furthermore, the "concatenation" or "sequential composition" operator $\cdot$ combines behaviors in sequence — i.e., ef describes all behaviors achieved by first completing a behavior from e, and then a behavior from f. Finally, the "iteration" or "Kleene star" operator * represents behavior achieved by running an arbitrary (possibly zero) number of patterns of events from e in sequence.

Example 2.2.

1. The regular expression $\mathtt{a} + \mathtt{bc}$ denotes the behavior where either the action $\mathtt{a}$ runs, *or* the behavior where $\mathtt{b}$ is followed by $\mathtt{c}$.

2. We can also write the regular expression $\mathtt{ab}^*\mathtt{c}$ which describes all sequences of events that start with $\mathtt{a}$ and are followed by some (possibly zero) number of $\mathtt{b}$'s, ending with the action $\mathtt{c}$.
3. Finally, $(\mathtt{a}+\mathtt{b})^*$ is a regular expression that denotes all behaviors that consist of a finite number of actions, each of which is either the event $\mathtt{a}$ or the event $\mathtt{b}$.

Remark 2.3. Some sources may use the term *rational expression* to refer to regular expressions as defined above. This is a historic distinction that is meant to separate behaviors that can be denoted by expressions (rational) versus behaviors that can be described by an automaton (regular). As it happens, these classes of behaviors are the same in all settings that we encounter in this booklet; we therefore do not make this distinction in our nomenclature.

2.1.1 Relational Semantics

One way to assign a meaning to regular expressions is to consider the effects that each action may have on the outside world. This naturally leads to an interpretation of regular expressions in terms of *relations*, which relate each "state" of the world to the possible states that may result from following a behavior of the expression. To do this we are going to need to know which states are possible, and how each primitive action may change them. This is encapsulated as follows.

Definition 2.4 (Interpretation)**.** An *interpretation* (of the alphabet A) is a pair $\langle S, \sigma \rangle$, where S is a set and σ is a function from A to relations on S, i.e., $\sigma\colon A \to 2^{S \times S}$. We often denote an interpretation simply by σ.

When we fix an interpretation, the relational semantics is fairly easy to derive inductively. For instance, if a state s can be transformed by e into s', and s' can be transformed by f into s'', then ef can transform s into s''; it follows that the relation denoted by ef is obtained by composing the relations denoted by e and f. This leads to the following (parameterized) semantics.

Definition 2.5 (Relational semantics)**.** Given an interpretation of the alphabet $\sigma\colon A \to 2^{S \times S}$, we define the σ-semantics $[\![-]\!]_\sigma\colon \mathsf{Exp} \to 2^{S \times S}$

inductively, as follows:

$$\begin{array}{ccc} [\![0]\!]_\sigma = \emptyset & [\![1]\!]_\sigma = \mathsf{id}_S & [\![\mathsf{a}]\!]_\sigma = \sigma(\mathsf{a}) \\ [\![e+f]\!]_\sigma = [\![e]\!]_\sigma \cup [\![f]\!]_\sigma & [\![e \cdot f]\!]_\sigma = [\![e]\!]_\sigma \circ [\![f]\!]_\sigma & [\![e^*]\!]_\sigma = [\![e]\!]_\sigma^* \end{array}$$

In the last case, we use * to denote both reflexive-transitive closure (on the right) and the Kleene star of regular expressions (on the left)

Example 2.6. Suppose you want to calculate the *integer square root* of $n \in \mathbb{N}$ — in other words, the largest number i such that $i^2 \leq n$. One way to arrive at this number is to start i at zero, and keep incrementing it while $(i+1)^2 \leq n$. A traditional program to achieve this could be:

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

The behavior of this program can be captured by this regular expression:

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

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

- `init` sets the variable i to zero.
- `guard` observes that $(i+1)^2 \leq n$.
- `incr` increments the value of i.
- `validate` observes that $(i+1)^2 > n$.

It may seem odd that some actions simply observe a property of a variable, without changing any value. But it does make sense to include them — after all, if $(i+1)^2 \leq n$ does not hold, then the program will not start a new iteration of the loop. Hence, the machine must at some point perform this check, which we model as an action. We will study such *tests* in more detail in Chapter 6.

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 $s\colon \{i, n\} \to \mathbb{N}$. Next, we can fix our interpretation of the primitive programs, as follows:

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

In the above, given a function $s\colon \{i, n\} \to \mathbb{N}$, $n \in \mathbb{N}$ and $v \in \{i, n\}$, we write $s[n/v]$ for the function that overrides the value of v to n in s, i.e.:

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

If you compute $[\![e]\!]_\sigma$, you will find that it relates functions $s\colon \{i, n\} \to \mathbb{N}$ to functions $s'\colon \{i, n\} \to \mathbb{N}$ where $s'(n) = s(n)$ and $s'(i)^2 \leq s(n) \leq (s'(i)+1)^2$. We will return to this program later in Exercise 2.1.

In the example above, you may note that the interpretations of `guard` and `validate` are relations that are not *total*, i.e., there exist $s \in S$ that do not admit an $s' \in S$ with $s\ \sigma(\texttt{guard})\ s'$, and similarly for $\sigma(\texttt{validate})$. This is intentional: the intended meaning is that there is no valid way to continue a computation with `guard` for these states, and so the outcome is discarded. Usually, this means that some non-deterministic branch of execution does not get to contribute to the overall results of the program. Another branch may still yield a result, although regular expressions and interpretations must be written carefully to guarantee this, as in the following example.

Example 2.7. One common pattern is to simulate an *if-then-else* construct using non-determinism and opposite tests. For instance, suppose we want to model the following piece of code using a regular expression, where `a` and `b` are placeholders for the code contained in each branch:

```
if $(i+1)^2 \leq n$ then
    a;
else
    b;
```

One way to accomplish this is to reuse `guard` and `validate` from the previous example and write $\texttt{guard} \cdot \texttt{a} + \texttt{validate} \cdot \texttt{b}$. Under the interpretation σ used earlier, the semantics will be a relation that applies `a` to states where $(i+1)^2 \leq n$, and `b` to states where $(i+1)^2 > n$ — i.e.,

$$\begin{aligned}\sigma(\texttt{guard} \cdot \texttt{a} + \texttt{validate} \cdot \texttt{b}) = {} & \{\langle s, t\rangle \in \sigma(\texttt{a}) \mid (s(i)+1)^2 \leq s(n)\} \cup \\ & \{\langle s, t\rangle \in \sigma(\texttt{b}) \mid (s(i)+1)^2 > s(n)\}\end{aligned}$$

We will discuss a variant of Kleene algebra in Chapter 6 that has this kind of composition as a first-class citizen.

2.1.2 Language semantics

The relational semantics is a very versatile tool for modeling programs with regular expressions, but it hinges on the interpretation for each primitive action symbol, as well as a set of states that may be affected by the actions. This makes it possible to reason about particular programs, but equivalence of programs in general (i.e., regardless of the interpretation) is harder to describe. This is where the *language semantics* of regular expressions comes in. Put simply, this model assigns to each expression the possible sequences of actions that may result from running the associated program. These *traces* are recorded purely using the action names, which means that this semantics is not parameterized by an interpretation; we can focus purely on the patterns of events.

To properly introduce this semantics, we first discuss some notation. A *word* or *trace* (over A) is a finite sequence of actions (from A). For instance, if $A = \{\texttt{a}, \texttt{b}\}$, then `abaa` is a word. Words can be *concatenated* by juxtaposition, i.e., if $w = \texttt{ab}$ and $x = \texttt{ba}$, then $wx = \texttt{abba}$. We write ϵ for the *empty* word, and note that it is neutral for concatenation on both sides, i.e., $w\epsilon = w = \epsilon w$ for all words w.

A *language* (over A) is a set of words (over A). When L and K are languages, we write $L \cdot K$ for the *concatenation* of L and K, i.e., the language $\{wx \mid w \in L, x \in K\}$, and L^* for the *Kleene star* of L, i.e., the language $\{w_1 \cdots w_n \mid w_1, \ldots, w_n \in L\}$. This makes A^* the set of all words over A. The Kleene star and multiplication operators are thus overloaded to also act on languages.

With this in hand, we are now ready to define the language semantics of regular expressions.

Definition 2.8 (Language semantics)**.** We define a map $[\![-]\!] \colon \mathsf{Exp} \to 2^{A^*}$ by induction as follows:

$$[\![0]\!] = \emptyset \qquad [\![1]\!] = \{\epsilon\} \qquad [\![\mathtt{a}]\!] = \{\mathtt{a}\}$$
$$[\![e + f]\!] = [\![e]\!] \cup [\![f]\!] \qquad [\![ef]\!] = [\![e]\!] \cdot [\![f]\!] \qquad [\![e^*]\!] = [\![e]\!]^*$$

Example 2.9. Let's calculate the languages of the regular expressions from Example 2.2:

$$[\![\mathtt{a} + \mathtt{bc}]\!] = \{\mathtt{a}, \mathtt{bc}\} \qquad [\![\mathtt{ab}^*\mathtt{c}]\!] = \{\mathtt{ac}, \mathtt{abc}, \mathtt{abbc}, \ldots\}$$

$$[\![(\mathtt{a} + \mathtt{b})^*]\!] = \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \ldots\}$$

From the programming perspective, the language of a regular expression is really an overapproximation of the behavior that may occur — an interpretation may give rise to a wide variety of subsets of the sequences of actions in the language. For instance, in the regular expression $\mathtt{a} + \mathtt{bc}$ the action $\mathtt{b}$ may halt the program, preventing $\mathtt{c}$ from being run. The important point, however, is that other interpretations may disagree, and so $\mathtt{bc}$ is included in the language semantics.

The language semantics is closely connected to the relational semantics, in the sense that for any $e \in \mathsf{Exp}$ and any interpretation $\langle S, \sigma \rangle$, we can reconstruct $[\![e]\!]_\sigma$ from $[\![e]\!]$, by way of the function $\hat{\sigma} \colon 2^{A^*} \to 2^{S \times S}$, which accumulates the pairs of the relations represented by each word:

$$\hat{\sigma}(L) = \bigcup \{\sigma(\mathtt{a}_1) \circ \cdots \circ \sigma(\mathtt{a}_n) \mid \mathtt{a}_1 \cdots \mathtt{a}_n \in L\}$$

Lemma 2.10. For each $e \in \mathsf{Exp}$ and interpretation $\langle S, \sigma \rangle$, it holds that $[\![e]\!]_\sigma = \hat{\sigma}([\![e]\!])$.

Proof sketch. The proof proceeds by induction on e, and relies on the fact that $\hat{\sigma}$ is compatible with the operators of language composition. Specifically, for all $L, K \subseteq A^*$, the following hold:

$$\hat{\sigma}(L \cup K) = \hat{\sigma}(L) \cup \hat{\sigma}(K) \qquad \hat{\sigma}(L \cdot K) = \hat{\sigma}(L) \circ \hat{\sigma}(K) \qquad \hat{\sigma}(L^*) = \hat{\sigma}(L)^*$$

The details are left as Exercise 2.10. □

A converse correspondence is also possible, but slightly more tricky to figure out. The idea here is to craft a particular interpretation $\langle A^*, \sharp \rangle$ that represents languages as relations, by defining

$$\sharp(\mathtt{a}) = \{\langle w, w\mathtt{a} \rangle \mid w \in A^*\}$$

We can then relate the relational semantics under this interpretation to languages, as follows.

Lemma 2.11. Let $e \in \mathsf{Exp}$. For all $w, x \in A^*$, we have that $\langle w, wx \rangle \in [\![e]\!]_\sharp$ if and only if $x \in [\![e]\!]$.

Proof sketch. This comes down to showing that $[\![e]\!]_\sharp = \{\langle w, wx \rangle \mid w \in A^*, x \in [\![e]\!]\}$. The proof again proceeds by induction on e; the details are left as Exercise 2.11. □

Together, these last two lemmas tell us that agreement under the language semantics is equivalent to agreement under *any* relational interpretation. This means that the language semantics fulfills its intention of abstracting over the meaning of primitive actions.

Theorem 2.12. Let $e, f \in \mathsf{Exp}$. It holds that $[\![e]\!] = [\![f]\!]$ iff for all interpretations $\langle S, \sigma \rangle$, $[\![e]\!]_\sigma = [\![f]\!]_\sigma$.

In light of the above, we will use "the semantics of regular expressions" to refer to the language semantics and the (universally quantified) relational semantics interchangeably in the sequel. After all, we are mostly interested in axiomatizing the equivalence induced by the semantics, and we know that both perspectives equate the same expressions.

2.2 Reasoning

Writing a program as a regular expression is a nice exercise, but not an end in itself. We want to use this representation of the program to reason about whether or not a program encoded in this way is equivalent to another program. In this section, we get to the essence of Kleene Algebra: the laws that underpin equivalence of regular expressions. If you favor the relational semantics, these are the pairs of regular expressions that yield the same expression under any interpretation. Alternatively, under the language semantics, it can be seen as the study of which expressions denote the same patterns of events.

As a very basic example, take any $e, f \in \mathsf{Exp}$. Now $e + f$ has the same semantics as $f + e$. After all, we can calculate as follows:

$$\llbracket e + f \rrbracket = \llbracket e \rrbracket \cup \llbracket f \rrbracket = \llbracket f \rrbracket \cup \llbracket e \rrbracket = \llbracket f + e \rrbracket$$

In other words, the operator $+$ is *commutative*. There are many correspondences like these, and we can use them when we want to prove properties of regular expressions. Here are a number of useful ones.

Lemma 2.13. The following hold for all $e, f, g \in \mathsf{Exp}$:

$$\llbracket e + 0 \rrbracket = \llbracket e \rrbracket \qquad \llbracket e + e \rrbracket = \llbracket e \rrbracket \qquad \llbracket e + f \rrbracket = \llbracket f + e \rrbracket$$

$$\llbracket e + (f + g) \rrbracket = \llbracket (e + f) + g \rrbracket \qquad \llbracket e \cdot (f \cdot g) \rrbracket = \llbracket (e \cdot f) \cdot g \rrbracket$$

$$\llbracket e \cdot (f + g) \rrbracket = \llbracket e \cdot f + e \cdot g \rrbracket \qquad \llbracket (e + f) \cdot g \rrbracket = \llbracket e \cdot g + f \cdot g \rrbracket$$

$$\llbracket e \cdot 1 \rrbracket = \llbracket e \rrbracket = \llbracket 1 \cdot e \rrbracket \qquad \llbracket e \cdot 0 \rrbracket = \llbracket 0 \rrbracket = \llbracket 0 \cdot e \rrbracket$$

$$\llbracket 1 + e \cdot e^* \rrbracket = \llbracket e^* \rrbracket = \llbracket 1 + e^* \cdot e \rrbracket$$

Proof sketch. Most proofs are similar to that of the third property, shown above. We highlight a particular case in Exercise 2.3. □

Most of these properties are fairly easy to understand intuitively. For instance, the rule $e + e = e$ says that a non-deterministic choice between a sequence of events from 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 $e + (f + g) = (e + f) + g$ — the operator $+$ is *associative*.

Example 2.14. We can use the laws from Lemma 2.13 to reason about equivalences. For instance, to show that $\llbracket 0^* \rrbracket = \llbracket 1 \rrbracket$, we can derive:

$$\begin{aligned}
\llbracket 0^* \rrbracket &= \llbracket 1 + 0 \cdot 0^* \rrbracket && (\llbracket e^* \rrbracket = \llbracket 1 + e \cdot e^* \rrbracket) \\
&= \llbracket 1 \rrbracket + \llbracket 0 \cdot 0^* \rrbracket && (\text{def. } \llbracket - \rrbracket) \\
&= \llbracket 1 \rrbracket + \llbracket 0 \rrbracket && (\llbracket 0 \cdot e \rrbracket = \llbracket 0 \rrbracket) \\
&= \llbracket 1 + 0 \rrbracket && (\text{def. } \llbracket - \rrbracket) \\
&= \llbracket 1 \rrbracket && (\llbracket e + 0 \rrbracket = \llbracket e \rrbracket)
\end{aligned}$$

Some properties of the Kleene star are conditional, i.e., they require proving some other property first. We list two particularly useful ones.

Lemma 2.15. The following hold for all $e, f, g \in \mathsf{Exp}$:

1. If $[\![e + f \cdot g]\!] \subseteq [\![g]\!]$ then $[\![f^* \cdot e]\!] \subseteq [\![g]\!]$.
2. If $[\![e + f \cdot g]\!] \subseteq [\![f]\!]$ then $[\![e \cdot g^*]\!] \subseteq [\![f]\!]$.

Intuitively, the first law above can be read as saying that, if the behavior of g includes both the behaviors of $f \cdot g$ and e, then any sequence of events described by g (including, in particular, the behavior of e) can be prepended with a sequence of events from f to obtain a new sequence of events also allowed by g. Since this can be repeated any number of times, it follows that the behavior of $f^* \cdot e$ must also be included in that of g; the second law can be explained similarly.

As it turns out, the properties of $[\![-]\!]$ that we have seen thus far are so useful, it pays off to isolate them into a *reasoning system*, as follows.

Definition 2.16 (Provable equivalence)**.** We define $\equiv$ as the smallest congruence on Exp that satisfies the following for all $e, f, g \in \mathsf{Exp}$:

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

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

$$(e + f) \cdot g \equiv e \cdot g + f \cdot g \qquad e \cdot 1 \equiv e \qquad 1 \cdot e \equiv e \qquad e \cdot 0 \equiv 0 \qquad 0 \cdot e \equiv 0$$

$$1 + e \cdot e^* \equiv e^* \qquad 1 + e^* \cdot e \equiv e^* \qquad \frac{e + f \cdot g \leqq g}{f^* \cdot e \leqq g} \qquad \frac{e + f \cdot g \leqq f}{e \cdot g^* \leqq f}$$

where we write $e \leqq f$ as a shorthand for $e + f \equiv f$.

Intuitively, the way we defined $\equiv$ means that if $e \equiv f$ holds for some $e, f \in \mathsf{Exp}$, then we can prove $[\![e]\!] = [\![f]\!]$ by using only the properties in Lemmas 2.13 and 2.15, as well as the definition of $[\![-]\!]$.

Lemma 2.17 (Soundness)**.** Let $e, f \in \mathsf{Exp}$. If $e \equiv f$, then $[\![e]\!] = [\![f]\!]$.

Lemma 2.17 is an instance of a more general correspondence between the laws in Definition 2.16 and the notion of Kleene algebra. To see this, we formally define "a Kleene algebra" using exactly those laws:

Definition 2.18 (Kleene algebra)**.** A Kleene algebra $\mathcal{K} = (X, +, \cdot, -^*, 0, 1)$ consists of a set X with two distinguished elements 0 and 1, two binary operations $+$ and $\cdot$ (usually omitted when writing the expressions) and a unary operation $-^*$ satisfying the laws in Definition 2.16 (*e.g.* $e + f \equiv f + e$, hence for all $x, y \in X$ $x + y = y + x$).

Every Kleene algebra induces a partial order $\leq$ derived from the $+$ operator in the same way that $\leqq$ is, i.e., $x \leq y$ if and only if $x + y = y$.

Proving Lemma 2.17 essentially amounts to proving that

$$(2^{A^*}, \cup, \cdot, -^*, \emptyset, \{\epsilon\})$$

is a Kleene algebra. In this model, $\leq$ is just set inclusion. Another well-known Kleene algebra is the family of all binary relations over a set X with the empty relation for 0, the identity relation for 1, union for $+$, relational composition for $\cdot$ and reflexive-transitive closure for $-^*$.

Returning to the laws in Definition 2.16, the first group of rules summarizes the properties of $+$, and essentially says that $\mathcal{K}$ is a join-semilattice. Together with the second group, which contains the properties of $\cdot$ and its interaction with $+$, it states that $\mathcal{K}$ is an idempotent semiring. The last rules axiomatize the star operator.

As suggested, $\leqq$ is a partial order up to $\equiv$. Moreover, the operators are monotone with respect to $\leqq$. We record these properties below; their proofs are left to Exercise 2.4.

Lemma 2.19. Let $e, f, g \in \mathsf{Exp}$. The following properties hold:

1. If $e \equiv f$ then $e \leqq f$.
2. If $e \leqq f$ and $f \leqq e$, then $e \equiv f$.
3. If $e \leqq f$ and $f \leqq g$, then $e \leqq g$.
4. If $e \leqq f$, then $e + g \leqq f + g$ and $g + e \leqq g + f$.
5. If $e \leqq f$, then $e \cdot g \leqq f \cdot g$ and $g \cdot e \leqq g \cdot f$.
6. If $e \leqq f$, then $e^* \leqq f^*$.

Remark 2.20. Some authors make $\leqq$ the primary object of study, and define $e \equiv f$ as a shorthand for "$e \leqq f$ and $f \leqq e$". This is a very natural way to present Kleene Algebra, which can simplify a number of proofs. The interdefinability of these relations as well as Lemma 2.19 allows us to switch between these perspectives at will.

We can now use $\equiv$ and $\leqq$ to reason cleanly about equivalence. We will use the associativity of $+$ and $\cdot$ as an excuse to abuse notation, and drop some parentheses, writing $e \cdot f \cdot g$ to mean either $e \cdot (f \cdot g)$ or $(e \cdot f) \cdot g$ (and similarly for $+$). Essentially, these conventions mean that we can employ the usual style of equational reasoning.

Example 2.21. Returning to Example 2.14, to prove that $[\![0^*]\!] = [\![1]\!]$, we can use Lemma 2.17 to instead prove that $0^* \equiv 1$, as follows.

$$\begin{aligned} 0^* &\equiv 1 + 0 \cdot 0^* && (e^* \equiv 1 + e \cdot e^*) \\ &\equiv 1 + 0 && (0 \cdot e \equiv 0, \text{ congruence}) \\ &\equiv 1 && (e + 0 \equiv e) \end{aligned}$$

Note how the derivation is less cluttered than the one in Example 2.14.

The only unfamiliar feature of reasoning using $\equiv$ comes from the last two laws in Definition 2.16, which we will refer to as the *left* and *right fixpoint* rule respectively, or simply as the *star axioms*. We can use these by either assuming that the conclusion holds and using that to prove the claim, followed by a proof of the premise (backward reasoning), or proving the premise and then using the consequence later (forward reasoning). Here is an example of the backwards reasoning style.

Lemma 2.22. The following equalities hold in Kleene algebras:

$$e^* \equiv e^* + 1 \qquad\qquad e^* e^* \equiv e^* \qquad\qquad e^{**} \equiv e^*$$

Proof. The first equality follows easily:

$$e^* \equiv 1 + ee^* \equiv 1 + 1 + ee^* \equiv 1 + e^* \equiv e^* + 1$$

For the second equality, we have:

$$e^* e^* \equiv (1 + ee^*)e^* \equiv e^* + ee^*e^* \geqq e^*$$

Furthermore, to prove that $e^*e^* \leqq e^*$, it suffices to show that $ee^* + e^* \leqq 1$ (by the left fixpoint rule). To this end, we derive that:

$$ee^* + e^* \leqq 1 + ee^* + e^* \equiv e^* + e^* \equiv e^*$$

For the third equality, we start by proving that $e^{**} \leqq e^*$. By the left fixpoint rule, it suffices to show that $e^*e^* + 1 \leqq e^*$. This follows from the second equality; after all, we can derive that:

$$e^*e^* + 1 \equiv e^* + 1 \equiv e^*$$

Conversely, to show that $e^* \leqq e^{**}$, it suffices (again by the left fixpoint rule) to prove that $ee^{**} + 1 \leqq e^{**}$, which can be done as follows:

$$ee^{**} + 1 \leqq e^*e^{**} + 1 \equiv e^{**} \qquad \square$$

Another example of backward reasoning follows.

Lemma 2.23. $e \cdot (f \cdot e)^* \equiv (e \cdot f)^* \cdot e$

Proof. By Lemma 2.19, it suffices to prove $e \cdot (f \cdot e)^* \leqq (e \cdot f)^* \cdot e$ and $(e \cdot f)^* \cdot e \leqq e \cdot (f \cdot e)^*$. To argue the former, we can apply the right fixpoint rule, which tells us that it is sufficent to prove

$$e + (e \cdot f)^* \cdot e \cdot f \cdot e \leqq (e \cdot f)^* \cdot e$$

This is true, as we can derive the following and apply Lemma 2.19(1).

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

The proof of $(e \cdot f)^* \cdot e \leqq e \cdot (f \cdot e)^*$ is similar, and left as Exercise 2.5. $\square$

Given how useful the provability relation is, a natural question to ask is whether it is "enough" to prove all equivalences that are true; more formally, is $\equiv$ *complete*, in the sense that $[\![e]\!] = [\![f]\!]$ implies $e \equiv f$? Answering this question requires a good deal of mathematical machinery, which will be developed over the next few chapters. By the end of Chapter 5, we will be able to answer in the positive.

2.3 Exercises

2.1. In Example 2.6, we have 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 program as a regular 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 as a regular expression using the technique from Example 2.7, writing $\texttt{lte} \cdot \texttt{writeL} + \texttt{gt} \cdot \texttt{writeR}$, where

- `lte` and `gt` are programs that check whether $M^2 \leq y$ and $M^2 > y$ respectively, and abort 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 a two-variable state.*

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

2.2. The snippet you saw above is actually part of a larger program, which finds the integer square root of y using 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 previous exercise.

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

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

2.3. Prove that, for all $e \in \mathsf{Exp}$, it holds that $[\![1 + e \cdot e^*]\!] = [\![e^*]\!]$.

2.4. Recall that $e \leqq f$ is shorthand for $e + f \equiv f$. Let's verify the claims in Lemma 2.19. In the following, let $e, f, g \in \mathsf{Exp}$; prove:

(a) If $e \equiv f$ then $e \leqq f$.

(b) If $e \leqq f$ and $f \leqq e$, then $e \equiv f$.

(c) If $e \leqq f$ and $f \leqq g$, then $e \leqq g$.

(d) If $e \leqq f$, then $e + g \leqq f + g$ and $g + e \leqq g + f$.

(e) If $e \leqq f$, then $e \cdot g \leqq f \cdot g$ and $g \cdot e \leqq g \cdot f$.

(f) If $e \leqq f$, then $e^* \leqq f^*$.

2.5. In Lemma 2.23, we showed that $e \cdot (f \cdot e)^* \leqq (e \cdot f)^* \cdot e$. Argue the other direction, i.e., that $(e \cdot f)^* \cdot e \leqq e \cdot (f \cdot e)^*$.

2.6. Let $e, f \in \mathsf{Exp}$. Prove that $(e + f)^* \equiv e^* \cdot (f \cdot e^*)^*$.

2.7. Let $e \in \mathsf{Exp}$. Prove that for all $n > 0$, we have $e^* \equiv (1+e)^{n-1}(e^n)^*$

2.8. Let $e, f, g \in \mathsf{Exp}$. Prove that if $ef \equiv fg$, then $e^*f \equiv fg^*$.

2.9. Let $e \in \mathsf{Exp}$. Show that $(e + 1)^* \equiv e^*$.

2.10. Let's fill in the proof sketch of Lemma 2.10.

(a) Show that for all $L, K \subseteq A^*$, the following hold:

$$\hat{\sigma}(L \cup K) = \hat{\sigma}(L) \cup \hat{\sigma}(K)$$

$$\hat{\sigma}(L \cdot K) = \hat{\sigma}(L) \circ \hat{\sigma}(K) \qquad \hat{\sigma}(L^*) = \hat{\sigma}(L)^*$$

(b) Use the above to prove that for each interpretation $\langle S, \sigma \rangle$ and $e \in \mathsf{Exp}$, it holds that $[\![e]\!]_\sigma = \hat{\sigma}([\![e]\!])$, by induction on e.

2.11. We now complete the sketched proof of Lemma 2.11. Let $e \in \mathsf{Exp}$. Show that for all $w, x \in A^*$, we have that $\langle w, wx \rangle \in [\![e]\!]_\sharp$ if and only if $x \in [\![e]\!]$. *Hint: prove something slightly stronger, namely* $[\![e]\!]_\sharp = \{\langle w, wx \rangle \mid w \in A^*, x \in [\![e]\!]\}$, *by induction on e.*

2.12. Use Lemma 2.17 to prove that $(\mathtt{a} + \mathtt{b})^* \not\equiv \mathtt{a}^* \cdot (\mathtt{b} \cdot \mathtt{a})^*$. Be precise in your arguments — both in how you use soundness, and what remains to be proved after that.

2.13. Use Lemma 2.17 *and* Theorem 2.12 to show that $(\mathtt{a} + \mathtt{b})^* \not\equiv \mathtt{a}^* \cdot (\mathtt{b} \cdot \mathtt{a})^*$. (Note that this does *not* contradict Exercise 2.6).

3

A primer on coalgebra

In this chapter, we discuss the very basics of a general theory of abstract machines, known as *(universal) coalgebra* [5]. This will give us a very natural way to define the operational semantics of a machine, as well as a sensible idea of when two machines agree on their semantics. Coalgebra will be particularly useful when we discuss the operational semantics of languages in the next chapter. Although we could develop the operational semantics of regular expressions concretely, the abstract coalgebraic perspective provides a unifying framework as we will see in Chapter 6, where all definitions (and even the proof of completeness) fit the same pattern as for Kleene algebra.

The material in this chapter is largely based on [5]. The interested reader is referred to op. cit., and also to [6] for an extensive discussion.

3.1 Functors

Suppose you encounter a machine, in the form of an opaque box with some combination of buttons and displays. When you press these buttons, you may change its internal state; as a result of your actions, some information may also appear on one of the displays. Universal coalgebra is a way of thinking of such a machine in terms of the interactions it allows, and how it evolves over time as a result of these interactions.

Let's model the states of the machine as some set X. In every state $x \in X$, the machine may offer some combination of output values, as well as opportunities for interaction which yield a new machine state. We can think of the possible values of these outputs and interactions as another set; because this set may be built from the possible states of the machine, we denote it with $F(X)$. The behavior of the machine can then be thought of as a function $t\colon X \to F(X)$ that assigns, to every state $x \in X$, a combination of outputs and interactions $t(x)$.

Example 3.1. Suppose our hypothetical machine has just one button. When somebody presses this button, either nothing happens (i.e., the user does not observe any change), or the machine self-combusts, leaving behind a heap of ashes (and its perplexed user). We can model the states of the machine as some set X, and its set of outputs and interactions as $M(X) = X + 1$: either $t(x) \in X$, which means the machine has transitioned to some new (but unobservable) state, or $t(x) = *$, in which case the machine has gone up in flames.

Example 3.2. Suppose we are studying a coffee machine. When you press a button $b \in B$, the machine pours out some type of coffee $c \in C$; it then transitions to a new state. With X as a set of machine states, the interface of this machine is given by $H(X) = (C \times X)^B$, and its behavior as a function $t\colon X \to H(X)$. When $x \in X$, $b \in B$ and $c \in C$, $t(x)(b) = (c, x')$ means that if button b is pressed in state x, the machine will pour coffee type c and transition to state x'.

If F is either M or H above, we can lift a function $f\colon X \to Y$ to a function $F(f)\colon F(X) \to F(Y)$, by applying f to the values from X that "live inside" elements of $F(X)$; this lifting is compatible with composition and the identity. This is formalised in the next definition.

Definition 3.3 (Functor). A *functor* F associates with every set X a set $F(X)$, and with every function $f\colon X \to Y$ a function $F(f)\colon F(X) \to F(Y)$, such that (1) for every set X, F preserves the identity function, i.e., $F(\mathsf{id}_X) = \mathsf{id}_{F(X)}$ and (2) for all functions $f\colon X \to Y$ and $g\colon Y \to Z$, F is compatible with composition, i.e., $F(g \circ f) = F(g) \circ F(f)$.

Example 3.4. Given a set X, let $M(X) = X + 1$, as in Example 3.1. We can then assign to every function $f\colon X \to Y$ a function $M(f)\colon M(X) \to M(Y)$ by setting $M(f)(x) = f(x)$ when $x \in X$, and $M(f)(*) = *$; you may check that this makes M a functor — see Exercise 3.1.

Example 3.5. Given a set X, let $H(X)$ be as in Example 3.2. For $f\colon X \to Y$, we can choose $H(f)\colon H(X) \to H(Y)$ as follows. Given $m\colon B \to C \times X$, we define $H(f)(m)\colon B \to C \times Y$ by setting $H(f)(m)(b) = (c, f(x))$ when $m(b) = (c, x)$. This defines a functor; see Exercise 3.2.

In what follows, we will refer to the functors M and H as defined above without further cross-references.

Example 3.6. Suppose $F(X) = 1$ when $X = \mathbb{R}$, and $F(X) = X$ otherwise. This *cannot* be a functor. To see why, suppose towards a contradiction that for each $f\colon X \to Y$ we can define an $F(f)\colon F(X) \to F(Y)$ satisfying both functor properties. Now consider the injection $i\colon \mathbb{N} \to \mathbb{R}$ and the floor function $\lfloor\cdot\rfloor\colon \mathbb{R} \to \mathbb{N}$. Clearly $\lfloor\cdot\rfloor \circ i$ is the identity on $\mathbb{N}$; since F is a functor, we have that $F(\lfloor\cdot\rfloor \circ i)$ is the identity on $F(\mathbb{N}) = \mathbb{N}$ by the first property. Now, $F(i)$ goes from $\mathbb{N}$ to $F(\mathbb{R}) = 1$, and is therefore constant; but then so is $F(\lfloor\cdot\rfloor \circ i) = F(\lfloor\cdot\rfloor) \circ F(i)$ by the second property — we have reached a contradiction.

3.2 Coalgebras and Homomorphisms

We are now ready to define coalgebras. The name *coalgebra* or *co-algebra* comes from category theory, where it is the dual to *algebra*, an abstraction of universal algebra (see for instance [7]). If an algebra tells you how to "compose" objects into new ones, then a coalgebra tells you how to "decompose" or unfold an object; this makes them suitable to model the behavior of machines, where one state evolves to the next.

Definition 3.7 (F-coalgebra)**.** Let F be a functor. An *F-coalgebra* is a pair (X, t) where X is a set and $t\colon X \to F(X)$ is a function.

Now suppose we have two abstract machines with the same buttons and displays, modeled as F-coalgebras for a certain functor F that describes their interface. How would we check whether the behavior of each state in the first machine also exists in the other? The coalgebraic way to do this is to demonstrate a function that maps a state of the first machine to a state of the second machine, in a way that preserves the observable values and next states. This leads to the following notion.

Definition 3.8 (Homomorphism)**.** Let F be a functor, and let (X, t) and (Y, u) be F-coalgebras. A function $f \colon X \to Y$ is called an *(F-coalgebra) morphism* if it holds that $u \circ f = F(f) \circ t$.

Example 3.9. An M-coalgebra morphism from (X, t) to (Y, u) is a function $f \colon X \to Y$ such that (1) if $t(x) = *$, then $u(f(x)) = *$ as well, and (2) if $t(x) \in X$, then $f(t(x)) = u(f(x))$. The first condition says that if the first machine is in state x and pressing the button detonates it, then so does pressing the button on the second machine in state $f(x)$. The second condition tells us that if pressing the button on the first machine in state x does set off the charge but takes us to state x', then pressing the button on the second machine in state $f(x)$ does not either, but takes us to $f(x')$. Thus, if the first machine catches fire after pressing the button n times starting in state x, then so does the second machine when repeating this experiment starting from state $f(x)$.

Example 3.10. An H-coalgebra morphism from (X, t) to (Y, u) is a function $f \colon X \to Y$ such that if $t(x)(b) = (c, x')$, then $u(f(x))(b) = (c, f(x'))$. This means that if pressing button b when the first machine is in state x yields coffee type c, then so does pressing this button on the second machine in state $f(x)$; moreover, a transition to x' in the first machine corresponds to a transition to $f(x')$ in the second machine.

To make sense of properties that relate compositions of functions like the one in Definition 3.8, it sometimes helps to have a graphical depiction like the diagram below: the (co)domains of u, f, $F(f)$ and t are drawn as nodes, and the functions between them as arrows.

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "t"'] & Y \arrow[d, "u"] \\
F(X) \arrow[r, "F(f)"] & FY
\end{tikzcd}$$

In the sequel, we will say that such a diagram *commutes* when composing functions along two paths between nodes gives the same function. For instance, the diagram above commutes precisely when f is an F-coalgebra morphism from (X, t) to (Y, u), i.e., when $u \circ f = F(f) \circ t$.

We introduced coalgebra morphisms as a way to recover the behavior of one machine inside the other. One might wonder whether we can create an F-coalgebra that unambiguously contains the behavior of *all* F-coalgebras. The following captures this idea.

Definition 3.11 (Final Coalgebra). Let F be a functor. An F-coalgebra (Ω, ω) is *final* if for every F-coalgebra (X, t) there exists precisely one F-coalgebra morphism $!_X \colon X \to \Omega$, i.e., such that the following commutes:

$$\begin{tikzcd}
X \arrow[r, dashed, "!_X"] \arrow[d, "t"'] & \Omega \arrow[d, "\omega"] \\
F(X) \arrow[r, dashed, "F(!_X)"] & F(\Omega)
\end{tikzcd}$$

In this diagram, we have dashed the horizontal arrows to emphasize that they arise from the vertical arrows.

Existence of a morphism into a final F-coalgebra guarantees that it can model the behavior of every state of every F-coalgebra, while uniqueness tells us that there is only one possible way to do this. A final coalgebra may not always exist for every functor, but when it does, its state set is typically suited to act as semantics for F-coalgebras, because they contain enough structure to describe the behavior of a state. The following examples illustrate this idea further.

Example 3.12. We return once more to the functor M. Let $\mathbb{N}^{\infty}_{>0}$ consist of all strictly positive natural numbers, as well as the symbol ∞. We can define an M-coalgebra $(\mathbb{N}^{\infty}_{>0}, \bullet)$, defining $\bullet \colon \mathbb{N}^{\infty}_{>0} \to \mathbb{N}^{\infty}_{>0} + 1$ by

$$\bullet(n) = \begin{cases} \infty & n = \infty \\ * & n = 1 \\ n - 1 & n > 1 \end{cases}$$

This is exactly the final M-coalgebra: given an M-coalgebra (X, t), we can define $!_X : X \to \mathbb{N}^{\infty}_{>0}$ by sending $x \in X$ to the number of button presses it takes to detonate the machine starting from state x, and to ∞ when this is not possible. Showing that this is an M-coalgebra morphism from (X, t) to $(\mathbb{N}^{\infty}_{>0}, \bullet)$ is left as Exercise 3.6.

To see that $!_X$ is the *only* such M-coalgebra morphism, let $f : X \to \mathbb{N}^{\infty}_{>0}$ be any M-coalgebra morphism from (X, t) to $(\mathbb{N}^{\infty}_{>0}, \bullet)$. If $t^n(x) = *$ for $n > 1$, then we show that $f(x) = !_X(x)$ by induction on n. In the base, $n = 1$ implies $t(x) = *$, and hence $\bullet(f(x)) = *$ because f is a morphism; therefore $f(x) = 1 = !_X(x)$. For the inductive step, $t^{n+1}(x) = *$ for some $n > 1$. In that case $\bullet(t(x)) = f(t(x))$ because f is a morphism, and $f(t(x)) = !_X(t(x))$ by induction; since $!_X(t(x)) = n$ by definition of $!_X$, we conclude that $\bullet(t(x)) = n$, and hence $t(x) = n + 1 = !_X(x)$.

Otherwise, if $t^n(x) \in X$ for all $n > 0$, then $M(\bullet^n)(f(x)) \in \mathbb{N}^{\infty}_{>0}$ for all $n > 0$ as well. This means that $f(x)$ can only be $!_X(x) = \infty$.

Example 3.13. The final H-coalgebra has as a set of states S, consisting of functions $s : B^+ \to C$ — intuitively, such a function assigns to a sequence of button presses $b_1 \cdots b_n$ the types of coffee poured as a result of this sequence. Its structure map $\sigma : S \to H(S)$ is defined by $\sigma(s)(b) = (s(b), s_b)$, where $s_b : B^+ \to C$ is given by $s_b(w) = s(bw)$.

Given an H-coalgebra (X, t), we assign to each $x \in X$ a function $!_X(x) : B^+ \to C$, which assigns to every $w \in B^+$ a value $!_X(x)(w) \in C$, as follows: given that $t(x)(b) = (c, x')$, we set $!_X(x)(b) = c$, while $!_X(x)(bw) = !_X(x')(w)$ for all $w \in B^+$. Note how the latter is defined by induction, as w is a shorter string than bw. Showing that this is is the only H-coalgebra morphism from (X, t) to (S, σ) is left as Exercise 3.7.

If a final coalgebra exists, it is unique "up to isomorphism" — in other words, two final coalgebras must be related by a bijective morphism, whose inverse is also a morphism. We record this formally, as follows.

Lemma 3.14. Let (Ω, ω) and (Ω', ω') both be final F-coalgebras for some functor F. There exists a bijective F-coalgebra morphism $f : \Omega \to \Omega'$ such that the inverse $f^{-1} : \Omega' \to \Omega$ is also an F-coalgebra morphism.

For this reason, we prefer to speak of *the* final coalgebra.

3.3 Bisimulations

Coalgebra morphisms can be used to show that the behavior of a state in one coalgebra is replicated by that of another state in another coalgebra. However, they may not be sufficient, as shown below.

Example 3.15. Consider M-coalgebras $(\{0,1\}, n)$ and $(\{0,1,2\}, m)$ with

$$n(0) = 1 \qquad n(1) = 0 \qquad m(0) = 1 \qquad m(1) = 2 \qquad m(2) = 0$$

Clearly, all states in both coalgebras have the same behavior, as pressing the button any number of times will keep the device intact. However, there is no M-coalgebra morphism from $(\{0,1\}, n)$ to $(\{0,1,2\}, m)$, or vice versa. The details are left as Exercise 4.2.

In situations such as the above, we need a different way to relate elements in coalgebras with the same behavior. This brings us to the notion of *bisimulation*, defined as follows.

Definition 3.16. Let F be a functor, and let (X, t) and (Y, u) be F-coalgebras. An *F-bisimulation* (between (X, t) and (Y, u)) is a relation $R \subseteq X \times Y$ that can be equipped with a coalgebra structure $\rho\colon R \to F(R)$ such that the projections $\pi_1\colon R \to X$ and $\pi_2\colon R \to Y$ given by $\pi_1(x, y) = x$ and $\pi_2(x, y) = y$ are F-coalgebra morphisms.

When there exists a bisimulation that relates $x \in X$ to $y \in Y$, we say that x and y are *bisimilar*, denoted $X, x \leftrightarroweq Y, y$, or simply $x \leftrightarroweq y$ when the coalgebras are clear from context.

This definition is somewhat abstract, but for concrete cases a bisimulation can usually be characterised rather simply in terms of the transition structure of the coalgebras. Let's see some examples.

Example 3.17. A bisimulation between M-coalgebras (X, t) and (Y, u) is precisely a relation $R \subseteq X \times Y$ such that if $x \mathrel{R} y$, then (1) $t(x) = *$ if and only if $u(y) = *$, and (2) if $t(x) \in X$ and $u(y) \in Y$, then $t(x) \mathrel{R} u(y)$. Note how $t(x) \in X$ if and only if $u(y) \in Y$ by the first condition, and so only one of the premises to the second condition needs to hold. Proving that this characterizes M-bisimulations is left as Exercise 3.10.

Let us return to the coalgebras $(\{0,1\}, n)$ and $(\{0,1,2\}, m)$ from Example 3.15. Using the characterization above, we can show that $R = \{0,1\} \times \{0,1,2\}$ is a bisimulation, and so all states are bisimilar.

Example 3.18. A bisimulation between H-coalgebras (X, t) and (Y, u) is a relation $R \subseteq X \times Y$ such that if $x \mathrel{R} y$, then for each $b \in B$ there exists a $c \in C$ such that $t(x)(b) = (c, x')$ and $u(y)(b) = (c, y')$ with $x' \mathrel{R} y'$. Validating this characterization is left as Exercise 3.11.

Given that the states related by a bisimulation are equivalent, it should stand to reason that we can shrink our coalgebra by identifying related states. The following result records this formally.

Lemma 3.19 ([5, Proposition 5.8])**.** Let (X, t) be an F-coalgebra for some F, and let R be a bisimulation on (X, t), with $\equiv_R$ its equivalence closure. We can equip $X/{\equiv_R}$ with an F-coalgebra structure $t/{\equiv_R}$ that makes $[-]\colon X \to X/R$ a homomorphism from (X, t) to $(X/{\equiv_R}, t/{\equiv_R})$.

Proof. We choose $t/{\equiv_R}\colon X/{\equiv_R} \to F(X/{\equiv_R})$ as

$$t/{\equiv_R}([x]) = F([-])(t(x)) \tag{3.1}$$

To see that this is well-defined, we should check that the choice of the representative x does not matter, i.e., that if $x \equiv_R x'$, then $F([-])(t(x)) = F([-])(t(x'))$. To this end, it suffices to check the case where $x \mathrel{R} x'$. Let $r\colon R \to FR$ be the coalgebra structure on R such that $t \circ \pi_i = F(\pi_i) \circ r$ for $i \in \{1, 2\}$. We then derive as follows

$$\begin{aligned}
F([-])(t(x)) &= F([-])(t(\pi_1(x, x'))) \\
&= F([-])(F(\pi_1)(r(x, x'))) \\
&= F([-] \circ \pi_1)(r(x, x')) \\
&= F([-] \circ \pi_2)(r(x, x')) && (*) \\
&= F([-])(F(\pi_2)(r(x, x'))) \\
&= F([-])(t(\pi_2(x, x'))) \\
&= F([-])(t(x'))
\end{aligned}$$

In $(*)$, we used the fact that $[-] \circ \pi_1 = [-] \circ \pi_2$ by construction of $\equiv_R$: after all, if $y \mathrel{R} y'$, then $y \equiv_R y'$. The other steps apply properties of functors, coalgebra morphisms, and the fact that $x \mathrel{R} x'$.

Finally, as (3.1) holds by construction, we have that $[-]$ is an F-coalgebra morphism from (X, t) to $(X/{\equiv_R}, t/{\equiv_R})$. □

3.4 Relating the notions of equivalence

We now have two ways of arguing that two states in a coalgebra have the same behavior: we can build a bisimulation relating the states, or we can show that both states are mapped to the same element in the final coalgebra. These two kinds of equivalence coincide for many functors, as we will detail now. In fact, one direction holds for all functors: if two states are bisimilar then they are mapped to the same element of the final coalgebra (this latter notion is often called *behavioral equivalence*).

Lemma 3.20. Let (X, t) be a coalgebra for some functor F, and let (Ω, ω) be a final F-coalgebra. If $x_1, x_2 \in X$ with $x_1 \Leftrightarrow x_2$, then $!_X(x_1) = !_X(x_2)$.

Proof. Let R be a bisimulation witnessing that $x_1 \Leftrightarrow x_2$, and let $\equiv_R$ be its equivalence closure. By Lemma 3.19, we can construct the coalgebra $(X/{\equiv_R}, t/{\equiv_R})$ in a way that makes $[-]\colon X \to X/{\equiv_R}$ a coalgebra morphism. Since $!_X$ and $!_{X/\equiv_R} \circ [-]$ are both F-coalgebra morphisms from (X, t) to the final coalgebra (Ω, ω), they must be the same morphism, by uniqueness. We can then conclude that

$$!_X(x_1) = !_{X/\equiv_R}([x_1]) = !_{X/\equiv_R}([x_2]) = !_X(x_2) \qquad \square$$

The previous lemma tells us that bisimulations are a sound proof technique for behavioral equivalence. That is: if we have built a bisimulation between two states, then we know they will be mapped to same state in the final coalgebra. The opposite direction does not hold in general, but we present below a class of functors for which it holds. We need to introduce the notion of kernel and kernel compatible functors.

Definition 3.21 (Kernel)**.** Let $f\colon X \to Y$ be a function. The *kernel* of f is the set $\mathsf{Ker}(f) = \{(x_1, x_2) \in X^2 \mid f(x_1) = f(x_2)\}$. The *projection maps* $\pi_1^f, \pi_2^f\colon \mathsf{Ker}(f) \to X$ are given by $\pi_i^f(x_1, x_2) = x_i$.

Intuitively, the kernel of a function is nothing more than the relation it induces; it should be clear that this is an equivalence relation.

Definition 3.22. Let F be a functor. We call F *kernel-compatible* when for all $f\colon X \to Y$ and $p, q \in FX$ with $F(f)(p) = F(f)(q)$ (i.e., $(p, q) \in \mathsf{Ker}(F(f))$), there exists an $r \in F(\mathsf{Ker}(f))$ s.t. $F(\pi_1^f)(r) = p$ and $F(\pi_2^f)(r) = q$. Equivalently, F is kernel-compatible when for all $f\colon X \to Y$ there exists an $m_f\colon \mathsf{Ker}(F(f)) \to F(\mathsf{Ker}(f))$ making the diagram below commute.

$$\begin{tikzcd}
 & \mathsf{Ker}(F(f)) \arrow[dl, bend right, "\pi_1^{F(f)}"'] \arrow[d, "m_f"'] \arrow[dr, bend left, "\pi_2^{F(f)}"] & \\
F(X) & F(\mathsf{Ker}(f)) \arrow[l, "F(\pi_1^f)"'] \arrow[r, "F(\pi_2^f)"] & F(X)
\end{tikzcd}$$

Remark 3.23. The experienced reader may recognize kernel-compatibility as a special case of *weak pullback preservation* [5]. We use the former because it allows us to avoid defining pullbacks, and it is all we need to relate behavioral equivalence to bisimilarity for Set-based coalgebras.

Kernel-compatibility may feel rather abstract. On an intuitive level, we can think of it as follows: if F is kernel-compatible and $F(f)$ identifies two elements u and v of FX, then m_f helps us "factor" u and v into a *single* piece of structure coming from F and elements u' and v' of X that appear inside u and v respectively, such that u' and v' are already identified by f. The following examples aim to clarify this.

Example 3.24. M is kernel-compatible: given $f\colon X \to Y$ and $u, v \in M(X)$ with $M(f)(u) = M(f)(v)$ (i.e., $(u, v) \in \mathsf{Ker}(M(f))$), either $u = * = v$, or $u, v \in X$ with $f(u) = f(v)$. In the former case, set $m_f(u, v) = *$, and $m_f(u, v) = (u, v)$ in the latter; either way, $m_f(u, v) \in M(\mathsf{Ker}(f))$. This also makes the required diagram commute; cf. Exercise 3.13.

Example 3.25. H is also kernel-compatible: given $f\colon X \to Y$ and $u, v \in H(X)$ with $H(f)(u) = H(f)(v)$, we can choose $m_f(u, v)$ by setting $m_f(u, v)(b) = (c_u, (x_u, x_v))$, where $u(b) = (c_u, x_u)$ and $v(b) = (c_v, x_v)$. Showing that $m_f(u, v) \in H(\mathsf{Ker}(f))$ and the commutativity of the diagram is left as Exercise 3.14.

We can now prove that for kernel-compatible functors it is indeed the case the behavioral equivalence implies bisimilarity.

Lemma 3.26 ([5, Proposition 5.9])**.** Let (X, t) and (Y, u) be F-coalgebras, and let h be an F-coalgebra homomorphism from (X, t) to (Y, u). If F is kernel-compatible, then $\mathsf{Ker}(h)$ is a bisimulation on (X, t).

Proof. We choose $p\colon \mathsf{Ker}(h) \to \mathsf{Ker}(Fh)$ by $p(x, x') = (t(x), t(x'))$. To see that this is well-defined, note that if $(x, x') \in \mathsf{Ker}(h)$, then $(t(x), t(x')) \in \mathsf{Ker}(F!_X)$ because $h(x) = h(x')$ and we have that

$$Fh(t(x)) = u(h(x)) = u(h(x')) = Fh(t(x'))$$

It now suffices to prove that $(\mathsf{Ker}(h), m_h \circ p)$ is a bisimulation on (X, t); to see this, consider the following diagram.

$$\begin{tikzcd}
X \arrow[dd, "t"'] & \mathsf{Ker}(h) \arrow[l, "\pi_1^h"'] \arrow[r, "\pi_2^h"] \arrow[d, "p"'] & X \arrow[dd, "t"] \\
& \mathsf{Ker}(Fh) \arrow[dl, "\pi_1^{Fh}"'] \arrow[d, "m_h"'] \arrow[dr, "\pi_2^{Fh}"] & \\
FX & F(\mathsf{Ker}(h)) \arrow[l, "F(\pi_1^h)"] \arrow[r, "F(\pi_2^h)"'] & FX
\end{tikzcd}$$

Here, the lower triangles commute by definition of m_h, and the upper quadrangles commute by construction of p. □

Corollary 3.27. Let (X, t) be an F-coalgebra for some functor F. Furthermore, let (Ω, ω) be a final F-coalgebra. If $x, x' \in X$ are such that $!_X(x) = !_X(x')$, and F is kernel-compatible, then $x \leftrightarroweq x'$.

Proof. Choose $\mathsf{Ker}(!_X)$ as the bisimulation witnessing that $x_1 \leftrightarroweq x_2$. □

3.5 Subcoalgebras

When treating a transition system, it often happens that there are subsets of states that only transition to one another, and not to states outside of the set. This notion can also be formalized within coalgebra.

Definition 3.28. Let (X, t) and (Y, u) be F-coalgebras for some functor F. We say (X, t) is a *subcoalgebra* of (Y, u) if there exists a coalgebra homomorphism $f\colon (X, t) \to (Y, u)$ that is injective (as a function).

One instance of the above could have Y be a subset of X, with the injection of Y into X being a homomorphism. This corresponds well to the intuition of states in Y only transitioning to states in Y.

We finish chapter with one more observation that will be useful later. Recall that a homomorphism $f\colon (X, t) \to (Y, u)$ shows how (Y, u) can "replicate" the behavior of (X, t). We can recover the relevant subcoalgebra of (Y, u) by means of this homomorphism, as follows.

Lemma 3.29 (cf. [5, Theorem 7.1]). Let F be kernel-compatible, let (X, t) and (Y, u) be F-coalgebras, and let $f\colon (X, t) \to (Y, u)$ be a homomorphism. Now $(X/\mathsf{Ker}(f), t/\mathsf{Ker}(f))$ (cf. Lemmas 3.19 and 3.26) is a subcoalgebra of (Y, u), and $f = i \circ [-]_{\mathsf{Ker}(f)}$, where i is the map from $X/\mathsf{Ker}(f)$ to Y given by $i([x]_{\mathsf{Ker}(f)}) = f(x)$.

Proof. First, let us check that i is a well-defined function: if $[x]_{\mathsf{Ker}(f)} = [x']_{\mathsf{Ker}(f)}$, then $f(x) = f(x')$, and hence $i([x]_{\mathsf{Ker}(f)}) = i([x']_{\mathsf{Ker}(f)})$.

Note that we already know that the left square below commutes; it remains to show that the right square below does, too.

$$\begin{tikzcd}
X \arrow[r, "{[-]_{\mathsf{Ker}(f)}}"] \arrow[d, "t"'] & X/\mathsf{Ker}(f) \arrow[r, "i"] \arrow[d, "t/\mathsf{Ker}(f)"'] & Y \arrow[d, "u"'] \\
FX \arrow[r, "{F[-]_{\mathsf{Ker}(f)}}"'] & F(X/\mathsf{Ker}(f)) \arrow[r, "Fi"'] & FY
\end{tikzcd}$$

To this end, we should show that $u(i([x]_{\mathsf{Ker}(f)})) = Fi(t/\mathsf{Ker}(f))([x]_{\mathsf{Ker}(f)})$ for any $x \in X$, which we do by deriving as follows:

$$\begin{aligned}
u(i([x]_{\mathsf{Ker}(f)})) &= u(f(x)) && \text{(def. } i\text{)} \\
&= F(f)(t(x)) && (f \text{ is a hom.)} \\
&= F(i \circ [-]_{\mathsf{Ker}(f)})(t(x)) && \text{(def. } i\text{)} \\
&= Fi(F[-]_{\mathsf{Ker}(f)}(t(x))) && (F \text{ is a functor)} \\
&= Fi(t/\mathsf{Ker}(f)([x]_{\mathsf{Ker}(f)})) && ([-]_{\mathsf{Ker}(f)} \text{ is a hom.)}
\end{aligned}$$

Finally, i is injective: if $i([x]_{\mathsf{Ker}(f)}) = i([x]_{\mathsf{Ker}(f)})$, then $f(x) = f(x')$, so $[x]_{\mathsf{Ker}(f)} = [x']_{\mathsf{Ker}(f)}(x')$; this concludes the proof. □

3.6 Exercises

3.1. Confirm that M, as defined in Example 3.4, is a functor.

3.2. Confirm that H, as defined in Example 3.5, is a functor.

3.3. For a set X, let $D(X) = 2 \times X^A$ and, for $f \colon X \to Y$, define $D(f) \colon D(X) \to D(Y)$ as follows. Given $(o, m) \colon 2 \times X^A$, we define $D(f)(o, m) \colon 2 \times Y^A$ by setting $D(f)(o, m) = (o, f \circ m)$. Prove that this defines a functor.

3.4. Reconsider the functor D from the previous exercise. Can you description of the conditions satisfied by a D-coalgebra homomorphism, along the same lines as Examples 3.9 and 3.10?

3.5. Let F be a functor, and let (X, t_X), (Y, t_Y) and (Z, t_Z) be F-coalgebras, with $f \colon X \to Y$ an F-coalgebra morphism from (X, t_X) to (Y, t_Y), and $g \colon Y \to Z$ an F-coalgebra morphism from (Y, t_Y) to (Z, t_Z). Show that $g \circ f \colon X \to Z$ is an F-coalgebra morphism from (X, t_X) to (Z, t_Z).

3.6. Let (X, t) be an M-coalgebra, and let $!_X \colon X \to \mathbb{N}^{\infty}_{>0}$ be defined as in Example 3.12. Show that $!_X$ is an M-coalgebra morphism from (X, t) to $(\mathbb{N}^{\infty}_{>0}, \bullet)$.

3.7. Let (X, t) be an H-coalgebra. Furthermore, let S be the set of functions from B^+ to C, with $\sigma \colon S \to H(S)$ and $!_X \colon X \to S$ defined as in Example 3.13. Show that $!_X$ is the *only* H-coalgebra morphism from (X, t) to (S, σ).

3.8. Prove Lemma 3.14. You may use the property proved in Exercise 3.5, and the fact that for any F-coalgebra (X, t), the identity id_X is an F-coalgebra morphism from (X, t) to itself.

3.9. Recall the two coalgebras from Example 3.15. Can you show formally that there is no homomorphism from one to the other?

3.10. Verify the characterization of M-bisimulations from Example 3.17.

3.11. Verify the characterization of H-bisimulations from Example 3.18.

3.12. Consider the functor D from Exercise 3.3. Can you characterize D-bisimulations, along the same lines as Examples 3.17 and 3.18?

3.13. Show that the choice of m_f in Example 3.24 makes the diagram from Definition 3.22 commute for the functor M.

3.14. Finish the argument towards kernel-compatibility of H, outlined in Example 3.25. In particular, show that the choice of m_f is valid — i.e., if $(u, v) \in \mathsf{Ker}(H(f))$ then $m_f(u, v) \in H(\mathsf{Ker}(f))$ — and that the diagram from Definition 3.22 commutes.

3.15. Show that the functor D from Exercise 3.3 is kernel-compatible.

4

Automata and Kleene's Theorem

We will now provide an *operational* view of regular languages through automata and, more abstractly, coalgebra. This latter perspective will provide the groundwork for the completeness proof in Chapter 5. This chapter is divided in three parts: first, we present the operational semantics of regular expressions; second, we dive into the coalgebraic view on automata and languages; third, we prove Kleene's theorem, which states that the denotational and operational semantics of regular expressions coincide — i.e., they provide equivalent semantics.

4.1 Derivatives

We start by introducing the notion of *derivative*, which is central to the study of the operational semantics of regular expressions. We will define two functions: one that assigns to each expression an *output value* in $\{0, 1\}$ signaling whether as a state it is final or not; and another that given a letter $\mathtt{a} \in A$ returns its $\mathtt{a}$-*derivative*, which is an expression that denotes all the words obtained from the first expression after *deleting* an $\mathtt{a}$. These functions were first proposed by Brzozowski [8], which is why they are known as *Brzozowski derivatives.*

Definition 4.1 (Brzozowski derivatives)**.** The *output function* $o\colon \mathsf{Exp} \to 2$ is defined inductively, as follows:

$$o(0) = 0 \qquad o(1) = 1 \qquad o(\mathtt{a}) = 0$$
$$o(e+f) = o(e) \vee o(f) \qquad o(ef) = o(e) \wedge o(f) \qquad o(e^*) = 1$$

Here, $b_1 \vee b_2$ is understood to be 0 when $b_1 = b_2 = 0$, and 1 otherwise; similarly, $b_1 \wedge b_2$ is defined as 1 when $b_1 = b_2 = 1$ and 0 otherwise.

We furthermore define the *transition function* $t\colon \mathsf{Exp} \to \mathsf{Exp}^A$ inductively as follows, where we use $e_\mathtt{a}$ as a shorthand for $t(e)(\mathtt{a})$:

$$0_\mathtt{a} = 1_\mathtt{a} = 0 \qquad (e+f)_\mathtt{a} = e_\mathtt{a} + f_\mathtt{a} \qquad (e^*)_\mathtt{a} = e_\mathtt{a} e^*$$

$$\mathtt{b}_\mathtt{a} = \begin{cases} 1 & \text{if } \mathtt{a} = \mathtt{b} \\ 0 & \text{if } \mathtt{a} \neq \mathtt{b} \end{cases} \qquad (ef)_\mathtt{a} = \begin{cases} e_\mathtt{a} f & \text{if } o(e) = 0 \\ e_\mathtt{a} f + f_\mathtt{a} & \text{otherwise} \end{cases}$$

Intuitively, for a regular expression e, we have $o(e) = 1$ if and only if $[\![e]\!]$ contains the empty word ϵ, while $e_\mathtt{a}$ denotes the language containing all words w such that $\mathtt{a}w$ is in the language denoted by e. Thus, derivatives tell us how to "deconstruct" a regular expression in terms of its *immediate* behavior (whether or not the empty word is accepted), and its *later* behavior (the words denoted that begin with a given letter). The next result, which is useful on a number of occasions, validates this intuition by telling us that this correspondence goes both ways: an expression can be "reconstructed" from its derivatives.

Theorem 4.2 (Fundamental theorem of Kleene Algebra [8, Theorem 6.4])**.** Every regular expression $e \in \mathsf{Exp}$ satisfies

$$e \equiv o(e) + \sum_{\mathtt{a} \in A} \mathtt{a} e_\mathtt{a}$$

Remark 4.3 (Generalized sum)**.** In the statement above, and in the sequel, we use a "generalized sum" operator $\sum$, interpreted intuitively the same as for numbers. This is a slight abuse of notation, because we use it to define a *term*, and it is not clear whether $\sum\{e, f, g\}$ denotes $f + (e + g)$ or $e + (g + f)$, or any other term described as "the sum of $\{e, f, g\}$". However, associativity and commutativity of $+$ up to $\equiv$ means that order and bracketing of sums does not matter in most contexts.

In a similar vein, we will often implicitly rely on familiar properties of summation that are valid in Kleene Algebra as well (in addition to the idempotence law $e + e \equiv e$) for the sake of brevity. Concretely, we will freely use the following properties of sums in the proof that follows, as well as in the sequel more generally, for all $X, Y \subseteq \mathsf{Exp}$ and $e \in \mathsf{Exp}$:

$$e + \sum_{f \in X} f \equiv \sum_{f \in X \cup \{e\}} f \qquad \sum_{f \in X \cup Y} f \equiv \sum_{f \in X} f + \sum_{f \in Y} f$$

$$e\Big(\sum_{f \in X} f\Big) \equiv \sum_{f \in X} e \cdot f \qquad \Big(\sum_{f \in X} f\Big)e \equiv \sum_{f \in X} f \cdot e$$

Corollary 4.4. For all $e \in \mathsf{Exp}$, we have that $[\![e]\!] = [\![o(e) + \sum_{\mathtt{a} \in A} \mathtt{a} e_{\mathtt{a}}]\!]$.

Proof of Theorem 4.2. We proceed by induction on the structure of e. The base cases are direct consequences of the laws for 0 and 1.

- When $e = 0$, we derive as follows.

$$0 \equiv 0 + \sum_{\mathtt{a} \in A} \mathtt{a}0 \equiv o(0) + \sum_{\mathtt{a} \in A} \mathtt{a}0_{\mathtt{a}}$$

- Next, when $e = 1$, we can calculate

$$1 \equiv 1 + 0 \equiv 1 + \sum_{\mathtt{a} \in A} \mathtt{a}0 \equiv o(1) + \sum_{\mathtt{a} \in A} \mathtt{a}1_{\mathtt{a}}$$

- Finally, when $e = \mathtt{b}$ for some $\mathtt{b} \in A$, we find that

$$\mathtt{b} \equiv 0 + \mathtt{b}1 + \sum_{\mathtt{a} \in A \setminus \{\mathtt{b}\}} \mathtt{a}0 \equiv o(\mathtt{b}) + \sum_{\mathtt{a} \in A} \mathtt{a}\mathtt{b}_{\mathtt{a}}$$

This leaves us with three inductive cases. For each, our induction hypothesis is that the statement is true for the proper subterms.

- When $e = e_1 + e_2$, we find that:

$$\begin{aligned} e_1 + e_2 &\equiv \Big(o(e_1) + \sum_{\mathtt{a} \in A} \mathtt{a}(e_1)_{\mathtt{a}}\Big) + \Big(o(e_2) + \sum_{\mathtt{a} \in A} \mathtt{a}(e_2)_{\mathtt{a}}\Big) && \text{(IH)} \\ &\equiv o(e_1) + o(e_2) + \sum_{\mathtt{a} \in A} \mathtt{a}((e_1)_{\mathtt{a}} + (e_2)_{\mathtt{a}}) \\ &\equiv o(e_1 + e_2) + \sum_{\mathtt{a} \in A} \mathtt{a}(e_1 + e_2)_{\mathtt{a}} \end{aligned}$$

- When $e = e_1e_2$, we derive:

$$\begin{aligned}
e_1e_2 &\equiv \Big(o(e_1) + \sum_{\mathsf{a}\in A} \mathsf{a}(e_1)_{\mathsf{a}}\Big)e_2 && \text{(IH)} \\
&\equiv o(e_1)e_2 + \sum_{\mathsf{a}\in A} \mathsf{a}(e_1)_{\mathsf{a}}e_2 \\
&\equiv o(e_1)\Big(o(e_2) + \sum_{\mathsf{a}\in A} \mathsf{a}(e_2)_{\mathsf{a}}\Big) + \sum_{\mathsf{a}\in A} \mathsf{a}(e_1)_{\mathsf{a}}e_2 && \text{(IH)} \\
&\equiv o(e_1e_2) + \sum_{\mathsf{a}\in A} \mathsf{a}((e_1)_{\mathsf{a}}e_2 + o(e_1)(e_2)_{\mathsf{a}}) \\
&\equiv o(e_1e_2) + \sum_{\mathsf{a}\in A} \mathsf{a}(e_1e_2)_{\mathsf{a}}
\end{aligned}$$

  Note how in this case the induction hypotheses for e_1 and e_2 are applied subsequently, and not at the same time.

- When $e = e_1^*$, the proof is a little bit more tricky. In the derivation below, we write x as a shorthand for $\sum_{\mathsf{a}\in A} \mathsf{a}(e_1)_{\mathsf{a}}$.

$$\begin{aligned}
e_1^* &\equiv (o(e_1) + x)^* && \text{(IH)} \\
&\equiv x^* && (\dagger) \\
&\equiv 1 + xx^* \\
&\equiv 1 + x(o(e_1) + x)^* && (\dagger) \\
&\equiv 1 + xe_1^* && \text{(IH)} \\
&\equiv 1 + \sum_{\mathsf{a}\in A} \mathsf{a}(e_1)_{\mathsf{a}}e_1^* && \text{(distributivity)} \\
&\equiv o(e_1^*) + \sum_{\mathsf{a}\in A} \mathsf{a}(e_1^*)_{\mathsf{a}}
\end{aligned}$$

  where in the two steps marked with $(\dagger)$, we use that for all $f \in \mathsf{Exp}$, $(f + 1)^* \equiv f$ (cf. Exercise 2.9). □

Remark 4.5. The name *Fundamental Theorem* is borrowed from [9], where a similar property is proved for formal power series (functions $f\colon A^* \to K$, for a semiring K). As explained there, the name is chosen in analogy to calculus: if we view prefixing with a as a kind of integration, then the theorem tells us that derivation is the opposite of integration, by obtaining e from the a-derivatives e_{a}, and the initial value $o(e)$.

4.2 Automata (coalgebraically)

The semantics of regular expressions $[\![-]\!]$ is *denotational*, in the sense that the interpretation of a regular expression arises from the that of its subexpressions. However, there is another way to assign semantics to regular expressions, which arises from the *operational* perspective. Here, we think of an expression as an *abstract machine*, which can evolve by performing certain computational steps; informally, the semantics of such an object is then taken as the set of all possible action sequences.

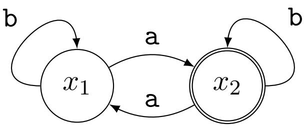
Remark 4.6. The definitions in this section will be presented in an elementary fashion. However, readers who have gone through Chapter 3 will be able to relate the definitions and claims here to the corresponding abstractions there, when instantiated with the appropriate functor. The remainder of this section is presented in an elementary fashion; we have inserted **Remarks in blue** to make the connection explicit, but readers who skipped Chapter 3 can disregard these.

More generally, the machines studied are the well-known *(deterministic) automata*; they form the operational model of regular expressions.

Definition 4.7. A *(deterministic) automaton* is a pair (X, d_X) where X is a set, and $d_X : X \to 2 \times X^A$ is a function. We will often write d_X as $\langle o_X, t_X \rangle$, where $o_X : X \to 2$ and $t_X : X \to X^A$ are functions such that $d_X(x) = \langle o_X(x), t_X(x) \rangle$. We usually refer to o_X as the *input derivative*, and t_X as the *output derivative*. We say (X, d_X) is finite when X is.

Remark 4.8. Clearly, an automaton as defined above is precisely a coalgebra for the functor D given by $D(X) = 2 \times X^A$ (cf. Exercise 3.3).

We often denote $t_X(x)(\mathtt{a})$ with $(x)_\mathtt{a}$ when t_X is clear from context. Each $x \in X$ is a *state*, and when $o_X(x) = 1$ we call x *accepting*. We can draw an automaton as a graph where states are nodes (with accepting states circled twice) and transitions are labelled edges. For instance, we can depict the automaton $(X, \langle o_X, t_X \rangle)$ where $X = \{x_1, x_2\}$, and o_X and t_X are as given below on the right with the graph below on the left.



$$\begin{array}{rclrcl} o_X(x_1) & = & 0 & o_X(x_2) & = & 1 \\ (x_1)_\mathtt{a} & = & x_2 & (x_1)_\mathtt{b} & = & x_1 \\ (x_2)_\mathtt{a} & = & x_1 & (x_2)_\mathtt{b} & = & x_2 \end{array}$$

Remark 4.9. The reader may be more familiar with automata presented as tuples (Q, δ, F), where Q is a set of *states*, $F \subseteq Q$ holds the *accepting states*, and $\delta\colon Q \times A \to Q$ is called the *transition function*. We can convert between these presentations easily, by sending $(X, \langle o_X, t_X \rangle)$ to (X, δ, F), where $\delta(x, \mathsf{a}) = (x)_{\mathsf{a}}$, and $F = \{x \in X \mid \beta(x) = 1\}$; conversely, (Q, δ, F) can be turned into $(X, \langle o_X, t_X \rangle)$ by choosing $o_X(x) = 1$ when $x \in F$ and $o_X(x) = 0$ otherwise, while $(x)_{\mathsf{a}} = \delta(x, \mathsf{a})$.

The Brzozowski derivatives $o\colon \mathsf{Exp} \to 2$ and $t\colon \mathsf{Exp} \to \mathsf{Exp}^A$ equip Exp with an automaton structure, making $(\mathsf{Exp}, \langle o, t \rangle)$ an (infinite) automaton; we return to this in Section 4.2.3.

Another important example of an automaton is carried by the set of languages 2^{A^*}. For $\mathsf{a} \in A$, the input derivative l_{a} of $l \in 2^{A^*}$ on a is defined by $l_{\mathsf{a}}(w) = l(\mathsf{a}w)$. The output of l is defined by $l(\epsilon)$. Together, these define an automaton $(2^{A^*}, \langle o_L, t_L \rangle)$. These maps are sometimes called *semantic Brzozowski derivatives*, or *simply language derivatives*.

4.2.1 Morphisms and finality

A central concept in the dicussion to come is the idea of a *morphism*. Intuitively, this is a map that respects input and output derivatives.

Definition 4.10. Given two deterministic automata $(S, \langle o_S, t_S \rangle)$ and $(T, \langle o_T, t_T \rangle)$, a function $h\colon S \to T$ is a *homomorphism* if it commutes with the output and transition functions, that is $o_T(h(s)) = o_S(s)$, and $t_T(h(s))(\mathsf{a}) = h(t_S(s)(\mathsf{a}))$, for all $\mathsf{a} \in A$.

Remark 4.11. Morphisms between automata as defined above are precisely morphisms between D-coalgebras — that is, the equations above correspond to the commutativity of the diagram below.

$$\begin{tikzcd}
S \arrow[r, "h"] \arrow[d, "\langle o_S, t_S \rangle"'] & T \arrow[d, "\langle o_T, t_T \rangle"] \\
2 \times S^A \arrow[r, "D(h)"'] & 2 \times T^A
\end{tikzcd}$$

Given an automaton $(S, \langle o_S, t_S \rangle)$, the derivative s_{a} of a state s for input $\mathsf{a} \in A$ can be extended to the word derivative s_w of s for input

$w \in A^*$ by defining, by induction on w, $s_\epsilon = s$ and $s_{\mathtt{a}w'} = (s_\mathtt{a})_{w'}$. This enables an easy definition of the semantics of a state s of a deterministic automaton: the language $L(s) \in 2^{A^*}$ recognized by s is given by:

$$L(s)(w) = o_S(s_w) \tag{4.1}$$

Example 4.12. The language recognized by state s_1 of the automaton on page 40 is the set of all words with an odd number of a's. It is easy to check that, for example, $L(s_1)(\mathtt{bab}) = o_S((s_1)_{\mathtt{bab}}) = o_S(s_2) = 1$.

L is defined from the state space of any automaton to 2^{A^*}. This function takes a very special place as a particular morphism that relates every automaton to the automaton formed by language derivatives.

Theorem 4.13 (Finality)**.** For any automaton $(S, \langle o_S, t_S\rangle)$, $L\colon S \to 2^{A^*}$ is the *only* homomorphism from $(S, \langle o_S, t_S\rangle)$ to $(2^{A^*}, \langle o_L, t_L\rangle)$.

Remark 4.14. Thus $(2^{A^*}, \langle o_L, t_L\rangle)$ forms the final D-coalgebra.

Proof. We have to prove that the diagram commutes and that L is unique. First the commutativity with the derivatives:

$$\begin{aligned} o_L(L(s)) &= L(s)(\epsilon) = o_S(s) \\ t_L(L(s))(\mathtt{a})(w) &= L(s)_\mathtt{a}(w) = L(s)(\mathtt{a}w) \\ &= L(s_\mathtt{a})(w) = L(t_S(s))(\mathtt{a})(w) \end{aligned}$$

For the one but last step, note that, by definition of L (see (4.1)):

$$L(s)(\mathtt{a}w) = o_S(s_{\mathtt{a}w}) = o_S((s_\mathtt{a})_w) = L(s_\mathtt{a})(w)$$

For the uniqueness, suppose there is a morphism $h\colon S \to 2^{A^*}$ such that, for every $s \in S$ and $\mathtt{a} \in A$, $o_L(h(s)) = o_S(s)$ and $h(s)_\mathtt{a} = h(s_\mathtt{a})$.

We prove by induction on $w \in A^*$ that $h(s)(w) = L(s)(w)$:

$$h(s)(\epsilon) = o_L(h(s)) = o_S(s) = L(s)(\epsilon)$$

$$h(s)(\mathtt{a}w) = (h(s)_\mathtt{a})(w) = h(s_\mathtt{a})(w) \overset{\text{(IH)}}{=} L(s_\mathtt{a})(w) = L(s)(\mathtt{a}w) \qquad \square$$

On its own, Theorem 4.13 may seem abstract. Its most immediate consequence is that we can relate homomorphisms to the language semantics of automata, by exploiting uniqueness.

Corollary 4.15. Given automata $(S, \langle o_S, t_S \rangle)$ and $(T, \langle o_T, t_T \rangle)$ with a morphism $h \colon S \to T$, it holds for $s \in S$ that $L(s) = L(h(s))$.

Proof. Follows from Theorem 4.13 and the fact that $L \colon S \to 2^{A^*}$ and $L \circ h \colon S \to 2^{A^*}$ are morphisms from $(S, \langle o_S, t_S \rangle)$ to $(2^{A^*}, \langle o_L, t_L \rangle)$. $\square$

4.2.2 Bisimulations and coinduction

A moment ago, we saw how states related by homomorphisms must have the same language. Unfortunately, the converse is not necessarily true: there exist automata with states that have the same language, but which are not related by any homomorphism (see Exercise 4.2). To get more mileage, we need a slightly more general notion, as follows.

Definition 4.16 (Bisimulation)**.** Given $(S, \langle o_S, t_S \rangle)$ and $(T, \langle o_T, t_T \rangle)$, we say that a relation $R \subseteq S \times T$ is a *bisimulation* if $\langle s, t \rangle \in R$ implies

$$o_S(s) = o_T(t) \quad \text{and} \quad (s_{\mathtt{a}}, t_{\mathtt{a}}) \in R \text{ for all } \mathtt{a} \in A$$

We write $s \sim t$ whenever there exists a bisimulation R containing (s, t).

Remark 4.17. Unsurprisingly, bisimulations between automata as defined above are in one-to-one correspondence with bisimulations between D-coalgebras — the conditions above guarantee the existence of a D-coalgebra structure on R, and vice versa.

The theorem guarantees that bisimularity is a sound and complete proof principle for language equivalence of deterministic automata.

Theorem 4.18 (Coinduction principle)**.** Given two deterministic automata $(S, \langle o_S, t_S \rangle)$ and $(T, \langle o_T, t_T \rangle)$ with $s \in S$ and $t \in T$:

$$s \sim t \iff L(s) = L(t)$$

Remark 4.19. Coalgebraically, the claim above follows immediately from Theorem 4.13, which tells us that L is the morphism into the final D-coalgebra, and the fact that D is kernel-compatible (Exercise 3.15), which we can then use to invoke Lemma 3.20 and Corollary 3.27.

Proof. For the forward implication, let R be a bisimulation. We prove, by induction on $w \in A^*$, that, for all $(s,t) \in R$, $w \in L(s) \Leftrightarrow w \in L(t)$.

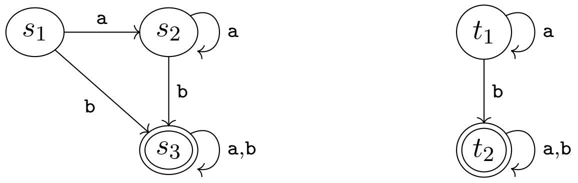
$$\begin{aligned}
&L(s)(\epsilon) = o_S(s) \overset{s\sim t}{=} o_T(t) = L(t)(\epsilon)\\
&L(s)(\mathtt{a}w') = o_S(s_{\mathtt{a}w'}) = o_S((s_\mathtt{a})_{w'}) = o_T((t_\mathtt{a})_{w'}) = L(t)(\mathtt{a}w')
\end{aligned}$$

The penultimate step follows by induction and the fact that $(s_\mathtt{a}, t_\mathtt{a}) \in R$.

The converse follows by the observation that the relation $\{(s,t) \mid L(s) = L(t)\}$ is a bisimulation; we leave the details to Exercise 4.4. □

The upshot of Theorem 4.18 is that we can use coinduction to determine whether two states s and t of two deterministic automata $(S, \langle o_S, t_S\rangle)$ and $(T, \langle o_T, t_T\rangle)$ recognize the same language: it is enough to construct a bisimulation containing (s,t) — indeed, if such a bisimulation does not exist, then the states *cannot* have the same language.

Example 4.20 (Coinduction). Let $(S, \langle o_S, t_S\rangle)$ and $(T, \langle o_T, t_T\rangle)$ be the deterministic automata over the alphabet $\{\mathtt{a}, \mathtt{b}\}$ given by



The relation $R = \{\langle s_1, t_1\rangle, \langle s_2, t_1\rangle, \langle s_3, t_2\rangle\}$ is a bisimulation:

$$\begin{array}{ll}
o_S(s_1) = o_S(s_2) = 0 = o_T(t_1) & o_S(s_3) = 1 = o_T(t_2)\\
(s_1)_\mathtt{a} = s_2 \; R \; t_1 = (t_1)_\mathtt{a} & (s_1)_b = s_3 \; R \; t_2 = (t_1)_\mathtt{b}\\
(s_2)_\mathtt{a} = s_2 \; R \; t_1 = (t_1)_\mathtt{a} & (s_2)_b = s_3 \; R \; t_2 = (t_1)_\mathtt{b}\\
(s_3)_\mathtt{a} = s_3 \; R \; t_2 = (t_2)_\mathtt{a} & (s_3)_\mathtt{b} = s_3 \; R \; t_2 = (t_2)_\mathtt{b}
\end{array}$$

Thus, by Theorem 4.18, $L(s_1) = L(t_1) = L(s_2)$ and $L(s_3) = L(t_3)$.

4.2.3 Soundness of Brzozowski derivatives

Recall that $(\mathsf{Exp}, \langle o, t\rangle)$ is an automaton and thus, by Theorem 4.13, we have a unique map $L\colon \mathsf{Exp} \to 2^{A^*}$ that acts as a morphism between $(\mathsf{Exp}, \langle o, t\rangle)$ and $(2^{A^*}, \langle o_L, t_L\rangle)$. Hence, L provides an alternative *operational* semantics for regular expressions. We now prove that, for any $e \in \mathsf{Exp}$, this semantics coincides with the one from Definition 2.8.

Theorem 4.21. For all $e \in \mathsf{Exp}$ and $w \in A^*$, $[\![e]\!] = L(e)$. That is,

$$w \in [\![e]\!] \Leftrightarrow w \in L(e)$$

Proof. By Theorem 4.13, it suffices to show that $[\![-]\!]$ acts like a morphism from $(\mathsf{Exp}, \langle o, t\rangle)$ to $(2^{A^*}, \langle o_L, t_L\rangle)$. Concretely, this comes down to proving that for all $e \in \mathsf{Exp}$, it holds that $o_L([\![e]\!]) = o(e)$, and furthermore that for $\mathtt{a} \in A$ we have that $[\![e_\mathtt{a}]\!] = [\![e]\!]_\mathtt{a}$.

For the first equality, we derive using Theorem 4.2 that:

$$o_L([\![e]\!]) = o_L([\![o(e) + \sum_{\mathtt{a}\in A} \mathtt{a}e_\mathtt{a}]\!]) = o_L([\![o(e)]\!]) = o(e)$$

In the second-to-last step, we use that for $b \in \{0,1\}$, it holds that $o_L([\![b]\!]) = b$. For the second equality, we again use Theorem 4.2:

$$[\![e]\!]_\mathtt{a} = [\![o(e) + \sum_{\mathtt{b}\in A} \mathtt{b}e_\mathtt{a}]\!]_\mathtt{a} = [\![\mathtt{a}e_\mathtt{a}]\!]_\mathtt{a} = [\![e_\mathtt{a}]\!] \qquad \square$$

Thus, the denotational semantics of regular expressions coincides with the operational semantics. Furthermore, because bisimulations are sound and complete for deciding language equivalence of states in automata, we can leverage this correspondence to decide equivalence of expressions. The construction of bisimulations can be mechanized and forms the basis of efficient decision procedures for language equivalence.

Brzozowski derivatives can be used to incrementally build a bisimulation between regular expressions: in contrast to other methods to build automata from expressions the automaton can be built "on-the-fly" and, furthermore, states can be simplified soundly using axioms of Kleene algebra leading to the construction of bisimulations up-to congruence. We refer to Chapter 7 for pointers to further reading on these topics.

4.3 Kleene's theorem

In this section, we present Kleene's theorem, which states the equivalence between the class of languages recognized by finite deterministic automata (here, "finite" means that the set of states S and the input alphabet A are finite) and the one denoted by regular expressions.

Theorem 4.22 (Kleene's Theorem)**.** For any $l \in 2^{A^*}$, $l = [\![e]\!]$, for some regular expression $e \in \mathsf{Exp}$ if and only if $l = L(s)$ for a state $s \in S$ of a finite deterministic automaton $(S, \langle o_S, t_S \rangle)$.

The proof of this theorem amounts to constructing an equivalent regular expression from a state of a deterministic automaton and, conversely, to constructing a deterministic automaton which has a state that is equivalent to a given regular expression. We will sketch these constructions in the next two sections.

4.3.1 Automata to expressions

To present the construction of a regular expression from a state of a finite deterministic automaton we build a system of equations on expressions and then solve it. Let $(S, \langle o_S, t_S \rangle)$ be a finite deterministic automaton. We associate with each $s \in S$ a variable x_s and an equation

$$x_s \equiv \sum_{\mathsf{a} \in A} \mathsf{a} x_{s_\mathsf{a}} + o_S(s) \tag{4.2}$$

The sum is well defined because A is finite. In this way, we build a system with n equations and n variables, where n is the size of S.

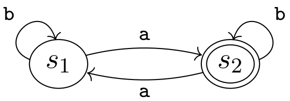
Definition 4.23 (Solutions to systems of equations)**.** Consider a system of equations $x_s \equiv \sum_{\mathsf{a} \in A} \mathsf{a} x_{s_\mathsf{a}} + o_S(s)$, $s \in S$ as in Equation (4.2). We define a solution to this system as a mapping that assigns to each variable x_s an expression $e_s \in \mathsf{Exp}$ (over A) satisfying $e_s \equiv t_s[e_s/x_s]$, that is, $e_s \equiv \sum_{\mathsf{a} \in A} \mathsf{a} e_{s_\mathsf{a}} + o_S(s)$.

Observe that if e_s is a solution to $x_s \equiv t_s$ and if $t_s \equiv t'_s$, then e_s is also a solution to $x_s \equiv t'_s$. Moreover, if the equation is of the shape $x \equiv tx + c$, then t^*c is the (unique) solution to the equation. This can be used to iteratively solve a system, see below for an example.

Our definition of solution is sound, in the sense that every solution is necessarily language equivalent to the corresponding state. This can be shown directly (see Exercise 4.5), but a rather elegant proof by coinduction (c.f. Section 4.2 and Exercise 4.6) is also possible.

We will now illustrate the construction of solutions with an example, and then we will present an alternative definition of the solution of a system of equations of the same form as (4.2) using matrices.

Example 4.24. Consider the following deterministic automaton over the alphabet $A = \{\mathtt{a}, \mathtt{b}\}$:



Let e_1 and e_2 denote e_{s_1} and e_{s_2}, respectively. The system of equations arising from the automaton is the following:

$$e_1 \equiv (\mathtt{a}e_2 + \mathtt{b}e_1) + 0 \qquad\qquad e_2 \equiv (\mathtt{a}e_1 + \mathtt{b}e_2) + 1$$

Using the laws of Kleene algebra we can rewrite the second equation to $e_2 \equiv (\mathtt{a}e_1 + 1) + \mathtt{b}e_2$ and then solve it, which yields $e_2 \equiv \mathtt{b}^*(\mathtt{a}e_1 + 1)$ or, simplified, $e_2 \equiv \mathtt{b}^*\mathtt{a}e_1 + \mathtt{b}^*$. We then replace e_2 in the first equation:

$$e_1 \equiv (\mathtt{a}(\mathtt{b}^*\mathtt{a}e_1 + \mathtt{b}^*) + \mathtt{b}e_1) + 0 \equiv (\mathtt{ab}^*\mathtt{a} + \mathtt{b})e_1 + \mathtt{ab}^*$$

Solving it and then replacing it in the equation for e_2 results in the following two expressions

$$e_1 \equiv (\mathtt{ab}^*\mathtt{a} + \mathtt{b})^*\mathtt{ab}^* \qquad e_2 \equiv \mathtt{b}^*\mathtt{a}(\mathtt{ab}^*\mathtt{a} + \mathtt{b})^*\mathtt{ab}^* + \mathtt{b}^*$$

which satisfy $[\![e_1]\!] = L(s_1)$ and $[\![e_2]\!] = L(s_2)$.

The methodology of the last example generalizes to larger automata, but can become tedious to work out by hand (we will return to it in Chapter 5). For now, we present an alternative general method to construct a solution to a system of n equations of the form

$$x_i \equiv \mathtt{a}_{i1}x_1 + \ldots + \mathtt{a}_{in}x_n + o_i \qquad 1 \leq i \leq n$$

We recall from [10] a matrix formalization of the above system and its solution. Given a system of n equations as above, construct an $n \times n$ matrix T with all the $\mathtt{a}_{ij}$, an $n \times 1$ vector O with all the o_i and an $n \times 1$ vector X with all the x_i, obtaining the matrix-vector equation

$$X = TX + O \tag{4.3}$$

Moreover, we define, for an $n \times n$ matrix R, with entries $e \in \mathsf{Exp}$, a matrix R^*, by induction on n. For $n = 1$, the matrix R has a single entry e and $R^* = e^*$. If $n > 1$, we divide the matrix into four submatrices

$$R = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

with A and D are square with dimensions, respectively, $m \times m$ (for some $m < n$) and $(n-m) \times (n-m)$. By the induction hypothesis, we can use A^*, B^*, C^* and D^*. This allows us to define

$$R^* = \begin{pmatrix} (A+BD^*C)^* & (A+BD^*C)^*BD^* \\ (D+CA^*B)^*CA^* & (D+CA^*B)^* \end{pmatrix}$$

Now, one can show that the vector T^*O is the least (with respect to the pointwise order $\leqq$ of regular expressions) solution to the system in (4.3) [11, Theorem A.3].

Example 4.25. Let us revisit Example 4.24 using this method. First, we construct the matrices X, T and O:

$$X = \begin{pmatrix} e_1 \\ e_2 \end{pmatrix} \qquad T = \begin{pmatrix} \mathtt{b} & \mathtt{a} \\ \mathtt{a} & \mathtt{b} \end{pmatrix} \qquad O = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Now we compute T^*

$$T^* = \begin{pmatrix} (\mathtt{b}+\mathtt{ab}^*\mathtt{a})^* & (\mathtt{b}+\mathtt{ab}^*\mathtt{a})^*\mathtt{ab}^* \\ (\mathtt{b}+\mathtt{ab}^*\mathtt{a})^*\mathtt{ab}^* & (\mathtt{b}+\mathtt{ab}^*\mathtt{a})^* \end{pmatrix}$$

and T^*O

$$T^*O = \begin{pmatrix} ((\mathtt{b}+\mathtt{ab}^*\mathtt{a})^*)0 + ((\mathtt{b}+\mathtt{ab}^*\mathtt{a})^*\mathtt{ab}^*)1 \\ ((\mathtt{b}+\mathtt{ab}^*\mathtt{a})^*\mathtt{ab}^*)0 + ((\mathtt{b}+\mathtt{ab}^*\mathtt{a})^*)1 \end{pmatrix} \equiv \begin{pmatrix} (\mathtt{b}+\mathtt{ab}^*\mathtt{a})^*\mathtt{ab}^* \\ (\mathtt{b}+\mathtt{ab}^*\mathtt{a})^* \end{pmatrix}$$

In the last step we used some of the Kleene algebra equations to simplify regular expressions. Now note that the expression computed for e_1 is (almost) the same as the one we obtained in Example 4.24, whereas the one for e_2 is quite different (syntactically), but still equivalent.

4.3.2 Expressions to automata

Given a regular expression e, we want to construct a finite deterministic automaton with the same language. We have shown that Exp, the set of regular expressions over A, carries an automaton structure given by Brzozowski derivatives. Hence, an obvious way of constructing an automaton with a state equivalent to e is to carve out all states reachable from e in the Brzozowski coalgebra $\langle o, t \rangle$. However, even for very simple expressions, this may still yield an infinite automaton. Take, for example, $(\mathtt{a}^*)^*$; computing input derivatives for $\mathtt{a}$ yields:

$$\begin{aligned}
((\mathtt{a}^*)^*)_{\mathtt{a}} &= (1\mathtt{a}^*)(\mathtt{a}^*)^* \\
((1\mathtt{a}^*)(\mathtt{a}^*)^*)_{\mathtt{a}} &= (0\mathtt{a}^* + 1\mathtt{a}^*)(\mathtt{a}^*)^* + (1\mathtt{a}^*)(\mathtt{a}^*)^* \\
((0\mathtt{a}^* + 1\mathtt{a}^*)(\mathtt{a}^*)^* + (1\mathtt{a}^*)(\mathtt{a}^*)^*)_{\mathtt{a}} &= ((0\mathtt{a}^* + 0\mathtt{a}^* + 1\mathtt{a}^*)(\mathtt{a}^*)^* \\
&\quad + (1\mathtt{a}^*)(\mathtt{a}^*)^*) + ((1\mathtt{a}^*)(\mathtt{a}^*)^*)_{\mathtt{a}} \\
&\vdots
\end{aligned}$$

All derivatives above are equivalent, but they are syntactically different and therefore they will not be identified as denoting the same state, which results in $\langle(\mathtt{a}^*)^*\rangle$ having an infinite state space. To achieve finiteness, it is enough to remove double occurrences of expressions e in sums of the form $\ldots + e + \ldots + e + \ldots$. This uses the laws for associativity, commutativity and idempotence of $+$ (ACI in brief), but note that it does not amount to taking the quotient of Exp w.r.t. the equivalence induced by ACI, which would also guarantee finiteness [8], [12] but would identify more expressions. For instance, the expressions $\mathtt{a}+\mathtt{b}$ and $\mathtt{b}+\mathtt{a}$ for $\mathtt{a},\mathtt{b} \in A$ with $\mathtt{a} \neq \mathtt{b}$ will not be identified in our procedure.

The following proposition provides a syntactic characterization of the derivative e_w, for a word $w \in A^+$ and $e \in \mathsf{Exp}$.

Proposition 4.26. Let $w \in A^*$ and $e, e_1, e_2 \in \mathsf{Exp}$. Then, the word derivatives of $(e_1 + e_2)$, e_1e_2 and e^* are of the form:

$$\begin{aligned}
(e_1 + e_2)_w &= (e_1)_w + (e_2)_w \\
(e_1e_2)_w &= (e_1)_{w_1}e_2 + \cdots + (e_1)_{w_k}e_2 + (e_2)_{w'_1} + \cdots + (e_2)_{w'_l} \\
(e^*)_w &= e_{w_1}e^* + \cdots + e_{w_m}e^*
\end{aligned}$$

for some fixed $k, l, m \geq 0$, $w_i, w'_i \in A^*$.

Proof. By induction on the length of $w \in A^*$. For $w = \epsilon$, the equalities hold trivially. For $w = \mathtt{a}w'$, we have:

$$(e_1 + e_2)_{\mathtt{a}w'} = ((e_1)_{\mathtt{a}} + (e_2)_{\mathtt{a}})_{w'} \overset{\text{(IH)}}{=} (e_1)_{\mathtt{a}w'} + (e_2)_{\mathtt{a}w'}$$

$$(e_1e_2)_{\mathtt{a}w'} = \begin{cases} ((e_1)_{\mathtt{a}}e_2)_{w'} & \text{if } o(e_1) = 0 \\ ((e_1)_{\mathtt{a}}e_2)_{w'} + (e_2)_{\mathtt{a}w'} & \text{otherwise} \end{cases}$$

Both cases now follow by induction. If $o(e_1) = 0$, we have

$$((e_1)_{\mathtt{a}} e_2)_{w'} \overset{\text{(IH)}}{=} (e_1)_{\mathtt{a}w_1} e_2 + \cdots + (e_1)_{\mathtt{a}w_k} e_2 + (e_2)_{w'_1} + \cdots + (e_2)_{w'_l}$$

Otherwise,

$$((e_1)_{\mathtt{a}} e_2)_{w'} + (e_2)_{\mathtt{a}w'} \overset{\text{(IH)}}{=} (e_1)_{\mathtt{a}w_1} e_2 + \cdots + (e_1)_{\mathtt{a}w_k} e_2 + (e_2)_{w'_1} + \cdots + (e_2)_{w'_l} + (e_2)_{\mathtt{a}w'}$$

Finally, for the third equality, we need to use the result we have just proved for sequential composition:

$$\begin{aligned}(e^*)_{\mathtt{a}w'} &= (e_{\mathtt{a}} e^*)_{w'} = e_{\mathtt{a}w_1} e^* + \cdots + e_{\mathtt{a}w_k} e^* + (e^*)_{w'_1} + \cdots + (e^*)_{w'_l} \\ &\overset{\text{(IH)}}{=} e_{u_1} e^* + \cdots + e_{u_m} e^* \end{aligned}$$ □

Using Proposition 4.26, we can now prove the following.

Theorem 4.27. Let $e \in \mathsf{Exp}$. The number of word derivatives e_w is finite, as long as double occurrences of expressions e_1 in sums of the form $\ldots + e_1 + \ldots + e_1 + \ldots$ are removed.

Proof. By induction on the structure of the expression. The base cases are trivial: they have either one, two or three distinct word derivatives. The sum $e_1 + e_2$ has as upper-bound for the number of derivatives, once we remove the double occurrences in the sum, $N \times M$; the concatenation $e_1 e_2$ has as bound 2^{N+M}; and the star $(e_1)^*$ has 2^N, where N and M are the inductive bounds for e_1 and e_2. □

We now illustrate the construction of an automaton using the procedure described above.

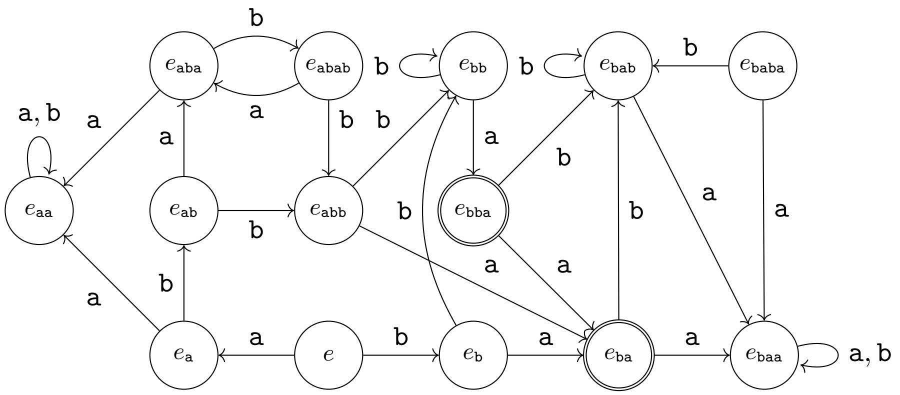
Example 4.28. Let $A = \{\mathtt{a}, \mathtt{b}\}$ and $e = (\mathtt{ab} + \mathtt{b})^*\mathtt{ba}$. We compute derivatives incrementally, marking with ✓ the derivatives that are not new and numbering the new ones:

$$\begin{aligned} e_\epsilon &= (\mathtt{ab} + \mathtt{b})^*\mathtt{ba} && \text{①} \\ e_{\mathtt{a}} &= (1\mathtt{b} + 0)(\mathtt{ab} + \mathtt{b})^*\mathtt{ba} + 0\mathtt{a} && \text{②} \\ e_{\mathtt{b}} &= (0\mathtt{b} + 1)(\mathtt{ab} + \mathtt{b})^*\mathtt{ba} + 1\mathtt{a} && \text{③} \end{aligned}$$

$$
\begin{array}{lll}
e_{\mathtt{aa}} & = (0\mathtt{b}+0)(\mathtt{ab}+\mathtt{b})^*\mathtt{ba}+0\mathtt{a} & \textcircled{4}\\
e_{\mathtt{ab}} & = (0\mathtt{b}+1+0)(\mathtt{ab}+\mathtt{b})^*\mathtt{ba}\quad +0\mathtt{a} & \textcircled{5}\\
e_{\mathtt{ba}} & = (0\mathtt{b}+0)(\mathtt{ab}+\mathtt{b})^*\mathtt{ba}+(1\mathtt{b}{+}0)(\mathtt{ab}+\mathtt{b})^*\mathtt{ba} & \\
 & \qquad\qquad\qquad\qquad +0\mathtt{a}+0\mathtt{a}+1 = e_{\mathtt{aa}}+e_{\mathtt{a}}+1 & \textcircled{6}\\
e_{\mathtt{bb}} & = (0\mathtt{b}+0)(\mathtt{ab}+\mathtt{b})^*\mathtt{ba}+(0\mathtt{b}{+}1)(\mathtt{ab}+\mathtt{b})^*\mathtt{ba} & \\
 & \qquad\qquad\qquad\qquad +1\mathtt{a}+0\mathtt{a}+0 = e_{\mathtt{aa}}+e_{\mathtt{b}}+0 & \textcircled{7}\\
e_{\mathtt{aaa}} & = (0\mathtt{b}+0)(\mathtt{ab}+\mathtt{b})^*\mathtt{ba}+0\mathtt{a} = e_{\mathtt{aa}} & \checkmark\\
e_{\mathtt{aab}} & = (0\mathtt{b}+0)(\mathtt{ab}+\mathtt{b})^*\mathtt{ba}+0\mathtt{a} = e_{\mathtt{aa}} & \checkmark\\
e_{\mathtt{aba}} & = (0\mathtt{b}+0)(\mathtt{ab}+\mathtt{b})^*\mathtt{ba}+(1\mathtt{b}{+}0)(\mathtt{ab}+\mathtt{b})^*\mathtt{ba}) & \\
 & \qquad\qquad\qquad\qquad +0\mathtt{a}+0\mathtt{a} = e_{\mathtt{aa}}+e_{\mathtt{a}} & \textcircled{8}\\
e_{\mathtt{abb}} & = (0\mathtt{b}+0)(\mathtt{ab}+\mathtt{b})^*\mathtt{ba}+(0\mathtt{b}{+}1)(\mathtt{ab}+\mathtt{b})^*\mathtt{ba} & \\
 & \qquad\qquad\qquad\qquad +1\mathtt{a}+0\mathtt{a} = e_{\mathtt{aa}}+e_{\mathtt{b}} & \textcircled{9}\\
e_{\mathtt{baa}} & = e_{\mathtt{aa}}+0 & \textcircled{10}\\
e_{\mathtt{bab}} & = e_{\mathtt{aa}}+e_{\mathtt{ab}}+0 & \textcircled{11}\\
e_{\mathtt{bba}} & = e_{\mathtt{aa}}+e_{\mathtt{ba}}+0 = e_{\mathtt{aa}}+e_{\mathtt{a}}\quad +1+0 & \textcircled{12}\\
e_{\mathtt{bbb}} & = e_{\mathtt{aa}}+e_{\mathtt{bb}}+0 = e_{\mathtt{bb}} & \checkmark\\
e_{\mathtt{abaa}} & = e_{\mathtt{aa}} & \checkmark\\
e_{\mathtt{abab}} & = e_{\mathtt{aa}}+e_{\mathtt{ab}} & \textcircled{13}\\
e_{\mathtt{abba}} & = e_{\mathtt{aa}}+e_{\mathtt{ba}} = e_{\mathtt{ba}} & \checkmark\\
e_{\mathtt{abbb}} & = e_{\mathtt{aa}}+e_{\mathtt{bb}} = e_{\mathtt{bb}} & \checkmark\\
e_{\mathtt{baaa}} & = e_{\mathtt{baa}} & \checkmark\\
e_{\mathtt{baab}} & = e_{\mathtt{baa}} & \checkmark\\
e_{\mathtt{baba}} & = e_{\mathtt{aa}}+e_{\mathtt{aba}}+0 = e_{\mathtt{aa}}+e_{\mathtt{a}}\ \ +0 & \textcircled{14}\\
e_{\mathtt{babb}} & = e_{\mathtt{aa}}+e_{\mathtt{abb}}+0 = e_{\mathtt{aa}}+e_{\mathtt{b}}\ \ +0 = e_{\mathtt{bb}} & \checkmark\\
e_{\mathtt{bbaa}} & = e_{\mathtt{aa}}+0 = e_{\mathtt{baa}} & \checkmark\\
e_{\mathtt{bbab}} & = e_{\mathtt{aa}}+e_{\mathtt{ab}}+0 = e_{\mathtt{bab}} & \checkmark\\
e_{\mathtt{ababa}} & = e_{\mathtt{aa}}+e_{\mathtt{aba}} = e_{\mathtt{aba}} & \checkmark\\
e_{\mathtt{ababb}} & = e_{\mathtt{aa}}+e_{\mathtt{abb}} = e_{\mathtt{abb}} & \checkmark\\
e_{\mathtt{babaa}} & = e_{\mathtt{aa}}+0 = e_{\mathtt{baa}} & \checkmark
\end{array}
$$

$e_{\mathtt{babab}} = e_{\mathtt{aa}} + e_{\mathtt{ab}} + 0 = e_{\mathtt{bab}}$ ✓

The resulting automaton will have 14 states (note also that only $e_{\mathtt{ba}}$ and $e_{\mathtt{bba}}$ have output value 1):



Here, we strictly followed the procedure described above. The goal was to construct a finite deterministic automaton with a state equivalent to e and we did not worry about its size. It is however obvious that, by allowing the simplification of certain expressions, the resulting automaton could be much smaller. For instance, let us add the following three rules: $0e$ can be replaced by 0; $1e$ can be replaced by e and 0 can be eliminated from sums. The calculations for e would then be:

$e_\epsilon = (\mathtt{ab} + \mathtt{b})^*\mathtt{ba}$ ①

$e_{\mathtt{a}} = \mathtt{b}(\mathtt{ab} + \mathtt{b})^*\mathtt{ba}$ ②

$e_{\mathtt{b}} = (\mathtt{ab} + \mathtt{b})^*\mathtt{ba} + \mathtt{a}$ ③

$e_{\mathtt{aa}} = 0$ ④

$e_{\mathtt{ab}} = (\mathtt{ab} + \mathtt{b})^*\mathtt{ba} = e$ ✓

$e_{\mathtt{ba}} = \mathtt{b}(\mathtt{ab} + \mathtt{b})^*\mathtt{ba} + 1$ ⑤

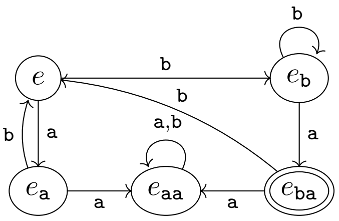
$e_{\mathtt{bb}} = (\mathtt{ab} + \mathtt{b})^*\mathtt{ba} + \mathtt{a} = e_{\mathtt{b}}$ ✓

$e_{\mathtt{aaa}} = e_{\mathtt{baa}} = 0 = e_{\mathtt{aa}}$ ✓

$e_{\mathtt{aab}} = e_{\mathtt{aa}}$ ✓

$e_{\mathtt{bab}} = (\mathtt{ab} + \mathtt{b})^*\mathtt{ba} = e$ ✓

The resulting automaton would then have 5 states:



In fact, this is the automaton with a minimal number of states recognizing the language denoted by e. Note that the axioms we considered to simplify e are not always enough to get the minimal automaton. For that one would have to consider the complete set of axioms, which we will present later in this chapter. However, in many cases the automaton computed using just a subset of the axioms is minimal: an empirical study about this phenomenon appears in [13].

With the constructions above from deterministic automata to regular expressions, and vice versa, the proof of Kleene's theorem is now straightforward.

Proof of Theorem 4.22 (Kleene's Theorem). Suppose $l = L(e)$ for some regular expression $e \in \mathsf{Exp}$. We construct an automaton $(S, \langle o_S, t_S\rangle)$ as described above and since $e \in S$ and o and t are just the restrictions of o and t to S we have that e, considered as a state in S, is equivalent to e, considered as a state in the coalgebra $(\mathsf{Exp}, \langle o, t\rangle)$.

For the converse, suppose $l = L(s)$, for state $s \in S$ of a finite deterministic automaton $(S, \langle o_S, t_S\rangle)$. We solve a system of equations as described above in Section 4.3.1 and the result is an expression e_s equivalent to s, that is $[\![e_s]\!] = L(s)$. $\square$

Remark 4.29. There are, of course, many more ways to construct an equivalent automaton from a regular expression. We focused on Brzozowski's construction here because it works well with the coalgebraic perspective, but you may be familiar with the syntax-directed translation proposed by Thompson [14]. Other constructions can be found in [15], [16], [17], [18]; we refer to [19] for an excellent overview.

To summarize, in this chapter we instantiated the coalgebraic framework to deterministic automata, obtaining a decidable notion of *bisimulation* that characterizes language equivalence (Section 4.2). We then showed that every regular expression can be converted into an equivalent deterministic automaton, meaning equivalence of regular expressions can be checked (Section 4.3.1). Finally, we argued that every automaton can be converted back to an expression (Section 4.3.2).

4.4 Exercises

4.1. Create an automaton for each of the following languages over $A = \{\mathtt{a}, \mathtt{b}\}$:

(a) The language L_0 of words where every third letter is an $\mathtt{a}$, unless it is preceded by a $\mathtt{b}$. Examples of words in L_0 include $\mathtt{aaa}$, ϵ, $\mathtt{b}$, and $\mathtt{bbb}$. Examples of words *not* in L_0 include $\mathtt{aab}$ and $\mathtt{aaabab}$.

(b) The language L_1 of words that do not contain three subsequent $\mathtt{a}$'s or $\mathtt{b}$'s. Examples of words in L_1 include ϵ, $\mathtt{ab}$ and $\mathtt{abaa}$. Examples of words *not* in L_1 include $\mathtt{aaa}$ and $\mathtt{abbba}$.

4.2. Demonstrate two automata $(S, \langle o_S, t_S \rangle)$ and $(T, \langle o_T, t_T \rangle)$ with states $s \in S$ and $t \in T$ such that $L(s) = L(t)$, but there *cannot* be a morphism between these automata that relates s to t. Argue in detail *why* such a morphism cannot exist.

4.3. Consider the automata drawn below:

$$\begin{tikzcd}
x \arrow[r, "{\mathtt{a}, \mathtt{b}}"] & y \arrow[r, "{\mathtt{a}, \mathtt{b}}"] & z \arrow[loop below, "{\mathtt{a}, \mathtt{b}}"] & u \arrow[r, "\mathtt{a}"] \arrow[rr, bend right, "\mathtt{b}"'] & v \arrow[r, bend left, "{\mathtt{a}, \mathtt{b}}"] & w \arrow[l, "{\mathtt{a}, \mathtt{b}}"]
\end{tikzcd}$$

Prove that x and u are bisimilar.

4.4. We return to the claim at the end of Theorem 4.18.

(a) Given two automata $(S, \langle o_S, t_S \rangle)$ and $(T, \langle o_T, t_T \rangle)$ and a morphism $h \colon S \to T$. Prove that the relation $\{(s, s') \in S \times S \mid h(s) = h(s')\}$ is a bisimulation.

(b) Given two automata $(S, \langle o_S, t_S \rangle)$ and $(T, \langle o_T, t_T \rangle)$. Prove that the relation $\{(s, t) \in S \times T \mid L(s) = L(t)\}$ is a bisimulation.

4.5. Let $(S, \langle o_S, t_S \rangle)$ be an automaton, and suppose that for each $s \in S$ we have an expression e_s, such that these expressions together form a solution to this automaton (c.f. Definition 4.23).

(a) Show that for $s \in S$, $o_S(s) = 1$ if and only if $o(e_s) = 1$.

(b) Show that for $s \in S$, $w \in A^*$ and $\mathtt{a} \in A$, we have that $\mathtt{a}w \in [\![e_s]\!]$ if and only if $w \in [\![e_{s_a}]\!]$.

(c) Use the two facts above to show that, for all $w \in A^*$ and $s \in S$, we have that $w \in L(s)$ if and only if $w \in [\![e_s]\!]$. In your proof, make sure you state clearly what claim you are proving by induction (and pay attention to the quantifiers).

4.6. We revisit Exercise 4.5 from a coalgebraic perspective. Let $(S, \langle o_S, t_S \rangle)$ be an automaton, and suppose that for $s \in S$ we have $e_s \in \mathsf{Exp}$ such that these expressions form a solution. Now consider the relation

$$R = \{\langle e, s \rangle \in \mathsf{Exp} \times S \mid [\![e]\!] = [\![e_s]\!]\}$$

Show that R is a bisimulation between $(S, \langle o_S, t_S \rangle)$ and the syntactic Brzozowski automaton, and use this to conclude that $[\![e_s]\!] = L(s)$ for all $s \in S$.

4.7. Use Brzozowski's construction to build an automaton for the expression $\mathtt{b}^*\mathtt{a}$.

4.8. Construct the Brzozowski automaton for $e = \mathtt{ab}^*$ and $f = \mathtt{a} + \mathtt{ab} + \mathtt{abb}^*$, and show that the states for e and f are bisimilar.

5

Completeness

In the previous chapters, we presented regular expressions, their denotational and operational semantics, and a set of laws to reason about their equivalence. In this chapter, we want to prove that the laws of Kleene Algebra are enough to reason about any equivalence of regular expressions. That is, given two regular expressions e and f that denote the same regular language, we can prove that $e \equiv f$. This property is called *completeness* with respect to the semantic model. The goal of this chapter is therefore to show that the laws of Kleene Algebra are *complete* for the language model. We will use that language equivalence and bisimilarity coincide for regular expressions, and show:

Theorem 5.1 (Soundness and completeness)**.** Let $e, f \in \mathsf{Exp}$; now

$$e \equiv f \iff e \sim f$$

The forward implication, called *soundness*, says that all equivalences proved using the laws of KA are in fact between bisimilar expressions. The converse, called *completeness*, says that all true equations between regular expressions can be proved using only the laws of KA.

Many of the results presented in this chapter have appeared in the literature (e.g. in [12], [20], [21]). We will include proofs inspired by these works, and in some cases we will present them differently. In particular, we give a new account using *locally finite matrices* which fills in a gap in previous coalgebraic proofs.

5.1 Matrices over Kleene algebras

We will start with what is perhaps the single most useful observation about Kleene algebra, which comes down to the following slogan:

Matrices over a Kleene algebra form a Kleene algebra!

We will first develop this idea for finite matrices. Matrices indexed by infinite sets are a bit more subtle, and we will spend some time working on that in the latter half of this section. For the remainder of this section, we fix a Kleene algebra $\mathcal{K} = (K, +, \cdot, -^*, 0, 1)$.

5.1.1 Finite matrices

To get an idea of how our slogan works for finite matrices, consider 2×2-matrices formed using elements of a Kleene algebra $\mathcal{K}$. We can define addition and multiplication of these matrices in the usual way:

$$\begin{pmatrix} e_1 & e_2 \\ e_3 & e_4 \end{pmatrix} + \begin{pmatrix} e_5 & e_6 \\ e_7 & e_8 \end{pmatrix} = \begin{pmatrix} e_1 + e_5 & e_2 + e_6 \\ e_3 + e_7 & e_4 + e_8 \end{pmatrix},$$

$$\begin{pmatrix} e_1 & e_2 \\ e_3 & e_4 \end{pmatrix} \cdot \begin{pmatrix} e_5 & e_6 \\ e_7 & e_8 \end{pmatrix} = \begin{pmatrix} e_1e_5 + e_2e_7 & e_1e_6 + e_2e_8 \\ e_3e_5 + e_4e_7 & e_3e_6 + e_4e_8 \end{pmatrix}.$$

It should be clear that the neutral elements for $+$ and $\cdot$ are, respectively:

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \qquad \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

The "only" question that remains is: how do we define an operation on matrices that satisfies the axioms of the Kleene star? As it turns out, for $n = 2$, the following is a valid choice:

$$\begin{pmatrix} e_1 & e_2 \\ e_3 & e_4 \end{pmatrix}^* = \begin{pmatrix} (e_1 + e_2e_4^*e_3)^* & (e_1 + e_2e_4^*e_3)^*e_2e_4^* \\ (e_4 + e_3e_1^*e_2)^*e_3e_1^* & (e_4 + e_3e_1^*e_2)^* \end{pmatrix},$$

This may look like an arbitrary soup of symbols right now, although the eagle-eyed reader may recognize the operation on matrices from page 48, defined to solve systems of equations. We will derive a way to compute the star of arbitrary $n \times n$-matrices for arbitrary n in a moment; for now, let us pauze and formalize what we already know.

Definition 5.2. Let Q be a set. A *Q-vector (over $\mathcal{K}$)* is a function $v\colon Q \to \mathcal{K}$, and a *$Q$-matrix (over $\mathcal{K}$)* is a function $M\colon Q \times Q \to \mathcal{K}$.

Addition of matrices and vectors is defined pointwise, e.g., for Q-matrices M and N, we define $M + N$ as the Q-matrix defined by:

$$(M + N)(q, q') = M(q, q') + N(q, q')$$

Multiplication of finite Q-matrices by vectors and other matrices works as usual: for Q-matrices M and N and Q-vectors v, we have that $M \cdot N$ and $M \cdot v$ are respectively the Q-matrix and Q-vector defined by:

$$(M \cdot N)(q, q') = \sum_{q''} M(q, q'') \cdot N(q'', q') \qquad (M \cdot v)(q) = \sum_{q''} M(q, q') \cdot v(q')$$

Note how the sums on the right-hand side are finite, because Q is finite.

Finally, given a Q-vector v and $x \in \mathcal{K}$, we write $v \fatsemi x$ for the *(right) scalar multiplication* of v by x, which is simply the Q-vector where

$$(v \fatsemi x)(q) = v(q) \cdot x$$

Lemma 5.3. Let X, Y and Z be finite Q-matrices. Let's write $\mathbf{1}$ for the identity Q-matrix, where $\mathbf{1}(q, q) = 1$ and $\mathbf{1}(q, q') = 0$ when $q \neq q'$, and $\mathbf{0}$ for the Q-matrix where $\mathbf{0}(q, q') = 0$. The following hold:

$$X + \mathbf{0} = X \qquad X + X = X \qquad X + Y = Y + X$$

$$X + (Y + Z) = (X + Y) + Z \qquad X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z$$

$$X \cdot (Y + Z) = X \cdot Y + X \cdot Z \qquad (X + Y) \cdot Z = X \cdot Z + Y \cdot Z$$

$$X \cdot \mathbf{1} = X = X \cdot \mathbf{1} \qquad X \cdot \mathbf{0} = \mathbf{0} = \mathbf{0} \cdot X$$

Proof. Each law can be proved by unfolding the definitions of the operators, and applying the corresponding law on the level of $\mathcal{K}$. □

Because the $+$ operation on matrices satisfies the same laws as expected from the $+$ in a Kleene algebra, we automatically get an order defined on Q-matrices in the same way: $M \leq N$ when $M + N = N$.

Next we show that matrices over a Kleene algebra satisfy the star axioms. We gave a direct definition of the star of a matrix on page 48. For our purposes it is more practical to define the star of a matrix using least solutions. These two definitions can be shown to be equivalent [11, Theorem A.3]. We start with the following technical construction.

Lemma 5.4. Let Q be a finite set, with M a Q-matrix and b a Q-vector. We can construct a Q-vector s such that $b + M \cdot s \leq s$, and furthermore if t is a Q-vector and $z \in \mathcal{K}$ with $b \fatsemi z + M \cdot t \leq t$, then $s \fatsemi z \leq t$.

Proof. To get an idea, let's look at the special case where Q is a singleton set. Here, M and b contain just one element of $\mathcal{K}$ — so we treat them as such. Moreover, addition and multiplication comes down to adding and multiplying single elements. Thus, we are really looking for $s \in \mathcal{K}$ where $b + M \cdot s \leq s$, and if $t \in \mathcal{K}$ such that $b \cdot z + M \cdot t \leq t$, then $s \cdot z \leq t$. But we already know exactly such an expression: it's $M^* \cdot b$. After all,

$$b + M \cdot M^* \cdot b = (1 + M \cdot M^*) \cdot b = M^* \cdot b$$

$$b \cdot z + M \cdot t \leq t \implies M^* \cdot b \cdot z \leq t$$

Our job is to generalize this approach to sets Q that contain *more* elements. To this end, we proceed by induction on (the size of) Q.

In the base, where $Q = \emptyset$, we can simply choose s to be the unique Q-vector $s \colon \emptyset \to \mathcal{K}$, which satisfies both conditions vacuously. For the inductive step, let $p \in Q$ be our *pivot*, and choose $Q' = Q \setminus \{p\}$. We are going to create a Q'-vector and Q'-matrix, and then use the induction hypothesis to obtain a Q'-vector; next, we will extend this Q'-vector into a Q-vector satisfying our objectives.

Let M' and b' respectively be the Q'-matrix and Q'-vector given by

$$M'(q, q') = M(q, q') + M(q, p) \cdot M(p, p)^* \cdot M(p, q')$$

$$b'(q) = b(q) + M(q, p) \cdot M(p, p)^* \cdot b(p)$$

By induction we obtain a Q'-vector s' such that $b' + M' \cdot s' \leq s'$, and moreover if t' is a Q'-vector such that $b' \fatsemi z + M' \cdot t' \leq t'$, then $s' \fatsemi z \leq t'$.

We extend s to a Q-vector as follows:

$$s(q) = \begin{cases} s'(q) & q \in Q' \\ M(p,p)^* \cdot \left(b(p) + \sum_{q' \in Q'} M(p,q') \cdot s'(q')\right) & q = p \end{cases}$$

We leave the proof that s satisfies our goal as an exercise. □

This construction then immediately gives us a way to define the star of a matrix, which we record in the following lemma.

Lemma 5.5 (Star of a matrix)**.** Let Q be a finite set, and let M be a Q-matrix. We can construct a matrix M^* such that the following hold:

(i) if s and b are Q-vectors s.t. $b + M \cdot s \leq s$, then $M^* \cdot b \leq s$; and

(ii) $\mathbf{1} + M \cdot M^* = M^*$, where $\mathbf{1}$ is as in Lemma 5.3.

Proof. For each $q \in Q$, we write u_q for the Q-vector where $u_q(q') = \mathbf{1}(q,q')$. Furthermore, we write s_q for the Q-vector such that $u_q + M \cdot s_q \leq s_q$, and if $z \in \mathcal{K}$ and t is a Q-vector such that $u_q \fatsemi z + M \cdot t \leq t$, then $s_q \fatsemi z \leq t$; note that we can construct this Q-vector, per Lemma 5.4.

We now choose the Q-matrix M^* as follows:

$$M^*(q,q') = s_{q'}(q)$$

It remains to show that M^* satisfies the two requirements above.

(i) Suppose that s and b are Q-vectors such that $b + M \cdot s \leq s$. We need to show that $M^* \cdot b \leq s$, that is show that for all $q \in Q$:

$$\sum_{q' \in Q} M^*(q,q') \cdot b(q') \leq s(q)$$

It suffices to show that, for all $q, q' \in Q$, we have $s_{q'}(q) \cdot b(q') \leqq s(q)$, that is $s_{q'} \fatsemi b(q') \leq s$. By construction of $s_{q'}$, it then suffices to show that $u_{q'} \fatsemi b(q') + M \cdot s \leq s$. To this end, we derive:

$$(u_{q'} \fatsemi b(q') + M \cdot s)(q) \leq (b + M \cdot s)(q) \leq s(q)$$

which completes this part of the proof.

(ii) For the second part of the claim, let $q, q' \in Q$; we derive as follows:

$$\begin{aligned}
&(\mathbf{1} + M \cdot M^*)(q, q') \\
&= \mathbf{1}(q, q') + \sum_{q'' \in Q} M(q, q'') \cdot M^*(q'', q') && \text{(by def.)} \\
&= u_{q'}(q) + \sum_{q'' \in Q} M(q, q'') \cdot s_{q'}(q'') && \text{(def. } u_{q'}, M^*) \\
&= (u_{q'} + M \cdot s_{q'})(q) && \text{(by def.)} \\
&= s_{q'}(q) && \text{(see below)} \\
&= M^*(q, q') && \text{(def. } M^*)
\end{aligned}$$

where the second to last equality follows from the fact that

$$u_{q'} + M \cdot (u_{q'} + M \cdot s_{q'}) \leq u_{q'} + M \cdot s_{q'}$$

and hence $s_{q'} \leq u_{q'} + M \cdot s_{q'}$, meaning that $u_{q'} + M \cdot s_{q'} = s_{q'}$. □

The left fixpoint rule now follows immediately:

Corollary 5.6. Let Q be a finite set, and let M and B be Q-matrices. If S is a Q-matrix such that $B + M \cdot S \leq S$, then $M^* \cdot B \leq S$.

A proof of the remaining axioms — i.e., that (1) $\mathbf{1} + M^*M = M^*$ and (2) if $B + S \cdot M \leq S$ then $B \cdot M^* \leq S$ — requires some thought. We leave these to the reader in Exercise 5.5, and record the following:

Theorem 5.7. The addition, multiplication and star operations on matrices satisfy all of the laws of Kleene algebra. In particular, in addition to the laws listed in Lemma 5.3, if X, Y and Z are Q-matrices for some finite set Q, then

$$\mathbf{1} + X \cdot X^* = X^* \qquad X + Y \cdot Z \leq Z \implies Y^* \cdot Z \leq Z$$

$$\mathbf{1} + X^* \cdot X = X^* \qquad X + Y \cdot Z \leq Y \implies X \cdot Z^* \leq Y$$

5.1.2 Infinite matrices

In the proof of completeness to come, we will model finite automata using finite matrices, and their corresponding expressions using the Kleene star of those matrices. However, we will also need to talk about potentially infinite automata. It is not at all clear that the operations we defined on finite matrices still make sense for infinite ones, and if they do, whether the same laws apply. In this subsection, we will study infinite matrices over a Kleene algebra, and argue that the same equations hold when the operations can be extended to infinite matrices.

The first operator to look at is addition. Fortunately, there is no problem here at all: because addition of matrices is defined pointwise, it can be extended immediately to infinite matrices. Furthermore, the laws that involve only addition, such as associativity, still work for the same reason that they do for finite matrices.

Next, we should look at multiplication. Here, we encounter a problem when we try to extend the old definition. Specificially, let Q be infinite, and let $M, N \colon Q \times Q \to \mathcal{K}$. If we try to define $M \cdot N$ as usual, i.e.:

$$(M \cdot N)(q, q') = \sum_{q'' \in Q} M(q, q'') \cdot N(q'', q')$$

we risk the possibility that the sum on the right is infinite! This would be ill-defined, because we only have finitary addition in $\mathcal{K}$. To work around this problem, we introduce the following notion.

Definition 5.8 (Locally-finite matrix)**.** Let $M \colon Q \times Q \to \mathcal{K}$ be a matrix for a possibly infinite set Q. When $q \in Q$, we write $R_M(q)$ for the *reach* of q, which is the smallest set satisfying the following rules:

$$\frac{}{q \in R_M(q)} \qquad \frac{q' \in R_M(q) \qquad M(q', q'') \neq 0}{q'' \in R_M(q)}$$

That is, if we regard M as the adjacency matrix of a directed graph, then $R_M(q)$ is the set of vertices that can be reached from the vertex q.

We say that M is *locally-finite* when for each $q \in Q$, $R_M(q)$ is finite.

Example 5.9. Every finite matrix is locally finite. Examples of infinite but locally-finite matrices include the units $\mathbf{0}, \mathbf{1} \colon Q \times Q \to \mathcal{K}$; after all, $R_{\mathbf{0}}(q) = \emptyset$ for all q, and $R_{\mathbf{1}}(q) = \{q\}$ for all q.

Example 5.10. Consider the matrix $M_{\mathsf{Exp}/\equiv}$ over $\equiv$-equivalence classes of expressions given by $M_{\mathsf{Exp}/\equiv}([e], [f]) = \sum\{\mathsf{a} \mid e_{\mathsf{a}} \equiv f\}$. This matrix is locally finite by Theorem 4.27 and the observation that $R_{M_{\mathsf{Exp}/\equiv}}([e]) = \{[e_w] \mid w \in A^*\}$; it will play a role in the completeness proof to follow.

Matrix multiplication makes sense when the left operand is locally finite. Specifically, if $M(q, q'') \cdot N(q'', q') \neq 0$, then certainly $M(q, q'') \neq 0$, and if this is the case then $q'' \in R_M(q)$; because there are only finitely many such q'' when M is locally-finite, there are only finitely many non-zero terms that contribute to the sum on the right.

Because matrix multiplication extends to some infinite matrices with the same definition, the same laws apply when both sides of the equation are defined. For instance, if $X, Y, Z \colon Q \times Q \to \mathcal{K}$ are matrices such that X, Y and $X \cdot Y$ are locally finite, then $X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z$.

Remark 5.11. As can be seen from the above, some care is necessary when stating equations involving operations on infinite matrices. For one thing, it is not sufficient to assume that X and Y are locally finite, because $X \cdot Y$ may not be. Take for instance $Q = \mathbb{N}$ and set

$$X(2k, 2k+1) = 1 \qquad\qquad Y(2k+1, 2k+2) = 1$$

with all other entries of X and Y being zero. In that case, X and Y are locally finite but $X \cdot Y$ is not (nor is $X + Y$, for that matter).

For the sake of brevity, we will omit such qualifications from equalities involving infinite matrices going forward, although we will remark on when certain expressions are defined, just to be rigorous.

The last operator that we need to look at is the Kleene star. Here, we will also limit ourselves to locally finite matrices. Intuitively, this makes sense because the Kleene star of a matrix corresponding to an automaton can be used to calculate a regular expression for the language of each state in that automaton. If the Kleene star of a matrix could be extended to infinite matrices in general, it would mean that we could calculate regular expressions corresponding to infinite automata — which would be problematic. On the other hand, locally finite matrices correspond to (potentially infinite) automata in which every state can reach only finitely many states; it should be clear that in that setting, it is still possible to compute a regular expression for every state.

Definition 5.12. Let $M \colon Q \times Q \to \mathcal{K}$ be a matrix, with Q potentially infinite. When $P \subseteq Q$, we write M_P for the P-indexed submatrix of M, i.e., $M_P \colon P \times P \to \mathcal{K}$ with $M_P(q, q') = M(q, q')$ for $q, q' \in P$.

If M is locally finite, we write M^* for the Q-indexed matrix where

$$M^*(q,q') = \begin{cases} M^*_{R_M(q)}(q,q') & q' \in R_M(q) \\ 0 & q' \not\in R_M(q) \end{cases}$$

Note that when M is locally finite, $R_M(q)$ is finite for every $q \in Q$, which means that $M_{R_M(q)}$ is a finite matrix, and hence $M^*_{R_M(q)}$ is defined.

It remains to verify the laws involving the Kleene star. Let us start with $\mathbf{1} + M \cdot M^* = M^*$. We will need the following technical property.

Lemma 5.13. Let $M \colon Q \times Q \to \mathcal{K}$ be a finite matrix, and let $R \subseteq Q$ such that if $q \in R$ and $M(q,q') \neq 0$, then $q' \in R$. Now $(M_R)^* = (M^*)_R$.

Proof. To see that $(M_R)^* \leq (M^*)_R$, it suffices to show that $M_R \cdot (M^*)_R + \mathbf{1} \leq (M^*)_R$. To this end, we derive as follows, for $q, q' \in R$:

$$\begin{aligned}(M_R \cdot (M^*)_R + \mathbf{1})(q,q') &= \sum_{q'' \in R} M_R(q,q'')(M^*)_R(q'',q') + [q = q'] \\ &= \sum_{q'' \in R} M(q,q') \cdot M^*(q'',q') + [q = q'] \\ &\leq \sum_{q'' \in Q} M(q,q') \cdot M^*(q'',q') + [q = q'] \\ &= (M \cdot M^* + \mathbf{1})(q,q') \\ &= M^*(q,q') = (M^*)_R(q,q')\end{aligned}$$

To show that $(M^*)_R \leq (M_R)^*$, we need a slightly more involved argument. Let $N \colon Q \times Q \to \mathcal{K}$ be the matrix defined by

$$N(q,q') = \begin{cases} (M_R)^*(q,q') & q, q' \in R \\ M^*(q,q') & \text{otherwise} \end{cases}$$

It now suffices to show that $M^* \leq N$, because $N_R = (M_R)^*$ by construction. To this end, we need only show that $M \cdot N + \mathbf{1} \leq N$. There are two cases; first, if $q, q' \in R$, then we derive

$$\begin{aligned}(M \cdot N + \mathbf{1})(q,q') &= \sum_{q'' \in Q} M(q,q'') \cdot N(q'',q') + [q = q'] \\ &= \sum_{q'' \in R} M_R(q,q'') \cdot (M_R)^*(q'',q') + [q = q'] \\ &= (M_R \cdot M_R^* + \mathbf{1})(q,q') \\ &= M_R^*(q,q') = N(q,q')\end{aligned}$$

Here, the second step is sound because if $q \in R$ and $M(q,q'') \neq 0$, then $q'' \in R$ as well. For the other case, where $q \notin R$ or $q' \notin R$, we find that

$$\begin{aligned}(M \cdot N + \mathbf{1})(q,q') &= \sum_{q'' \in Q} M(q,q'') \cdot N(q'',q') + [q = q'] \\ &\leq \sum_{q'' \in Q} M(q,q'') \cdot M^*(q'',q') + [q = q'] \\ &= (M \cdot M^* + \mathbf{1})(q,q'') \\ &= M^*(q,q') = N(q,q')\end{aligned}$$

In the second step, we use $N \leq M^*$; after all, if $q, q' \in R$ then $N(q,q') = (M_R)^*(q,q') \leq (M^*)_R(q,q') = M^*(q,q')$ (by the derivation above), and if $q \notin R$ or $q' \notin R$, then $N(q,q') = M^*(q,q')$ by definition. $\square$

With this property in hand, we can now verify the first equation.

Lemma 5.14. If $M \colon Q \times Q \to \mathcal{K}$ is locally finite, then $\mathbf{1} + M \cdot M^* = M^*$.

Proof. We can derive for $q, q' \in Q$ as follows:

$$\begin{aligned}&(\mathbf{1} + M \cdot M^*)(q,q') \\ &= [q = q'] + \sum_{q'' \in Q} M(q,q'') \cdot M^*(q'',q') \\ &= [q = q'] + \sum_{\substack{q'' \in R_M(q) \\ \text{s.t. } q' \in R_M(q'')}} M_{R_M(q)}(q,q'') \cdot (M_{R_M(q'')})^*(q'',q') \\ &= [q = q'] + \sum_{\substack{q'' \in R_M(q) \\ \text{s.t. } q' \in R_M(q)}} M_{R_M(q)}(q,q'') \cdot (M_{R_M(q)})^*(q'',q') \qquad (\dagger) \\ &= (\mathbf{1} + M_{R_M(q)} \cdot (M_{R_M(q)})^*)(q,q') = (M_{R_M(q)})^*(q,q') = M^*(q,q')\end{aligned}$$

In the step marked with (†), we use that because $q'' \in R_M(q)$, it holds that $R_M(q'') \subseteq R_M(q)$, so if $q'' \in R_M(q)$ with $q' \in R_M(q'')$, then

$$\begin{aligned}(M_{R_M(q)})^*(q'', q') &= ((M_{R_M(q)})^*)_{R_M(q'')}(q'', q') \\ &= ((M_{R_M(q)})_{R_M(q'')})^*(q'', q') && \text{(Lemma 5.13)} \\ &= (M_{R_M(q'')})^*(q'', q')\end{aligned}$$

In the same step, we also quietly used the fact that if $(M_{R_M})^*(q'', q') \neq 0$, then $q' \in R_M(q'')$; this is a consequence of Exercise 5.5.2(a). □

We move on to the proof of the left fixpoint rule: i.e., that $B + MS \leq S$ implies $M^*B \leq S$. As before, we instead prove a variant of the claim for vectors. The generalisation to matrices then follows.

Lemma 5.15. Let $M\colon Q \times Q \to \mathcal{K}$ be locally finite, and let $b, s\colon Q \to \mathcal{K}$. If $b + M \cdot s \leq s$, then $M^* \cdot b \leq s$ as well.

Proof. To prove that $M^* \cdot b \leq s$, we should show that for every $q \in Q$,

$$\sum_{q' \in Q} M^*(q, q')b(q') \leq s(q)$$

For all $q' \notin R_M(q)$ we have $M^*(q, q') = 0 \leq s(q)$ (c.f. Exercise 5.5.2(b)). So the above is equivalent to showing that for all $q \in Q$, we have that

$$\sum_{q' \in R_M(q)} (M_{R_M(q)})^*(q, q') \cdot b_{R_M(q)}(q') \leq s_{R_M(q)}(q)$$

which is true if for $q \in Q$, we have that $(M_{R_M(q)})^* \cdot b_{R_M(q)} \leq s_{R_M(q)}$. Since this is an inequation of *finite* vectors, the latter holds precisely when $b_{R_M(q)} + (M_{R_M(q)}) \cdot s_{R_M(q)} \leq s_{R_M(q)}$ (c.f. Lemma 5.5). This follows from our assumption that $b + M \cdot s \leq s$ by restricting the vectors on both sides to $R_M(q)$. □

The right-handed laws about M^*, i.e., that (1) $M^* \cdot M + \mathbf{1} \equiv M^*$, (2) and if $B + S \cdot M \leq S$ then $B \cdot M^* \leq S$, can be derived in the same way we derived these for the finite case — see Exercise 5.5.

This concludes our study of the laws of Kleene algebra for matrices indexed by infinite sets; we will leverage these in the sections to come.

5.2 Soundness

Recall that bisimilarity for deterministic automata coincides with language equivalence, that is $L(e_1) = L(e_2) \iff e_1 \sim e_2$.

The original proof soundness and completeness for Kleene algebras is due to Kozen [22], who later presented an alternative proof [20]. A coalgebraic proof of soundness and completeness was presented by Jacobs [21]. The proof we present is a slight variation on these.

For soundness, the right to left implication, it is enough to prove that provable equivalence is a bisimulation relation. This will have as consequence that the Kleene algebra axiomatization of regular expressions is sound: for all $e_1, e_2 \in \mathsf{Exp}$, if $e_1 \equiv e_2$ then $e_1 \sim e_2$.

Theorem 5.16 (Soundness)**.** Provable equivalence is a bisimulation, that is, for every $e_1, e_2 \in \mathsf{Exp}$, if $e_1 \equiv e_2$ then

$$o(e_1) = o(e_2) \text{ and } (e_1)_a \equiv (e_2)_a$$

Proof. By induction on the length of derivations for $e_1 \equiv e_2$.

For the base cases, i.e., all but the last two equations in Definition 2.16, everything follows from unwinding the definitions of o and t. We show the proof for the equations $e_1 + e_2 \equiv e_2 + e_1$ and $ee^* + 1 \equiv e^*$.

$$\begin{aligned} o(e_1 + e_2) &= o(e_1) \vee o(e_2) = o(e_2 + e_1) \\ (e_1 + e_2)_a &\equiv (e_1)_a + (e_2)_a \equiv (e_2)_a + (e_1)_a \equiv (e_2 + e_1)_a \end{aligned}$$

$$\begin{aligned} (ee^* + 1)_a &\equiv e_a e^* + o(e)(e^*)_a + 0 \\ o(ee^* + 1) &= o(ee^*) \vee o(1) = o(ee^*) \vee 1 = 1 = o(e^*) \\ &\equiv \begin{cases} (e^*)_a e 0 (e^*)_a + 0 & \text{if } o(e_1) = 0 \\ (e^*)_a + 1(e^*)_a + 0 & \text{otherwise} \end{cases} \\ &\equiv (e^*)_a \end{aligned}$$

For the inductive step, we illustrate the where $e_2^* e_1 \leqq f$ because $e_1 + e_2 f \leqq f$; the other case is proved precisely in the same way. Suppose we have just derived $e_2^* e_1 \leqq f$, using $e_1 + e_2 f \leqq f$ as a premise. Then by induction we know that $o(e_1 + e_2 f) \leq o(f)$ and $(e_1 + e_2 f)_a \leqq f_a$. We want to prove that $o(e_2^* e_1) \leq o(f)$ and $(e_2^* e_1)_a \leqq f_a$. The first, which simplifies to $o(e_1) \leq o(f)$, follows from the induction hypothesis. For the second inequality, we calculate:

$$\begin{aligned}
(e_2^* e_1)_a &\equiv (e_2)_a e_2^* e_1 + (e_1)_a \\
&\leqq (e_2)_a f + (e_1)_a && (e_2^* e_1 \leqq f) \\
&\leqq (e_2)_a f + o(e_2) f_a + (e_1)_a \\
&\equiv (e_2 f)_a + (e_1)_a \\
&\equiv (e_1 + e_2 f)_a \\
&\leqq f_a && \text{(induction hypothesis)}
\end{aligned}$$

This completes the proof. □

5.3 The road to completeness

Let $\mathsf{Exp}/_{\equiv}$ denote the set of expressions modulo provable equivalence. This induces the quotient map $[-] \colon \mathsf{Exp} \to \mathsf{Exp}/_{\equiv}$ given by

$$[e] = \{e' \mid e \equiv e'\}$$

Soundness (Theorem 5.16) tells us that $\equiv$ is a bisimulation on Exp. Since $\equiv$ is its own equivalence closure, Lemma 3.19 then gives us a coalgebra structure on $\mathsf{Exp}/_{\equiv}$ $\langle \overline{o}, \overline{t} \rangle \colon \mathsf{Exp}/_{\equiv} \to 2 \times (\mathsf{Exp}/_{\equiv})^A$ such that $[-] \colon \mathsf{Exp} \to \mathsf{Exp}/_{\equiv}$ is a coalgebra homomorphism. Concretely, $\overline{o}$ and $\overline{t}$ work as expected: given $e \in \mathsf{Exp}$, $\overline{o}([e]) = o(e)$ and $\overline{t}([e])(a) = [e_a]$.

We will show a coalgebraic proof of completeness (that is, $e_1 \sim e_2 \Rightarrow e_1 \equiv e_2$). Our proof is inspired by the one given by Jacobs [21], which can be seen as a coalgebraic review of Kozen's proof [20], though we present some lemmas (and respective proofs) slightly differently. We first present an overview of all the steps necessary for completeness.

Step ① First, let $L \colon \mathsf{Exp}/_{\equiv} \to 2^{A^*}$ be the final homomorphism, and let $(I, \langle o_I, t_I \rangle)$ be the D-coalgebra obtained by quotienting $(\mathsf{Exp}/_{\equiv}, \langle \overline{o}, \overline{t} \rangle)$ using $\mathsf{Ker}(L)$, per Lemma 3.19. By Lemma 3.29, this brings us the situation in the diagram below, with $q \colon \mathsf{Exp}/_{\equiv} \to I$ being the quotient map, and m being the injective homomorphism from $(I, \langle o_I, t_I \rangle)$ to

$(2^{A^*}, \langle o_L, t_L \rangle)$.

$$\begin{tikzcd}
\mathsf{Exp}/_{\equiv} \arrow[r, "q"] \arrow[d, "\langle \overline{o}, \overline{t} \rangle"'] \arrow[rr, bend left, "L"] & I \arrow[r, "m"] \arrow[d, "\langle o_I, t_I \rangle"] & 2^{A^*} \arrow[d] \\
2 \times (\mathsf{Exp}/_{\equiv})^A \arrow[r] & 2 \times I^A \arrow[r] & 2 \times (2^{A^*})^A
\end{tikzcd}$$

Step ② We prove that the quotient map q is a bijection. The key idea is to show that both $(\mathsf{Exp}/_{\equiv}, \langle \overline{o}, \overline{t} \rangle)$ and $(I, \langle o_I, t_I \rangle)$ are final among *locally finite* automata — i.e., those where every state is contained in a finite subcoalgebra. This is true for $(\mathsf{Exp}/_{\equiv}, \langle \overline{o}, \overline{t} \rangle)$ by Theorem 4.27.

We prove that $(\mathsf{Exp}/_{\equiv}, \langle \overline{o}, \overline{t} \rangle)$ is final using three intermediate steps:

(Lemma 5.18) For any locally finite automaton $(S, \langle o_S, t_S \rangle)$, there exists a homomorphism $\lceil - \rceil_S \colon (S, \langle o_S, t_S \rangle) \to (\mathsf{Exp}/_{\equiv}, \langle \overline{o}, \overline{t} \rangle)$.

(Lemma 5.19) If f is a homomorphism from $(S, \langle o_S, t_S \rangle)$ to $(T, \langle o_T, t_T \rangle)$ and both are locally finite, then it holds that $\lceil f(s) \rceil_T = \lceil s \rceil_S$.

(Lemma 5.21) The homomorphism $\lceil - \rceil_{\mathsf{Exp}/_{\equiv}}$ is the identity.

These three lemmas together imply that for any locally finite automaton $(S, \langle o_S, t_S \rangle)$, there exists a *unique* coalgebra homomorphism

$$\lceil - \rceil_S \colon (S, \langle o_S, t_S \rangle) \to (\mathsf{Exp}/_{\equiv}, \langle \overline{o}, \overline{t} \rangle)$$

Hence, $(\mathsf{Exp}/_{\equiv}, \langle \overline{o}, \overline{t} \rangle)$ is final among locally finite coalgebras.

In fact with these facts in hand, the finality of $(I, \langle o_I, t_I \rangle)$ is easy by comparison. We just need to observe the following two facts:

(i) For any locally finite automaton $(S, \langle o_S, t_S \rangle)$, we can construct a coalgebra homomorphism into $(I, \langle o_I, t_I \rangle)$ as follows:

$$(S, \langle o_S, t_S \rangle) \xrightarrow{\lceil - \rceil_S} (\mathsf{Exp}/_{\equiv}, \langle \overline{o}, \overline{t} \rangle) \xrightarrow{q} (I, o_I, t_I \rangle)$$

Here, $\lceil - \rceil_S$ is as in the statement of Lemma 5.18, and q is the quotient morphism from the diagram above.

(ii) If there exist two homomorphisms $f, g\colon (S, \langle o_S, t_S\rangle) \to (I, \langle o_I, t_I\rangle)$, then $f = g$. After all, by finality of 2^{A^*}, we know that, for any $s \in S$, $L(s) = m \circ g(s)$ and $L(s) = m \circ f(s)$. Hence, we have that $m \circ f = m \circ g$ and, since m is injective, $f = g$.

Step ③ From ① and ②, $q\colon (\mathsf{Exp}/_{\equiv}, \langle \overline{o}, \overline{t}\rangle) \to (I, \langle o_I, t_I\rangle)$ is an isomorphism, and hence injective. This makes $L = m \circ q\colon \mathsf{Exp}/_{\equiv} \to 2^{A^*}$ injective, which is the key to prove completeness.

Theorem 5.17 (Completeness)**.** If $e_1 \sim e_2$ then $e_1 \equiv e_2$.

Proof. For all $e_1, e_2 \in \mathsf{Exp}$, if $e_1 \sim e_2$ then they are mapped into the same element in the final coalgebra: $L(e_1) = L(e_2)$. Since $[-]$ is a homomorphism, $L([e_1]) = L([e_2])$, where we use L to denote the map into the final coalgebra both from Exp and $\mathsf{Exp}/_{\equiv}$. By ③, L is injective and thus $[e_1] = [e_2]$, that is, by definition of $[-]$, $e_1 \equiv e_2$. □

5.4 Filling in the details

It remains to prove the properties of $\mathsf{Exp}/_{\equiv}$ alluded to above. We start by constructing the desired map, and showing that it is a homomorphism.

Lemma 5.18. Let $(S, \langle o_S, t_S\rangle)$ be a locally finite automaton. There exists a coalgebra homomorphism $\lceil - \rceil_S\colon (S, \langle o_S, t_S\rangle) \to (\mathsf{Exp}/_{\equiv}, \langle \overline{o}, \overline{t}\rangle)$.

Proof. Let $M_S\colon Q \times Q \to \mathsf{Exp}$ be the transition matrix of S, given by

$$M_S(q, q') = \sum\{\mathsf{a} \in A \mid t_S(q)(\mathsf{a}) = q'\}$$

Because $(S, \langle o_S, t_S\rangle)$ is locally finite, so is M_S.

If we view $o_S\colon Q \to 2$ as a vector of expressions, then $M_S^* o_S$ is another vector of expressions.[1] We choose $\lceil q \rceil_S = [(M_S^* o_S)(q)]$. To see that $\lceil - \rceil_S$ is a homomorphism, we must prove two equalities:

$$\overline{o}(\lceil q \rceil_S) = o_S(q) \qquad\qquad \overline{t}(\lceil q \rceil_S)(\mathsf{a}) = \lceil t_S(q)(\mathsf{a}) \rceil_S$$

[1]Note that $(S, \langle o_S, t_S\rangle)$ is locally finite, which makes M_S a locally finite matrix.

For the first equality, we derive as follows:

$$\begin{aligned}
\overline{o}(\lceil q \rceil_S) &= \overline{o}([M_S^* o_S(q)]) \\
&= o(M_S^* o_S(q)) \\
&\equiv o(o_S(q) + M_S M_S^* o_S(q)) \\
&= o(o_S(q)) + o(M_S M_S^* o_S(q)) \\
&= o(o_S(q)) + o\Big(\sum_{q'} M_S(q,q')(M_S^* o_S)(q')\Big) \\
&= o(o_S(q)) + \sum_{q'} o(M_S(q,q'))o((M_S^* o_S)(q')) \\
&\equiv o(o_S(q)) \\
&= o_S(q)
\end{aligned}$$

where the second-to-last step holds since $o(M_S(q,q')) = 0$.

For the second equality, we start by noting the following:

$$\begin{aligned}
(M_S^* o_S(q))_{\mathtt{a}} &\equiv (o_S(q) + M_S M_S^* o_S(q))_{\mathtt{a}} && (\text{Lemma 5.14}) \\
&\equiv (M_S M_S^* o_S(q))_{\mathtt{a}} && ((o_S(q))_{\mathtt{a}} = 0) \\
&\equiv \Big(\sum_{q'} M_S(q,q') M_S^* o_S(q')\Big)_{\mathtt{a}} \\
&\equiv \Big(\Big(\sum \{\mathtt{b} \in A \mid t_S(q)(\mathtt{b}) = q'\}\Big) \cdot M_S^* o_S(q')\Big)_{\mathtt{a}} \\
&\equiv \Big(\sum \{\mathtt{b} \cdot M_S^* o_S(t_S(q)(\mathtt{b})) \mid \mathtt{b} \in A\}\Big)_{\mathtt{a}} \\
&\equiv \sum \{(\mathtt{b} \cdot M_S^* o_S(t_S(q)(\mathtt{b})))_{\mathtt{a}} \mid \mathtt{b} \in A\} \\
&\equiv (\mathtt{a} \cdot M_S^* o_S(t_S(q)(\mathtt{a})))_{\mathtt{a}} \\
&\equiv M_S^* o_S(t_S(q)(\mathtt{a}))
\end{aligned}$$

From this, we can derive that

$$\begin{aligned}
\overline{t}(\lceil q \rceil_S)(a) &= \overline{t}([M_S^* o_S(q)])(\mathtt{a}) \\
&= [(M_S^* o_S(q))_{\mathtt{a}}] \\
&= [M_S^* o_S(t_S(q)(\mathtt{a}))] \\
&= \lceil t_S(q)(\mathtt{a}) \rceil
\end{aligned}$$

This concludes the proof. □

Next, we show that the homomorphisms into $(\mathsf{Exp}/_{\equiv}, \overline{o}, \overline{t})$ commute with homomorphisms among locally finite coalgebras.

Lemma 5.19. Let $(S, \langle o_S, t_S \rangle)$ and $(T, \langle o_T, t_T \rangle)$ be locally finite automata. For any homomorphism $f\colon (S, \langle o_S, t_S\rangle) \to (T, \langle o_T, t_T \rangle)$, it holds that $\lceil f(s) \rceil_T = \lceil s \rceil_S$.

Proof. Let $M_f\colon S \times T \to \mathsf{Exp}$ be given by $M_f(q, q') = [f(q) = q']$. We start by claiming that $M_f \cdot M_T \equiv M_S \cdot M_f$. To see this, derive as follows:

$$\begin{aligned}
(M_f \cdot M_T)(q, q') &\equiv \sum_{q''} M_f(q, q'') \cdot M_T(q'', q') \\
&\equiv M_T(f(q), q') \\
&\equiv \sum \{\mathtt{a} \in A \mid t_T(f(q))(\mathtt{a}) = q'\} \\
&\equiv \sum \{\mathtt{a} \in A \mid f(t_S(q)(\mathtt{a})) = q'\} \qquad \text{(homomorphism)} \\
&\equiv \sum_{q''} \Big\{ \sum \{\mathtt{a} \mid t_S(q)(\mathtt{a}) = q''\} \mid f(q'') = q' \Big\} \\
&\equiv \sum_{q''} M_S(t_S(q)(\mathtt{a}), q'') \cdot M_f(q'', q') \\
&\equiv (M_S \cdot M_f)(t_S(q)(\mathtt{a}), q')
\end{aligned}$$

In the previous section, we argued that the laws of KA hold for matrices, to the extent that the expressions on both sides are well-defined (e.g., the Kleene star is applied only to locally finite matrices). Because M_A is locally finite, we can conclude from the fact that $M_f \cdot M_T \equiv M_S \cdot M_f$ and Exercise 5.3 below that $M_f \cdot M_T^* \equiv M_S^* \cdot M_f$.

Next, we claim that $M_f \cdot o_T \equiv o_S$; this is easier to show:

$$(M_f \cdot o_T)(q) \equiv \sum_{q'} M_f(q, q') o_T(q') \equiv o_T(f(q)) = o_S(q)$$

Putting this together, we can derive that

$$\begin{aligned}
\lceil f(s) \rceil_T &= [M_T^* \cdot o_T(f(s))] \\
&= [M_f \cdot M_T^* \cdot o_T(s)] \\
&= [M_S^* \cdot M_f \cdot o_T(s)] \\
&= [M_S^* \cdot o_S(s)] \\
&= \lceil s \rceil_S
\end{aligned}$$

This completes the proof. $\square$

To place the final piece of the puzzle, we need the following property of $\lceil - \rceil$; the proof is left as Exercise 5.7.

Lemma 5.20. Let $e, f \in \mathsf{Exp}$. If $e \leqq f$, then $\lceil [e] \rceil_{\mathsf{Exp}/\equiv} \leq \lceil [f] \rceil_{\mathsf{Exp}/\equiv}$.

With this in hand, we can now prove the last lemma, which says that $\lceil - \rceil_{\mathsf{Exp}/\equiv} : (\mathsf{Exp}/_\equiv, \langle \overline{o}, \overline{t} \rangle) \to (\mathsf{Exp}/_\equiv, \langle \overline{o}, \overline{t} \rangle)$ acts as the identity.

Lemma 5.21. For all $e \in \mathsf{Exp}$, it holds that $\lceil [e] \rceil_{\mathsf{Exp}/\equiv} = [e]$.

Proof. In the proof below, we treat $\mathsf{Exp}/_\equiv$ as a Kleene algebra, extending the operators on Exp to the equivalence classes; for instance, $[f] + [g] = [f + g]$. This is well-defined because $\equiv$ is a congruence with respect to all operators. To prevent some clutter, we will treat $0, 1 \in 2$ as equivalence classes of expressions. We will also write $[f] \leq [g]$ when $f \leqq g$. Note that this makes $\leq$ a partial order on $\mathsf{Exp}/_\equiv$. To prove the claim, it now suffices to show that $\lceil [e] \rceil_{\mathsf{Exp}/\equiv}([e]) \leq [e]$ and $[e] \leq \lceil [e] \rceil_{\mathsf{Exp}/\equiv}([e])$

To show that $\lceil [e] \rceil_{\mathsf{Exp}/\equiv} \leq [e]$, we can think of id as a vector. If $M_{\mathsf{Exp}/\equiv}$ is the matrix corresponding to $\overline{t} \colon \mathsf{Exp}/_\equiv \to (\mathsf{Exp}/_\equiv)^A$, and we think of $\overline{o} \colon \mathsf{Exp}/_\equiv \to 2$ as a vector, then we can derive as follows:

$$\begin{aligned}
(M_{\mathsf{Exp}/\equiv} \cdot \mathsf{id} + \overline{o})([e]) &= \sum_{[f]} M_{\mathsf{Exp}/\equiv}([e], [f]) \cdot [f] + \overline{o}([e]) \\
&= \sum_{[f]} \sum_{e_{\mathsf{a}} \equiv f} [\mathsf{a} \cdot f] + o(e) \\
&= \sum_{\mathsf{a}} [\mathsf{a} \cdot e_{\mathsf{a}}] + o(e) \\
&= [e] && \text{(Theorem 4.2)}
\end{aligned}$$

It then follows that $\lceil [e] \rceil_{\mathsf{Exp}/\equiv} = [M^*_{\mathsf{Exp}/\equiv} \cdot \overline{o}([e])] \leq [e]$.

In the other direction, we prove that for $e, f \in \mathsf{Exp}$, we have that $[e] \cdot \lceil [f] \rceil_{\mathsf{Exp}/\equiv} \leq \lceil [e \cdot f] \rceil_{\mathsf{Exp}/\equiv}$. This implies the main claim, because

$$[e] = [e] \cdot [1] \leq [e] \cdot \lceil [1] \rceil_{\mathsf{Exp}/\equiv} \leq \lceil [e \cdot 1] \rceil_{\mathsf{Exp}/\equiv} = \lceil [e] \rceil_{\mathsf{Exp}/\equiv}$$

The proof proceeds by induction on e, while still quantifying over all $f \in \mathsf{Exp}$. In the base, there are three cases to consider:

- If $e = 0$, then the claim is trivial, because we have
$$[0] \cdot \lceil [f] \rceil_{\mathsf{Exp}/\equiv} = [0] \leq \lceil [0 \cdot f] \rceil_{\mathsf{Exp}/\equiv}$$
- If $e = 1$, then the claim also trivializes, because
$$[1] \cdot \lceil [f] \rceil_{\mathsf{Exp}/\equiv} = \lceil [f] \rceil_{\mathsf{Exp}/\equiv} = \lceil [1 \cdot f] \rceil_{\mathsf{Exp}/\equiv}$$
- If $e = \mathsf{a}$ for some $\mathsf{a} \in A$, then note that $[(\mathsf{a} \cdot f)_{\mathsf{a}}] = [f]$, and so $[\mathsf{a}] \leq M_{\mathsf{Exp}/\equiv}([\mathsf{a} \cdot f], [f])$. With this in mind, we can derive
$$\begin{aligned}
[\mathsf{a}] \cdot \lceil [f] \rceil_{\mathsf{Exp}/\equiv} &= [\mathsf{a}] \cdot (M^*_{\mathsf{Exp}/\equiv} \cdot \overline{o})([f]) \\
&\leq M_{\mathsf{Exp}/\equiv}([\mathsf{a} \cdot f], [f]) \cdot (M^*_{\mathsf{Exp}/\equiv} \cdot \overline{o})([f]) \\
&\leq (M^*_{\mathsf{Exp}/\equiv} \cdot \overline{o})([\mathsf{a} \cdot f]) \\
&= \lceil [\mathsf{a} \cdot f] \rceil_{\mathsf{Exp}/\equiv}
\end{aligned}$$

For the inductive step, there are three more cases.

- If $e = e_1 + e_2$, then by induction we know that for $i \in \{1, 2\}$:
$$[e_i] \cdot \lceil [f] \rceil_{\mathsf{Exp}/\equiv} \leq \lceil [e_i \cdot f] \rceil_{\mathsf{Exp}/\equiv}$$
  We can then derive that
$$\begin{aligned}
[e] \cdot \lceil [f] \rceil_{\mathsf{Exp}/\equiv} &= [e_1] \cdot \lceil [f] \rceil_{\mathsf{Exp}/\equiv} + [e_2] \cdot \lceil [f] \rceil_{\mathsf{Exp}/\equiv} \\
&\leq \lceil [e_1 \cdot f] \rceil_{\mathsf{Exp}/\equiv} + \lceil [e_2 \cdot f] \rceil_{\mathsf{Exp}/\equiv} && \text{(IH)} \\
&\leq \lceil [e \cdot f] \rceil_{\mathsf{Exp}/\equiv} && \text{(Lemma 5.20)}
\end{aligned}$$
- If $e = e_1 \cdot e_2$, then the following hold for all $f \in \mathsf{Exp}$ and $i \in \{1, 2\}$:
$$[e_i] \cdot \lceil [f] \rceil_{\mathsf{Exp}/\equiv} \leq \lceil [e_i \cdot f] \rceil_{\mathsf{Exp}/\equiv}$$
  With these facts in hand, we can then derive for any $f \in \mathsf{Exp}$ that
$$\begin{aligned}
[e] \cdot \lceil [f] \rceil_{\mathsf{Exp}/\equiv} &= [e_1] \cdot ([e_2] \cdot \lceil [f] \rceil_{\mathsf{Exp}/\equiv}) \\
&\leq [e_1] \cdot \lceil [e_2 \cdot f] \rceil_{\mathsf{Exp}/\equiv} && \text{(IH)} \\
&\leq \lceil [e_1 \cdot (e_2 \cdot f)] \rceil_{\mathsf{Exp}/\equiv} && \text{(IH)} \\
&= \lceil [e \cdot f] \rceil_{\mathsf{Exp}/\equiv}
\end{aligned}$$

- If $e = e_1^*$, then we derive as follows:

$$\begin{aligned}
&[e_1] \cdot \lceil [e \cdot f] \rceil_{\mathsf{Exp}/\equiv} + \lceil [f] \rceil_{\mathsf{Exp}/\equiv} \\
&\quad \leq \lceil [e_1 \cdot (e \cdot f)] \rceil_{\mathsf{Exp}/\equiv} + \lceil [f] \rceil_{\mathsf{Exp}/\equiv} && \text{(IH)} \\
&\quad \leq \lceil [e \cdot f] \rceil_{\mathsf{Exp}/\equiv} && \text{(Lemma 5.20)}
\end{aligned}$$

By the fixpoint rule, the above then lets us derive that

$$[e] \cdot \lceil [f] \rceil_{\mathsf{Exp}/\equiv} = [e_1]^* \cdot \lceil [f] \rceil_{\mathsf{Exp}/\equiv} \leq \lceil [e \cdot f] \rceil_{\mathsf{Exp}/\equiv} \qquad \square$$

5.5 Leveraging completeness

Completeness now enables the use of coinduction to prove that expressions are provably equivalent using the axioms of Kleene algebra.

Let us illustrate the proof of the denesting rule $(a+b)^* \equiv (a^*b)^*a^*$, which was left as an exercise in Chapter 2. We construct the relation

$$R = \{\langle (a+b)^*, (a^*b)^*a^* \rangle, \langle (a+b)^*, (a^*b)(a^*b)^*a^* + a^* \rangle\}$$

and observe that it is a bisimulation (in the calculations below we use some simplifications, which is no problem because that provable equivalence is a bisimulation; this means that actually the relation R above is formally a bisimulation up to a finite set of (sound) axioms):

$$\begin{aligned}
((a+b)^*)_a &= (a+b)^* \\
((a+b)^*)_b &= (a+b)^* \\
((a^*b)^*a^*)_a &= (a^*b)(a^*b)^*a^* + a^* \\
((a^*b)^*a^*)_b &= (a^*b)^*a^* \\
((a^*b)(a^*b)^*a^* + a^*)_a &= (a^*b)(a^*b)^*a^* + a^* \\
((a^*b)(a^*b)^*a^* + a^*)_b &= (a^*b)^*a^*
\end{aligned}$$

The output values of all the expressions above are 1. Thus, $(a+b)^* \sim (a^*b)^*a^*$ which implies, by completeness, $(a+b)^* \equiv (a^*b)^*a^*$.

For another example, which we looked at the end of Section 4.3.1, take the expressions $(b+ab^*a)^*$ and $b^*a(ab^*a+b)^*ab^* + b^*$. The relation

$$R = \{\langle (b+ab^*a)^*, b^*a(ab^*a+b)^*ab^*+b^* \rangle, \langle b^*a(b+ab^*a)^*, (ab^*a+b)^*ab^* \rangle\}$$

is a bisimulation. It is very easy to check: we show the automaton structure underlying each expression:

$$\begin{tikzcd}
\boxed{\boxed{(b+ab^*a)^*}} \arrow[loop left, "b"] \arrow[d, bend left, "a"] & & \boxed{\boxed{b^*a(ab^*a+b)^*ab^*+b^*}} \arrow[loop right, "b"] \arrow[d, bend left, "a"] \\
\boxed{b^*a(b+ab^*a)^*} \arrow[loop left, "b"] \arrow[u, bend left, "a"] & & \boxed{(ab^*a+b)^*ab^*} \arrow[loop right, "b"] \arrow[u, bend left, "a"]
\end{tikzcd}$$

The algebraic proof requires a bit more of ingenuity, using the denesting (left as Exercise 2.6) and shifting rule (proved in Lemma 2.23):

$$\begin{aligned}
(b+ab^*a)^* &\equiv (b^*ab^*a)^*b^* && \text{(denesting rule)}\\
&\equiv b^*(ab^*a^*b^*)^* && \text{(shifting rule)}\\
&\equiv b^*(ab^*a^*b^*(ab^*a^*b^*)^*+1) && (e^* \equiv 1+ee^*)\\
&\equiv b^*ab^*a^*b^*(ab^*a^*b^*)^*+b^* \\
& && \text{(left distributivity and } e1 \equiv e)\\
&\equiv b^*a(b^*a^*b^*a)^*b^*a^*b^*+b^* && \text{(shifting rule)}\\
&\equiv b^*a(b+ab^*a)^*a^*b^*+b^* && \text{(denesting rule)}\\
&\equiv b^*a(ab^*a+b)^*a^*b^*+b^* && \text{(commutativity of +)}
\end{aligned}$$

5.6 Exercises

5.1. Let Q be a finite set, and let X, Y and Z be Q-matrices. Prove the left-distributivity claimed in Lemma 5.3, namely that

$$X \cdot (Y+Z) \equiv X \cdot Y + X \cdot Z$$

5.2. Let $\mathcal{K}$ be a Kleene algebra, and let $M\colon Q \times Q \to \mathcal{K}$.

(a) Show that if M is finite and $q \in Q$, then $R_M(q) = R_{M^*}(q)$. *Hint: use that* $M^* = \mathbf{1} + M \cdot M^*$.

(b) Show the same property, assuming that M is locally finite.

5.3. Let Q_0 and Q_1 be finite sets, let M be a Q_0-by-Q_1 matrix, let X be a Q_1-matrix, and let Y be a Q_0-matrix.

Show that $M \cdot X \equiv Y \cdot M$ implies $M \cdot X^* \equiv Y^* \cdot M$.

Hint: if you're stuck, suppose $e, f, g \in \mathsf{Exp}$ with $e \cdot f \equiv g \cdot e$, and prove that $e \cdot f^* \equiv g^* \cdot e$ (c.f. Exercise 2.8). The structure of this proof can be adapted to your needs.

5.4. Complete the proof of Lemma 5.4. That is, show that the s proposed in the proof is such that $b + M \cdot s \leqq s$, and furthermore if t is a Q-vector and $z \in \mathsf{Exp}$ with $b \fatsemi z + M \cdot t \leqq t$, then $s \fatsemi z \leqq t$.

Hint: For the second property, first derive the following from the assumption that t is a Q-vector such that $b \fatsemi z + M \cdot t \leqq t$:

$$M(p,p)^* \cdot \Big(b(p) \cdot z + \sum_{q' \in Q'} M(p,q') \cdot t(q')\Big) \leqq t(p)$$

Then use the induction hypothesis on a Q'-vector t' where $t'(q) = t(q)$ to complete the proof.

5.5. In this exercise, we will prove the missing axioms about the Kleene star operator applied to a finite matrix M, i.e., that (1) $M^* \cdot M + \mathbf{1} = M^*$ and (2) if $B + S \cdot M \leq S$, then $B \cdot M^* \leq S$.

(a) Replay the arguments from Lemmas 5.4 and 5.5 to define, for finite matrices M, a matrix $M^\dagger$ such that (1) $M^\dagger \cdot M + \mathbf{1} = M^\dagger$ and (2) if $B + S \cdot M \leq S$, then $B \cdot M^\dagger \leq S$.

(b) Use the properties of $M^\dagger$ proved above, as well as the properties of M^* from Lemma 5.5 and Corollary 5.6 to prove that for finite matrices M, we have that $M^* = M^\dagger$.

(c) Conclude that M^* satisfies properties (1) and (2) above.

5.6. We consider some of the remaining cases of Theorem 5.16:

(a) Show that $o((e_1 \cdot e_2) \cdot e_3) = o(e_1 \cdot (e_2 \cdot e_3))$ and $((e_1 \cdot e_2) \cdot e_3)_a \equiv (e_1 \cdot (e_2 \cdot e_3))_a$ (one of the base cases).

(b) Show that if $o(e_1) = o(e_2)$ and $(e_1)_a \equiv (e_2)_a$, then $o(e_1 \cdot f) = o(e_2 \cdot f)$ and $(e_1 \cdot f)_a \equiv (e_2 \cdot f)_a$ (one of the inductive steps).

5.7. In this exercise, we prove Lemma 5.20: if $e \leqq f$, then $\lceil [e] \rceil_{\mathsf{Exp}/_\equiv} \leq \lceil [f] \rceil_{\mathsf{Exp}/_\equiv}$. First, we need the matrix M_+ over $\mathsf{Exp}/_\equiv$ given by

$M_+([g],[h]) = 1$ if $g \equiv h + f$, and $M_+([g],[h]) = 0$ otherwise. We also define the right-multiplication of a vector b by a matrix M over Q as another vector over Q, in the expected way:

$$(b \cdot M)(q) = \sum_{q'} b(q') \cdot M(q', q)$$

This multiplication interacts well with other matrix-matrix and matrix-vector multiplications, so we can drop parentheses.

(a) Prove for any $b\colon \mathsf{Exp}/_{\equiv} \to \mathsf{Exp}/_{\equiv}$ and any $g \in \mathsf{Exp}$ we have that $(b \cdot M_+)([g]) = b([g + f])$.

(b) Use the fixpoint laws to show that $M^*_{\mathsf{Exp}/_{\equiv}} \cdot o \leq M^*_{\mathsf{Exp}/_{\equiv}} \cdot o \cdot M_+$.

(c) Use the two facts above to conclude $\lceil [e] \rceil_{\mathsf{Exp}/_{\equiv}} \leq \lceil [f] \rceil_{\mathsf{Exp}/_{\equiv}}$.

5.8. Let R and S be relations. Show that $M_{R \circ S} \equiv M_R \cdot M_S$.

5.9. Let R be a relation. Show that $M_{R^*} \equiv M_R^*$.

Hint: show that $M_R^* \leqq M_{R^*}$ and $M_{R^*} \leqq M_R^*$. For both inclusions, you need to use facts about the star of a matrix, but the implication is only necessary for one of them.

6

Kleene algebra with tests

Previous chapters studied Kleene Algebra and equational reasoning over regular expressions. As we mentioned in the introduction, regular expressions offer a useful algebraic perspective to reason about imperative programs. However, in order to increase the breadth of reasoning and more faithfully model imperative programs we need to precisely capture *control flow* structures such as conditionals and while loops. Recall the two while programs from the introduction.

```
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;
```

In this chapter we want to show how we can encode and reason about equivalence of these programs using regular expressions in a way that the effects of control flow are taken as first class citizens. The key piece missing in regular expressions is a distinguished notion of *test* that enables the conditional execution of program actions. We will explain how one can develop a theory of regular expressions—retaining all the elements from the previous chapters—in which we can have both tests and program actions. This will lead to the notion of Kleene algebra with tests (KAT), an equational system that combines Kleene and Boolean algebra originally proposed by Kozen [23]. Within KAT one can model imperative programming constructs and assertions, which can then be used for its application in compiler optimization, program transformation, or dataflow analysis [24], [25], [26]. For instance, we will encode conditionals and loops as follows:

$$\texttt{while a do } e \triangleq (\mathtt{a}e)^*\overline{\mathtt{a}} \qquad \texttt{if a then } e \texttt{ else } f \triangleq \mathtt{a}e + \overline{\mathtt{a}}f$$

More recently, a variant of KAT has been used to model forwarding behavior of such networks [27]. Due to its tractable equational theory, it was used as a foundation for verification of important properties of networks, such as reachability and access control.

6.1 Syntax and Semantics

In this section we describe the syntax and semantics of KAT, where we treat both the language and relational interpretation of KAT expressions.

In order to reason about common programming constructs such as if-then-else statements and while-loops, we need a notion of conditional in our syntax. To this end, Boolean algebra was considered in combination with Kleene algebra, where the Boolean algebra provides tests that are used to express these conditional programming constructs. Formally, this leads to the following definition of a Kleene algebra with tests:

Definition 6.1 (Kleene algebra with tests). A Kleene algebra with tests is a two-sorted structure $(A, B, +, \cdot, -^*, ^-, 0, 1)$ where

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

The operator $^-$ denotes negation.

We unpack this definition. A Kleene algebra was defined in Definition 2.18. A Boolean algebra $(B, +, \cdot, ^-, 0, 1)$ consists of a nonempty set B with two distinguished elements 0 and 1, two binary operations $+$ and $\cdot$ (usually omitted when writing the expressions) and a unary operation $^-$ satisfying the following laws for all $a, b, c \in B$:

$$\begin{aligned}
a + 0 = a = a \cdot 1 &\qquad a \cdot (b \cdot c) = (a \cdot b) \cdot c \\
a + b = b + a &\qquad a + (b + c) = (a + b) + c \\
a \cdot b = b \cdot a &\qquad a + (a \cdot b) = a = a \cdot (a + b) \\
0 = a \cdot \overline{a} &\qquad a + (b \cdot c) = (a + b) \cdot (a + c) \\
1 = a + \overline{a} &\qquad a \cdot (b + c) = (a \cdot b) + (a \cdot c)
\end{aligned}$$

The last item of Definition 6.1 ensures that $+$ (respectively $\cdot$, 0, 1) in the Boolean algebra coincides with $+$ (respectively $\cdot$, 0, 1) in the Kleene algebra.

Given a set P of (primitive) action symbols and a set B of (primitive) test symbols, we can define the free Kleene algebra with tests on generators $\mathsf{P} \cup \mathsf{B}$ as follows. Syntactically, the set BExp of Boolean tests is given by:

$$\mathsf{BExp} \ni b ::= \mathtt{b} \in \mathsf{B} \mid b_1 b_2 \mid b_1 + b_2 \mid \overline{b} \mid \mathsf{0} \mid \mathsf{1}$$

The set of KAT expressions is given by

$$\mathsf{KatExp} \ni e, f ::= \mathtt{p} \in \mathsf{P} \mid b \in \mathsf{BExp} \mid ef \mid e + f \mid e^*$$

The free Kleene algebra with tests on generators $\mathsf{P} \cup \mathsf{B}$ is obtained by quotienting BExp by the axioms of Boolean algebra and KatExp by the axioms of Kleene algebra.

We denoted provable equivalence of two KA terms e, f with $e \equiv f$. We now generalise this notation and write $e \equiv_{\mathsf{KAT}} f$ to denote that two KA expressions are provable equivalent using the axioms of KAT, and $b \equiv_{\mathsf{BA}} c$ to denote that two Boolean algebra terms b and c are provably equivalent using the axioms of BA. Similarly, in this chapter we sometimes write $e \equiv_{\mathsf{KA}} f$ to make it clear e and f are provably equivalent using the axioms of KA.

Example 6.2 (Basic conditional laws). We prove, using the axioms of Boolean and Kleene algebra, two familiar identities about conditionals.

$$\texttt{if } \mathsf{a} \texttt{ then } e \texttt{ else } e = e$$

and

$$\texttt{if } \mathsf{a} \texttt{ then } (\texttt{if } \bar{\mathsf{a}} \texttt{ then } e) \texttt{ else } f = \texttt{if } \bar{\mathsf{a}} \texttt{ then } f$$

Using our encodings the above laws translate to the following provable equivalence of KAT terms:

$$\mathsf{a}e + \bar{\mathsf{a}}e \equiv e \qquad \text{and} \qquad \mathsf{a}(\bar{\mathsf{a}}e) + \bar{\mathsf{a}}f \equiv \bar{\mathsf{a}}f$$

These in turn can be proved as follows.

$$\begin{aligned} \mathsf{a}e + \bar{\mathsf{a}}e &\equiv (\mathsf{a} + \bar{\mathsf{a}})e && \text{(distributivity)} \\ &\equiv 1e && (\mathsf{a} + \bar{\mathsf{a}} \equiv 1) \\ &\equiv e && (1e \equiv e) \end{aligned}$$

$$\begin{aligned} \mathsf{a}(\bar{\mathsf{a}}e) + \bar{\mathsf{a}}f &\equiv (\mathsf{a}\bar{\mathsf{a}})e + \bar{\mathsf{a}}f && \text{(associativity)} \\ &\equiv 0e + \bar{\mathsf{a}}f && (\mathsf{a}\bar{\mathsf{a}} \equiv 0) \\ &\equiv 0 + \bar{\mathsf{a}}f && (0e \equiv 0) \\ &\equiv \bar{\mathsf{a}}f && (0 + e \equiv e) \end{aligned}$$

6.1.1 Language semantics

Guarded strings were introduced in [28] as an abstract interpretation for program schemes. They are like ordinary strings over an input alphabet P, but the symbols in P alternate with the atoms of the free Boolean algebra generated by B. The set At of atoms is given by $\mathsf{At} = 2^{\mathsf{B}}$. We define the set GS of guarded strings by

$$\mathsf{GS} = (\mathsf{At} \times \mathsf{P})^* \mathsf{At}$$

Kozen [29] showed that the regular sets of guarded strings play the same role in KAT as regular languages play in Kleene algebra (both sets are actually the final coalgebra of a given functor).

Definition 6.3 (Language semantics)**.** Each KAT expression e denotes a set $[\![e]\!]$ of guarded strings defined inductively on the structure of e as follows:

$$\begin{array}{lcl} [\![\mathtt{p}]\!] & = & \{\alpha \mathtt{p} \beta \mid \alpha, \beta \in \mathsf{At}\} \\ [\![b]\!] & = & \{\alpha \mid \alpha \leqq b\} \\ [\![e+f]\!] & = & [\![e]\!] \cup [\![f]\!] \\ [\![ef]\!] & = & [\![e]\!] \diamond [\![f]\!] \\ [\![e^*]\!] & = & \bigcup_{n \geq 0} [\![e]\!]^n \end{array}$$

where $\diamond$ is the fusion product defined as follows. Given two guarded strings $x = \alpha_0 \mathtt{p}_0 \ldots \mathtt{p}_{\mathtt{n}-1} \alpha_n$ and $y = \beta_0 \mathtt{q}_1 \ldots \mathtt{q}_{\mathtt{n}-1} \beta_n$, we define the fusion product of x and y by $x \diamond y = \alpha_0 \mathtt{p}_0 \ldots \mathtt{p}_{\mathtt{n}} \alpha_n \mathtt{q}_1 \ldots \mathtt{q}_{\mathtt{n}-1} \beta_n$, if $\alpha_n = \beta_0$, otherwise $x \diamond y$ is undefined. Then, given $X, Y \subseteq \mathsf{GS}$, $X \diamond Y$ is the set containing all existing fusion products $x \diamond y$ of $x \in X$ and $y \in Y$ and X^n is defined inductively as $X^0 = X$ and $X^{n+1} = X \diamond X^n$.

A set of guarded strings is regular if it is equal to $[\![e]\!]$ for some KAT expression e. Note that a guarded string x is itself a KAT expression and $[\![x]\!] = \{x\}$.

Example 6.4. Consider the KAT expression $\mathtt{a} + \mathtt{b}\mathtt{p}$ over $\mathsf{B} = \{\mathtt{a}, \mathtt{b}\}$ and $\mathsf{P} = \{\mathtt{p}\}$. We compute tits language semantics as follows:

$$\begin{array}{lcl} [\![\mathtt{a} + \mathtt{b}\mathtt{p}]\!] & = & [\![\mathtt{a}]\!] \cup ([\![\mathtt{b}]\!] \diamond [\![\mathtt{p}]\!]) \\ & = & \{\alpha \mid \alpha \leqq \mathtt{a}\} \cup (\{\alpha \mid \alpha \leqq \mathtt{b}\} \diamond \{\alpha \mathtt{p} \beta \mid \alpha, \beta \in \mathsf{At}\}) \\ & = & \{\alpha \mid \alpha \leqq \mathtt{a}\} \cup \{\alpha \mathtt{p} \beta \mid \alpha \leqq \mathtt{b}, \beta \in \mathsf{At}\} \end{array}$$

Now consider the expression $\mathtt{a}e + \bar{\mathtt{a}}e$ and note that its semantics can be computed as follows:

$$\begin{array}{lcl} [\![\mathtt{a}e + \bar{\mathtt{a}}e]\!] & = & [\![\mathtt{a}]\!] \diamond [\![e]\!] \cup [\![\bar{\mathtt{a}}]\!] \diamond [\![e]\!] \\ & = & ([\![\mathtt{a}]\!] \cup [\![\bar{\mathtt{a}}]\!]) \diamond [\![e]\!] \\ & = & (\{\alpha \mid \alpha \leqq \mathtt{a}\} \cup \ \{\alpha \mid \alpha \leqq \bar{\mathtt{a}}\}) \diamond [\![e]\!] \\ & = & \mathsf{At} \diamond [\![e]\!] \\ & = & [\![e]\!] \end{array}$$

Recall we had proved in Example 6.2 that $\mathtt{a}e + \bar{\mathtt{a}}e$ is provably equivalent to e so the above confirms the soundness of reasoning. Indeed, as we will see all the axioms of KAT are sound (and complete!) wrt reasoning about the guarded string semantics.

6.1.2 Relational semantics

This subsection gives an interpretation of KAT expressions as binary relations, a common model of input-output behavior for many programming languages. We show that the language model is sound and complete for this interpretation. Thus KAT equivalence implies program equivalence for any programming language with a suitable relational semantics.

Definition 6.5 (Relational Interpretation). Let $i = (\mathsf{State}, \mathsf{eval}, \mathsf{sat})$ be a triple consisting of a set of *states* State; for each action $\mathtt{p} \in \mathsf{P}$, a binary relation $\mathsf{eval}(\mathtt{p}) \subseteq \mathsf{State} \times \mathsf{State}$; and for each primitive test $\mathtt{b} \in \mathsf{B}$, a set of states $\mathsf{sat}(\mathtt{b}) \subseteq \mathsf{State}$. Then we define the *relational interpretation* of an expression e with respect to i, written $[\![-]\!]_i \colon \mathsf{KatExp} \to 2^{\mathsf{State}\times\mathsf{State}}$, inductively as follows:

$$[\![\mathtt{p}]\!]_i = \mathsf{eval}(\mathtt{p}) \qquad [\![b]\!]_i = \triangle(\mathsf{sat}^\dagger(b)) \qquad [\![e^*]\!]_i = [\![e]\!]_i^*$$
$$[\![e \cdot f]\!]_i = [\![e]\!]_i \circ [\![f]\!]_i \qquad [\![e + f]\!]_i = [\![e]\!]_i \cup [\![f]\!]_i$$

where $\mathsf{sat}^\dagger \colon \mathsf{BExp} \to 2^{\mathsf{State}}$ is the usual lifting of $\mathsf{sat} \colon \mathsf{B} \to 2^{\mathsf{State}}$ to Boolean expressions over B; $\triangle \colon 2^X \to 2^{X\times X}$ is the diagonal relation operator on sets (that is, from a set X it creates the relation with every pair (x, x)); and, as before, for a relation R we use R^* to denote the reflexive-transitive closure of a relation.

Example 6.6. Let us consider again the expression $\mathtt{a}+\mathtt{bp}$ over $\mathsf{B} = \{\mathtt{a}, \mathtt{b}\}$ and $\mathsf{P} = \{\mathtt{p}\}$ and the triple $i = (\mathsf{State}, \mathsf{eval}, \mathsf{sat})$ given by $\mathsf{State} = \{s_1, s_2\}$, $\mathsf{eval}(\mathtt{p}) = \{(s_1, s_2)\}$, and $\mathsf{sat}(\mathtt{a}) = \{s_2\}, \mathsf{sat}(\mathtt{b}) = \{s_1, s_2\}$.

It is easy to see that $[\![\mathtt{a}]\!]_i = \{(s_2, s_2)\}$, $[\![\mathtt{b}]\!]_i = \{(s_1, s_1), (s_2, s_2)\}$ and $[\![\mathtt{p}]\!]_i = \{(s_1, s_2)\}$. Putting it all together we obtain:

$$[\![\mathtt{a} + \mathtt{bp}]\!]_i = \{(s_2, s_2), (s_1, s_2)\}$$

The following result says that the language model from the previous section abstracts the various relational interpretations in a sound and complete way. It was first proved for KAT by Kozen *et al.* [23].

Theorem 6.7. The language model is sound and complete for the relational model:

$$[\![e]\!] = [\![f]\!] \quad\iff\quad \forall i.\, [\![e]\!]_i = [\![f]\!]_i$$

It is worth noting that Theorem 6.7 also holds for refinement (i.e., with $\subseteq$ instead of $=$).

6.2 Automata and Bisimulation

Later in [30], Kozen showed that the deterministic version of automata on guarded strings (already defined in [29]) fits neatly in the coalgebraic framework: two expressions are bisimilar if and only if they recognize the same set of guarded strings.

Definition 6.8 (KAT automata)**.** A (deterministic) KAT automaton is a pair $(S, \langle o_S, t_S\rangle)$ where $o_S \colon S \to 2^{\mathsf{At}}$ and $t_S \colon S \to S^{\mathsf{At}\times\mathsf{P}}$.

It might be worth noting that 2^{At} is the free Boolean algebra on B– this fact is important for the connection with the denotational semantics. KAT automata are F-coalgebras for $F(X) = 2^{\mathsf{At}} \times X^{\mathsf{At}\times P}$.

We can obtain a KAT automaton corresponding to each KAT expression by using the following generalization of Brzozowski derivatives for KAT expressions[1].

Definition 6.9 (Brzozowski derivatives for KAT expressions)**.** Given a KAT expression $e \in \mathsf{KatExp}$, we define $E\colon \mathsf{KatExp} \to 2^{\mathsf{At}}$ and $D\colon \mathsf{KatExp} \to \mathsf{KatExp}^{\mathsf{At}\times\mathsf{P}}$ by induction on the structure of e. First, $E(e)$ is given by:

$$E(\mathtt{p}) = \emptyset \qquad E(\mathtt{b}) = \{\alpha \in \mathsf{At} \mid \alpha \leqq \mathtt{b}\} \qquad E(ef) = E(e) \cap E(f)$$
$$E(e+f) = E(e) \cup E(f) \qquad E(e^*) = \mathsf{At}$$

Next, we define $e_{\alpha\mathtt{q}} = D(e)(\langle \alpha, \mathtt{q}\rangle)$ by

$$\mathtt{p}_{\alpha\mathtt{q}} = \begin{cases} 1 & \text{if } \mathtt{p} = \mathtt{q} \\ 0 & \text{if } \mathtt{p} \neq \mathtt{q} \end{cases} \qquad \mathtt{b}_{\alpha\mathtt{q}} = 0 \quad (ef)_{\alpha\mathtt{q}} = \begin{cases} e_{\alpha\mathtt{q}} f + f_{\alpha\mathtt{q}} & \text{if } \alpha \in E(e) \\ e_{\alpha\mathtt{q}} f & \text{if } \alpha \notin E(e) \end{cases}$$

$$(e+f)_{\alpha\mathtt{q}} = e_{\alpha\mathtt{q}} + f_{\alpha\mathtt{q}} \qquad\qquad (e^*)_{\alpha\mathtt{q}} = e_{\alpha\mathtt{q}} e^*$$

The functions $\langle E, D\rangle$ provide KatExp with the structure of an F-coalgebra for $F(X) = 2^{\mathsf{At}} \times X^{\mathsf{At}\times P}$. This leads, by finality, to the

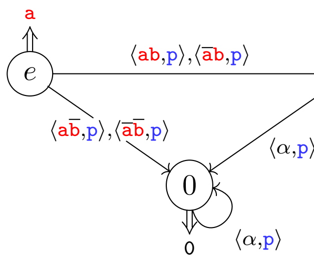
[1]As with regular expressions to ensure the set of derivatives is finite we need to take them modulo ACI.

existence of a unique homomorphism

$$\begin{tikzcd}
\mathsf{KatExp} \arrow[r, dashed, "L"] \arrow[d, "\langle E,D\rangle"'] & (2^{\mathsf{At}})^{(\mathsf{At}\times\mathsf{P})^*} \cong 2^{\mathsf{GS}} \arrow[d] \\
2^{\mathsf{At}} \times \mathsf{KatExp}^{\mathsf{At}\times\mathsf{P}} \arrow[r, dashed] & 2^{\mathsf{At}} \times (2^{\mathsf{GS}})^{\mathsf{At}\times\mathsf{P}}
\end{tikzcd}$$

which assigns to each expression the language of guarded strings that it denotes. In other words, $L(e) = [\![e]\!]$.

Example 6.10. The automaton corresponding to $e = \mathtt{a} + \mathtt{bp}$ would be



a
1
e
1
0
0
$\langle \mathtt{ab},\mathtt{p}\rangle, \langle \bar{\mathtt{a}}\mathtt{b},\mathtt{p}\rangle$
$\langle \mathtt{a}\bar{\mathtt{b}},\mathtt{p}\rangle, \langle \bar{\mathtt{a}}\bar{\mathtt{b}},\mathtt{p}\rangle$
$\langle \alpha,\mathtt{p}\rangle$
$\langle \alpha,\mathtt{p}\rangle$

since, for $\mathsf{B} = \{\mathtt{a}, \mathtt{b}\}$, $\mathsf{At} = \{\mathtt{ab}, \mathtt{a}\bar{\mathtt{b}}, \bar{\mathtt{a}}\mathtt{b}, \bar{\mathtt{a}}\bar{\mathtt{b}}\}$ and

$$e_{\mathtt{ab},\mathtt{p}} = 0 + (\mathtt{bp})_{\mathtt{ab},\mathtt{p}} = \mathtt{p}_{\mathtt{ab},\mathtt{p}} = 1 \qquad e_{\bar{\mathtt{a}}\mathtt{b},\mathtt{p}} = 0 + (\mathtt{bp})_{\bar{\mathtt{a}}\mathtt{b},\mathtt{p}} = \mathtt{p}_{\bar{\mathtt{a}}\mathtt{b},\mathtt{p}} = 1$$
$$e_{\mathtt{a}\bar{\mathtt{b}},\mathtt{p}} = 0 + (\mathtt{bp})_{\mathtt{a}\bar{\mathtt{b}},\mathtt{p}} = 0 \qquad e_{\bar{\mathtt{a}}\bar{\mathtt{b}},\mathtt{p}} = 0$$

$$E(e) = \{\alpha \mid \alpha \leqq \mathtt{a}\} = \{\mathtt{ab}, \mathtt{a}\bar{\mathtt{b}}\} \quad E(0) = \emptyset \quad E(1) = \mathsf{At}$$

Above we represent the output $o_S(s)$ of a state by $\Rightarrow b$ where $b \in 2^{\mathsf{At}}$.

We now instantiate the definition of coalgebraic bisimulation for KAT automata, which are F-coalgebras for $F(X) = 2^{\mathsf{At}} \times X^{\mathsf{At}\times P}$.

Definition 6.11 (KAT bisimulation)**.** Given two KAT automata $(S, \langle o_X, t_X\rangle)$ and $(Y, \langle o_Y, t_Y\rangle)$ we say that $R \subseteq X \times Y$ is a bisimulation if for all $(x, y) \in R$ it holds:

1. $o_X(x) = o_Y(y)$, and
2. for all $\alpha \in \mathsf{At}$ and $\mathtt{p} \in \mathsf{P}$, $(t_X(x)(\alpha, \mathtt{p}), t_Y(y)(\alpha, \mathtt{p})) \in R$.

We denote the largest KAT bisimulation by $\sim$, and we can use it as a proof principle to prove the equivalence of expressions.

Theorem 6.12 (KAT coinduction)**.** Let e, f be KAT expressions.

$$e \sim f \Rightarrow [\![e]\!] = [\![f]\!]$$

Proof. Recall that the automaton structure on KAT expressions is given by Brzozowski derivative $\langle E, D\rangle$. We prove $[\![e]\!] = [\![f]\!]$ induction on the length of guarded strings. The base case is $\alpha \in \mathsf{At}$ and we use the fact that since $e \sim f$ then $E(e) = E(f)$:

$$\alpha \in [\![e]\!] \iff \alpha \in E(e) \iff \alpha \in E(f) \iff \alpha \in [\![f]\!]$$

The inductive case is a guarded string $\alpha \mathtt{p} x$. Note that because $e \sim f$ then we have that, for all $\alpha \in \mathsf{At}$ and $\mathtt{p} \in \mathsf{P}$, $D(e)(\alpha\mathtt{p}) \sim D(f)(\alpha\mathtt{p})$. The induction hypothesis gives us that for any $e \sim f$, it holds $x \in [\![e]\!] \iff x \in [\![f]\!]$. We can now put these two facts together and reason:

$$\begin{aligned} \alpha\mathtt{p}x \in [\![e]\!] &\iff x \in D(e)(\alpha\mathtt{p}) \\ &\iff x \in D(f)(\alpha\mathtt{p}) \\ &\iff \alpha\mathtt{p}x \in [\![f]\!] \end{aligned}$$ □

Theorem 6.12 provides an alternative way to prove two expressions are equivalent: build the automaton corresponding to each expression using Brzozowski derivatives and then check if they are bisimilar.

Example 6.13. Consider the expressions

$$(\mathtt{a}(\mathtt{bp} + \bar{\mathtt{b}}\mathtt{q}))^*\bar{\mathtt{a}} \qquad \text{and} \qquad (\mathtt{abp})^*(\bar{\mathtt{a}} + \bar{\mathtt{b}})(\mathtt{aq}(\mathtt{abp})^*(\bar{\mathtt{a}} + \bar{\mathtt{b}}))^*\bar{\mathtt{a}}$$

These are the encodings of the programs in the introduction (with e and f instantiated to $\mathtt{p}$ and $\mathtt{q}$ so that we can build the whole automata). We now present two alternative proofs of this fact: an axiomatic derivation and a bisimulation proof.

Axiomatic derivation: We first rewrite both terms, and then prove equality through two inequalities.

$$\begin{aligned} (\mathtt{a}(\mathtt{bp} + \bar{\mathtt{b}}\mathtt{q}))^*\bar{\mathtt{a}} &\equiv (\mathtt{abp} + \mathtt{a}\bar{\mathtt{b}}\mathtt{q})^*\bar{\mathtt{a}} && \text{(distributivity)} \\ &\equiv ((\mathtt{abp})^*\mathtt{a}\bar{\mathtt{b}}\mathtt{q})^*(\mathtt{abp})^*\bar{\mathtt{a}} && \text{(Denesting, ??)} \\ &\equiv (\mathtt{abp})^*(\mathtt{a}\bar{\mathtt{b}}\mathtt{q}(\mathtt{abp})^*)^*\bar{\mathtt{a}} && \text{(Lemma 2.23)} \end{aligned}$$

For the second term, we first prove three intermediate results:

$$\overline{\mathtt{a}}(\mathtt{a}e)^* \equiv \overline{\mathtt{a}} \tag{6.1}$$

The right-to-left direction is immediate, as $(\mathtt{a}e)^* \equiv 1 + (\mathtt{a}e)(\mathtt{a}e)^*$. For the left-to-right direction we observe that

$$\overline{\mathtt{a}} + \overline{\mathtt{a}}(\mathtt{a}e) \leqq \overline{\mathtt{a}}$$

because $\overline{\mathtt{a}}(\mathtt{a}e) \equiv \mathtt{0}$. Then we obtain the left-to-right direction of 6.1 immediately from the star axioms.

With 6.1 we can prove the following:

$$(\mathtt{a}e\overline{\mathtt{a}})^* \equiv 1 + \mathtt{a}e\overline{\mathtt{a}}(\mathtt{a}e\overline{\mathtt{a}})^* \equiv 1 + \mathtt{a}e\overline{\mathtt{a}} \tag{6.2}$$

Below we abbreviate $\mathtt{aq}(\mathtt{abp})^*$ with x.

$$\begin{aligned}
&(\mathtt{abp})^*\overline{\mathtt{b}}(x(\overline{\mathtt{a}} + \overline{\mathtt{b}}))^*\overline{\mathtt{a}} \\
&\equiv (\mathtt{abp})^*\overline{\mathtt{b}}(x\overline{\mathtt{a}} + x\overline{\mathtt{b}})^*\overline{\mathtt{a}} && \text{(distributivity)} \\
&\equiv (\mathtt{abp})^*\overline{\mathtt{b}}((x\overline{\mathtt{a}})^*x\overline{\mathtt{b}})^*(x\overline{\mathtt{a}})^*\overline{\mathtt{a}} && (\mathbf{??}) \\
&\equiv (\mathtt{abp})^*(\overline{\mathtt{b}}(x\overline{\mathtt{a}})^*x)^*\overline{\mathtt{b}}(x\overline{\mathtt{a}})^*\overline{\mathtt{a}} && \text{(Lemma 2.23)} \\
&\equiv (\mathtt{abp})^*(\overline{\mathtt{b}}(1 + x\overline{\mathtt{a}})x)^*\overline{\mathtt{b}}(1 + x\overline{\mathtt{a}})\overline{\mathtt{a}} && (6.2) \\
&\equiv (\mathtt{abp})^*(\overline{\mathtt{b}}x + \overline{\mathtt{b}}x\overline{\mathtt{a}}x)^*(\overline{\mathtt{b}}\overline{\mathtt{a}} + \overline{\mathtt{b}}x\overline{\mathtt{a}}) && \text{(distributivity)} \\
&\equiv (\mathtt{abp})^*(\overline{\mathtt{b}}x)^*(\overline{\mathtt{b}}\overline{\mathtt{a}} + \overline{\mathtt{b}}x\overline{\mathtt{a}}) && (x = \mathtt{aq}(\mathtt{abp})^*,\ \overline{\mathtt{a}}\mathtt{a} \equiv 0)
\end{aligned}$$

Then we rewrite the second program:

$$\begin{aligned}
&(\mathtt{abp})^*(\overline{\mathtt{a}} + \overline{\mathtt{b}})(x(\overline{\mathtt{a}} + \overline{\mathtt{b}}))^*\overline{\mathtt{a}} \\
&\equiv (\mathtt{abp})^*\overline{\mathtt{a}}(x(\overline{\mathtt{a}} + \overline{\mathtt{b}}))^*\overline{\mathtt{a}} + (\mathtt{abp})^*\overline{\mathtt{b}}(x(\overline{\mathtt{a}} + \overline{\mathtt{b}}))^*\overline{\mathtt{a}} && \text{(distributivity)} \\
&\equiv (\mathtt{abp})^*\overline{\mathtt{a}} + (\mathtt{abp})^*(\overline{\mathtt{b}}x)^*(\overline{\mathtt{b}}\overline{\mathtt{a}} + \overline{\mathtt{b}}x\overline{\mathtt{a}}) && \text{(6.1, result above)} \\
&\equiv (\mathtt{abp})^*(1 + (\overline{\mathtt{b}}x)^*\overline{\mathtt{b}}x)\overline{\mathtt{a}} + (\mathtt{abp})^*(\overline{\mathtt{b}}x)^*\overline{\mathtt{b}}\overline{\mathtt{a}} && \text{(distributivity)} \\
&\equiv (\mathtt{abp})^*(\overline{\mathtt{b}}x)^*\overline{\mathtt{a}} + (\mathtt{abp})^*(\overline{\mathtt{b}}x)^*\overline{\mathtt{b}}\overline{\mathtt{a}} && \text{(star axiom)} \\
&\equiv (\mathtt{abp})^*(\overline{\mathtt{b}}x)^*(\overline{\mathtt{a}} + \overline{\mathtt{b}}\overline{\mathtt{a}}) && \text{(distributivity)}
\end{aligned}$$

Using that $(\mathtt{a}(\mathtt{bp} + \overline{\mathtt{b}}\mathtt{q}))^*\overline{\mathtt{a}} \equiv (\mathtt{abp})^*(\mathtt{a}\overline{\mathtt{b}}\mathtt{q}(\mathtt{abp})^*)^*\overline{\mathtt{a}} \equiv (\mathtt{abp})^*(\overline{\mathtt{b}}x)^*\overline{\mathtt{a}}$, we can conclude, where we make use of the fact that $\overline{\mathtt{a}} + \overline{\mathtt{b}}\overline{\mathtt{a}} \equiv \overline{\mathtt{a}}$.

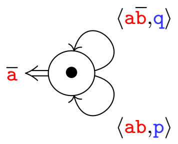
Bisimulation proof: We now calculate the derivatives of these expressions. We start with $(\mathtt{a}(\mathtt{bp}+\overline{\mathtt{b}}\mathtt{q}))^*\overline{\mathtt{a}}$ and observe that $E((\mathtt{a}(\mathtt{bp}+\overline{\mathtt{b}}\mathtt{q}))^*\overline{\mathtt{a}}) =$

$\overline{\mathsf{a}}$ and that many derivatives are equivalent to $\mathsf{0}$, namely for $\alpha\mathsf{p}$, with $\alpha = \overline{\mathsf{a}}\mathsf{b}$ or $\alpha = \overline{\mathsf{a}}\overline{\mathsf{b}}$ or $\alpha = \mathsf{a}\overline{\mathsf{b}}$, and $\beta\mathsf{q}$, with $\beta = \overline{\mathsf{a}}\mathsf{b}$ or $\beta = \overline{\mathsf{a}}\overline{\mathsf{b}}$ or $\beta = \mathsf{ab}$. It remains to compute the derivatives for ab, p and $\mathsf{a}\overline{\mathsf{b}}, \mathsf{q}$:

$$\begin{aligned}
((\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))^*\overline{\mathsf{a}})_{\mathsf{ab},\mathsf{p}} &= ((\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))^*)_{\mathsf{ab},\mathsf{p}}\overline{\mathsf{a}} + (\overline{\mathsf{a}})_{\mathsf{ab},\mathsf{p}} \\
&= (\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))_{\mathsf{ab},\mathsf{p}}(\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))^*\overline{\mathsf{a}} + \mathsf{0} \\
&= (\mathsf{0} + (\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q})_{\mathsf{ab},\mathsf{p}})(\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))^*\overline{\mathsf{a}} + \mathsf{0} \\
&= (\mathsf{0} + ((\mathsf{bp})_{\mathsf{ab},\mathsf{p}} + (\overline{\mathsf{b}}\mathsf{q})_{\mathsf{ab},\mathsf{p}}))(\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))^*\overline{\mathsf{a}} + \mathsf{0} \\
&= (\mathsf{0} + ((\mathsf{0}+\mathsf{1}) + (\mathsf{0}+\mathsf{0}))(\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))^*\overline{\mathsf{a}} + \mathsf{0}
\end{aligned}$$

$$\begin{aligned}
((\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))^*\overline{\mathsf{a}})_{\mathsf{a}\overline{\mathsf{b}},\mathsf{q}} &= ((\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))^*)_{\mathsf{a}\overline{\mathsf{b}},\mathsf{q}}\overline{\mathsf{a}} + (\overline{\mathsf{a}})_{\mathsf{a}\overline{\mathsf{b}},\mathsf{q}} \\
&= (\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))_{\mathsf{a}\overline{\mathsf{b}},\mathsf{q}}(\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))^*\overline{\mathsf{a}} + \mathsf{0} \\
&= (\mathsf{0} + (\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q})_{\mathsf{a}\overline{\mathsf{b}},\mathsf{q}})(\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))^*\overline{\mathsf{a}} + \mathsf{0} \\
&= (\mathsf{0} + ((\mathsf{bp})_{\mathsf{a}\overline{\mathsf{b}},\mathsf{q}} + (\overline{\mathsf{b}}\mathsf{q})_{\mathsf{a}\overline{\mathsf{b}},\mathsf{q}}))(\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))^*\overline{\mathsf{a}} + \mathsf{0} \\
&= (\mathsf{0} + ((\mathsf{0}+\mathsf{0}) + (\mathsf{0}+\mathsf{1}))(\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))^*\overline{\mathsf{a}} + \mathsf{0}
\end{aligned}$$

We can rewrite the last expression of both derivatives to a simpler provably equivalent expression: $(\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))^*\overline{\mathsf{a}}$, which is the expression we started with! Hence, $(\mathsf{a}(\mathsf{bp}+\overline{\mathsf{b}}\mathsf{q}))^*\overline{\mathsf{a}}$ corresponds to this automaton:

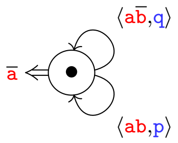


Let us now consider $(\mathsf{abp})^*(\overline{\mathsf{a}}+\overline{\mathsf{b}})(\mathsf{aq}(\mathsf{abp})^*(\overline{\mathsf{a}}+\overline{\mathsf{b}}))^*\overline{\mathsf{a}}$. Note that $E((\mathsf{abp})^*(\overline{\mathsf{a}}+\overline{\mathsf{b}})(\mathsf{aq}(\mathsf{abp})^*(\overline{\mathsf{a}}+\overline{\mathsf{b}}))^*\overline{\mathsf{a}}) = \overline{\mathsf{a}}$. As before, many derivatives are $\mathsf{0}$, namely for $\alpha\mathsf{p}$, with $\alpha = \overline{\mathsf{a}}\mathsf{b}$ or $\alpha = \overline{\mathsf{a}}\overline{\mathsf{b}}$ or $\alpha = \mathsf{a}\overline{\mathsf{b}}$, and $\beta\mathsf{q}$, with $\beta = \overline{\mathsf{a}}\mathsf{b}$ or $\beta = \overline{\mathsf{a}}\overline{\mathsf{b}}$ or $\beta = \mathsf{ab}$. We calculate the remaining derivatives:

$$\begin{aligned}
&((\mathsf{abp})^*(\overline{\mathsf{a}}+\overline{\mathsf{b}})(\mathsf{aq}(\mathsf{abp})^*(\overline{\mathsf{a}}+\overline{\mathsf{b}}))^*\overline{\mathsf{a}})_{\mathsf{ab},\mathsf{p}} \\
&= ((\mathsf{abp})^*)_{\mathsf{ab},\mathsf{p}}(\overline{\mathsf{a}}+\overline{\mathsf{b}})(\mathsf{aq}(\mathsf{abp})^*(\overline{\mathsf{a}}+\overline{\mathsf{b}}))^*\overline{\mathsf{a}} \\
&\quad + ((\overline{\mathsf{a}}+\overline{\mathsf{b}})(\mathsf{aq}(\mathsf{abp})^*(\overline{\mathsf{a}}+\overline{\mathsf{b}}))^*\overline{\mathsf{a}})_{\mathsf{ab},\mathsf{p}} \\
&= (\mathsf{abp})_{\mathsf{ab},\mathsf{p}}(\mathsf{abp})^*(\overline{\mathsf{a}}+\overline{\mathsf{b}})(\mathsf{aq}(\mathsf{abp})^*(\overline{\mathsf{a}}+\overline{\mathsf{b}}))^*\overline{\mathsf{a}} + \mathsf{0} \\
&= \mathsf{1}(\mathsf{abp})^*(\overline{\mathsf{a}}+\overline{\mathsf{b}})(\mathsf{aq}(\mathsf{abp})^*(\overline{\mathsf{a}}+\overline{\mathsf{b}}))^*\overline{\mathsf{a}} + \mathsf{0}
\end{aligned}$$

$$
\begin{aligned}
&((\mathtt{abp})^*(\bar{\mathtt{a}}+\bar{\mathtt{b}})(\mathtt{aq}(\mathtt{abp})^*(\bar{\mathtt{a}}+\bar{\mathtt{b}}))^*\bar{\mathtt{a}})_{\mathtt{a}\bar{\mathtt{b}},\mathtt{q}}\\
&= ((\mathtt{abp})^*)_{\mathtt{a}\bar{\mathtt{b}},\mathtt{q}}(\bar{\mathtt{a}}+\bar{\mathtt{b}})(\mathtt{aq}(\mathtt{abp})^*(\bar{\mathtt{a}}+\bar{\mathtt{b}}))^*\bar{\mathtt{a}}\\
&\quad + ((\bar{\mathtt{a}}+\bar{\mathtt{b}})(\mathtt{aq}(\mathtt{abp})^*(\bar{\mathtt{a}}+\bar{\mathtt{b}}))^*\bar{\mathtt{a}})_{\mathtt{a}\bar{\mathtt{b}},\mathtt{q}}\\
&= (\mathtt{abp})_{\mathtt{a}\bar{\mathtt{b}},\mathtt{q}}(\mathtt{abp})^*(\bar{\mathtt{a}}+\bar{\mathtt{b}})(\mathtt{aq}(\mathtt{abp})^*(\bar{\mathtt{a}}\\
&\quad +\bar{\mathtt{b}}))^*\bar{\mathtt{a}} + (0 + ((\mathtt{aq}(\mathtt{abp})^*(\bar{\mathtt{a}}+\bar{\mathtt{b}}))^*\bar{\mathtt{a}})_{\mathtt{a}\bar{\mathtt{b}},\mathtt{q}})\\
&= 0 + ((\mathtt{aq}(\mathtt{abp})^*(\bar{\mathtt{a}}+\bar{\mathtt{b}})))_{\mathtt{a}\bar{\mathtt{b}},\mathtt{q}}((\mathtt{aq}(\mathtt{abp})^*(\bar{\mathtt{a}}+\bar{\mathtt{b}}))^*\bar{\mathtt{a}} + (\bar{\mathtt{a}})_{\mathtt{a}\bar{\mathtt{b}},\mathtt{q}}\\
&= 0 + 1(\mathtt{abp})^*(\bar{\mathtt{a}}+\bar{\mathtt{b}})(\mathtt{aq}(\mathtt{abp})^*(\bar{\mathtt{a}}+\bar{\mathtt{b}}))^*\bar{\mathtt{a}} + 0
\end{aligned}
$$

Both derivatives simplify to $(\mathtt{abp})^*(\bar{\mathtt{a}}+\bar{\mathtt{b}})(\mathtt{aq}(\mathtt{abp})^*(\bar{\mathtt{a}}+\bar{\mathtt{b}}))^*\bar{\mathtt{a}}$, the expression we started with, and therefore the automaton corresponding to this expression is the same as the previous one (modulo renaming):


$\langle \mathtt{a}\bar{\mathtt{b}},\mathtt{q}\rangle$
$\bar{\mathtt{a}}$
$\langle \mathtt{ab},\mathtt{p}\rangle$

So the bisimulation witnessing this equivalence is the relation:

$$R = \{(e, f) \mid e \equiv \mathtt{a}(\mathtt{bp}+\bar{\mathtt{b}}\mathtt{q})^*\bar{\mathtt{a}},\, f \equiv (\mathtt{abp})^*(\bar{\mathtt{a}}+\bar{\mathtt{b}})(\mathtt{aq}(\mathtt{abp})^*(\bar{\mathtt{a}}+\bar{\mathtt{b}}))^*\bar{\mathtt{a}}\}.$$

This example shows a contrast between the axiomatic and the bisimulation reasoning. Whereas the axiomatic proof might be more insightful for the human reader, the bisimulation proof is completely mechanic and more amenable to implementations.

6.3 Completeness, a sketch

For KAT it was shown in [23] that there exists a similar result as we proved for KA in the previous chapter:

Theorem 6.14 (Soundness and Completeness KAT).

$$e \equiv_{\mathsf{KAT}} f \Leftrightarrow [\![e]\!] = [\![f]\!]$$

In this section we give a sketch of this proof, and show that it is an instance of a more general technique for proving completeness of KA with additional axioms.

The completeness result of KAT relies on two principal insights:

1. Each test can be written as a disjunction of atoms: for a test $b \in \mathsf{BExp}$, we know that
$$b \equiv_{\mathsf{BA}} \sum_{\alpha \leq b} \alpha$$
2. KAT expressions can then be viewed as plain regular expressions over the alphabet $\mathsf{P} \cup \mathsf{At}$.

Let r be the translation of a KAT expression into its equivalent version where all tests are atoms. Formally, let $r\colon \mathsf{Exp}(\mathsf{P} \cup \mathsf{BExp}) \to \mathsf{Exp}(\mathsf{P} \cup \mathsf{At})$, where $\mathsf{Exp}(\mathsf{P} \cup \mathsf{BExp})$ is simply KatExp, be the unique homomorphism such that $r(p) = p$ for $p \in \mathsf{P}$ and $r(b) = \sum_{\alpha \leq b} \alpha$ for $b \in \mathsf{BExp}$. We distinguish the KAT language interpretation in terms of guarded strings, and the language interpretation of regular expressions via $[\![-]\!]_G$ and $[\![-]\!]$.

The completeness proof then consists of the following steps for $e, f \in \mathsf{KatExp}$, where we use completeness of KA:

$$\begin{aligned} [\![e]\!]_G = [\![f]\!]_G &\Rightarrow [\![r(e)]\!] = [\![r(f)]\!] \\ &\Rightarrow r(e) \equiv_{\mathsf{KA}} r(f) \\ &\Rightarrow e \equiv_{\mathsf{KAT}} f \end{aligned}$$

This strategy for proving completeness is an instance of a more general technique, captured in a framework called *Kleene algebra with hypotheses* [3], [31], [32], [33], [34], [35], [36], [37]. Intuitively, additional axioms are added to KA, and completeness is established through completeness of KA via a notion of reduction (the function r described above). The reduction is used to capture syntactically (i.e., on the level of expressions) the behaviour of the newly added axioms, such that the regular language semantics of the reduced terms ($r(e)$ and $r(f)$ above) coincides with the semantics of the KA with additional axioms. More pointers to further reading on this topic can be found in Chapter 7.

6.4 Exercises

6.1. In the following, let $b, c \in \mathsf{BExp}$.

(a) Suppose that b and c are tests such that $b+c \equiv 1$ and $b \cdot c \equiv 0$. Prove that $b \equiv \overline{c}$.

(b) Prove that $\overline{\overline{b}} \equiv b$. *Hint: use the previous exercise.*

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

(d) 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 $b + b \equiv b$.

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

6.2. Let $b, c \in \mathsf{BExp}$ and $e, f, g \in \mathsf{KatExp}$.

(a) Show that it holds that $(b + e)^* \equiv e^*$.

(b) 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.

(c) Suppose **if** b **then** $e \cdot f$ **else** $g \leqq f$. Show that $(\textbf{while } b \textbf{ do } e) \cdot g \leqq f$.

Hint: You need to use one of the star axioms.

6.3. (a) Show, using the axioms of KAT that the following programs are equivalent: (`while` (`a` `or` `b`) e)(`if` `b` `then` f `else` g) and (`while` (`a` `or` `b`) e) g.

(b) Show the above equivalence using automata with tests and bisimulation, for e instantiated to `p`, f to `q` and g to `r`.

6.4. *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 $b, c, d \in \mathsf{BExp}$ and $e, f \in \mathsf{KatExp}$ such that $\{b\}e\{c\}$ and $\{c\}f\{d\}$ hold, prove that $\{b\}e \cdot f\{d\}$ holds.

(b) 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 $b, c \in \mathsf{BExp}$ and $e \in \mathsf{KatExp}$ 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* $e \cdot f \leqq e' \cdot f$.

7
Further Reading

Regular expressions were proposed by Kleene [38]. There exists a variety of axioms for Kleene algebra; possibilities include the ones proposed by Salomaa [39], Conway [12], Krob [40], Boffa [41], and Kozen [22]. The axioms presented in this booklet are due to Kozen [22]. The idea of encoding traditional program structures using regular expressions can also be traced back to Kozen [42].

Equivalence for Kleene algebras is efficiently decidable: in this booklet we have observed how automata can be built from regular expressions, and bisimulation can be used to determine language equivalence between automata. There are various techniques to obtain automata from regular expressions, with various drawbacks in terms of efficiency and complexity [14], [15], [16], [17], [18], [19]. Similarly, there are different strategies for establishing whether two automata are bisimilar. A recent efficient method for this problem was proposed in [43], in which one can also find an overview of (references to) other procedures for establishing bisimulations. A decision procedure for Kleene algebra that was proven correct in a proof assistant was provided in [44].

Kleene algebra captures reasoning about programs in a general setting. However, for many applications, it is interesting to impose additional structure on the algebraic theory or add extra primitives to KA. Such extensions can be useful to express additional equivalences for programs that are not captured by the (rather general) KA axioms, but are required in a specific domain.

The most prominent KA extension is KAT [42], where Kleene algebra is enriched with a Boolean algebra to represent control flow of while programs; Kleene star is used for loops, and Boolean tests are used for conditions of such loops, as well as the conditions in if-then-else statements. KAT subsumes propositional Hoare logic. It was proven to be complete and decidable in [45]. A more efficient decision procedure was developed in [46], which was certified in a proof assistant.

A variation on KAT is Kleene algebra with observations (KAO [47]), which can be used to model programs with control flow in a concurrent setting. The main difference with KAT is that tests in KAO do not follow the KAT axiom $p \wedge q = p \cdot q$. Instead, the axiom $p \wedge q \leq p \cdot q$ is used, which semantically can be understood as 'the behaviour of $p \wedge q$ is included in the behaviour of $p \cdot q$'. In a concurrent setting this is necessary, as $p \wedge q$ is one event, but between p and q in $p \cdot q$, other events can take place in a parallel thread. Hence, $p \wedge q$ and $p \cdot q$ do not necessarily have the same behaviour in a concurrent setting.

KAT has also been extended with mutable state in [48] and with networking primitives; the latter extension is known as NetKAT [27], [49]. NetKAT can be used to analyse packet behaviour in networks.

Recently, GKAT was introduced [50], to formalise the deterministic fragment of KAT. Deciding equivalence in GKAT is easier than deciding equivalence in KAT, from the point of view of complexity.

Kleene algebra with domain [51] extends KA by axioms for a domain and codomain operation. Among other things, this allows for the definition of modal operators. Modal Kleene algebra is Kleene algebra enriched by forward and backward box and diamond operators, defined via domain and codomain operations [52], [53]. Modal Kleene algebra can be used as an abstract program semantics, and it subsumes for instance propositional dynamic logic and propositional Hoare logic.

Kleene algebra with a top element has been studied in [54], [55]. Completeness results are provided for language and relational models of KA with top, and KAT with top. Kleene algebra with tests and top can be used to model incorrectness logic [56][1]. KA with top is also relevant when reasoning about abnormal termination [57].

Many of the aforementioned extensions of KA come with proofs of completeness w.r.t. the appropriate language or relational models and decidability. Such results are desirable in the context of program verification, e.g., in a proof assistant, such structures make it possible to write algebraic proofs of correctness, and to mechanize some of the steps: when two expressions e and f representing two programs happen to be provably equivalent in for instance KAT, one does not need to provide a proof, one can simply call a certified decision procedure [44], [46]. Obtaining such decidability and completeness results is often non-trivial. In this book we saw a coalgebraic proof for KA completeness. Kozen's original completeness proof [22] rewrites every regular expression to a normal form making use of a matrix representation of finite automata.

To develop a unifying perspective on extensions of KA, and a more modular approach to proving completeness and decidability for new KA extensions, a general framework called *Kleene algebra with hypotheses* [3], [31], [32], [33], [34], [35], [36], [37] was proposed, which encompasses for instance KAT, KAO and NetKAT. Kleene algebra with hypotheses comes with a canonical language model constructed from a given set of hypotheses in terms of *closed* languages. In the case of KAT for instance, the canonical language model obtained from hypotheses corresponds to the familiar interpretation of expressions as languages of guarded strings. The hypotheses framework cannot be used when the resulting languages are not regular, which is the case for instance with commutative KA [58]. For an overview of results on Kleene algebra with hypotheses, see [37].

From a logical perspective, KA with hypotheses is about axiomatizing the *Horn theory* of regular expressions, which is about proving valid *conditional* equivalences between regular expressions, like "if $\mathtt{ab} \equiv 0$, then $(\mathtt{ab})^* \equiv 1$". The challenge here is that such implications are undecidable [59], in general; the focus has therefore been in dealing with certain forms of hypotheses, such as those of the form $e \equiv 0$ [31].

Kleene Algebra assumes left-distributivity. One can also consider KA without left-distributivity, and interpret a regular expression as a

[1]The completeness proof for KAT with top given in this paper is false.

process and consider bisimulation equivalence instead. In [60] a sound axiomatisation of bisimilarity for regular expressions is provided. This axiomatisation was shown to be complete in [61].

There is limited work on bisimulation semantics for extensions of Kleene algebra. Recently, a fragment of GKAT (skip-free GKAT) was shown to be complete w.r.t. a bisimulation semantics [62]. Simulation rather than bisimulation is characterised in probabilistic versions of Kleene algebra [63], [64]. Probabilistic KA was introduced for resolving non-deterministic choices as they occur. Completeness of probabilistic KA w.r.t. an appropriate class of automata is considered in [65]. Probabilistic GKAT modulo bisimilarity is studied in [66].

The beforementioned extensions of Kleene algebra all consider programs in a sequential setting. Kleene algebras that model programs with parallel behaviour have also been studied widely. Concurrent Kleene algebra was proposed in [67], [68], and models concurrency via an axiom called the exchange law as a combination of interleaving and certain parts of programs taking place "truly concurrently" (i.e. there is no causal relation between these events). An overview of the history of CKA and the exchange law can be found in [69]. Completeness of CKA is established in [70], [71], where [70] is based on [72].

Similar to KA, CKA can also be extended with hypotheses, which is explored in [73]. Among other extensions, this framework can capture a Kleene algebra with concurrency and tests (CKAO), a version of CKA used to analyse concurrent programs with state (POCKA [74]) and an extension of CKA that can be used to reason about concurrent behaviours in networks (CNetKAT [75]). DyNetKAT [76] is a different concurrent extension of NetKAT that does not use CKA but instead uses a parallel operator in the style of process algebra.

Synchronous Kleene algebra (SKA [77]) was proposed as a way to model lock-step concurrency using KA. The appropriate language model for SKA is much simpler than for CKA, but less expressive because arbitrary interleaving between programs is not possible. In fact, SKA is an instance of Kleene algebra with hypotheses, and not of concurrent Kleene algebra with hypotheses. Completeness is established in [78].

Lastly, there are probabilistic versions of CKA. In [79] the authors consider probabilistic CKA to capture nondeterminism, concurrency and probability. Completeness is not discussed, but the algebraic theory is shown to be sound for a set of probabilistic automata modulo probabilistic simulation. In [80] the authors focus on a true-concurrent model for probabilistic concurrent Kleene algebra. Soundness is established, and completeness is left open.

References

[1] T. Hoare, I. J. Hayes, J. He, C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. Sufrin, "Laws of programming," *Commun. ACM*, vol. 30, no. 8, 1987, pp. 672–686. DOI: 10.1145/27651.27653.

[2] A. Silva, "Kleene coalgebra," Ph.D. dissertation, Radboud University, 2010. [Online]. Available: https://hdl.handle.net/2066/83205.

[3] T. Kappé, "Concurrent Kleene algebra: Completeness and decidability," Ph.D. dissertation, University College London, 2020. [Online]. Available: http://discovery.ucl.ac.uk/10109361/.

[4] J. Wagemaker, "Extensions of (concurrent) Kleene algebra," Ph.D. dissertation, Radboud University, 2022. [Online]. Available: https://hdl.handle.net/2066/273924.

[5] J. J. M. M. Rutten, "Universal coalgebra: A theory of systems," *Theor. Comput. Sci.*, vol. 249, no. 1, 2000, pp. 3–80. DOI: 10.1016/S0304-3975(00)00056-6.

[6] J. J. M. M. Rutten, *The Method of Coalgebra: exercises in coinduction*. Amsterdam: CWI. [Online]. Available: https://ir.cwi.nl/pub/28550.

[7] S. Burris and H.P. Sankappanavar, *A Course in Universal Algebra*, ser. Graduate Texts in Mathematics. Springer, 1981.

[8] J. A. Brzozowski, "Derivatives of regular expressions," *J. ACM*, vol. 11, no. 4, 1964, pp. 481–494. DOI: 10.1145/321239.321249.

[9] J. J. M. M. Rutten, "Behavioural differential equations: A coinductive calculus of streams, automata, and power series," *Theor. Comput. Sci.*, vol. 308, no. 1-3, 2003, pp. 1–53. DOI: 10.1016/S0304-3975(02)00895-2.

[10] D. Kozen, "Kleene algebra with tests," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 3, 1997, pp. 427–443. DOI: 10.1145/256167.256195.

[11] D. Kozen, *Automata and Computability*, 1st. Berlin, Heidelberg: Springer-Verlag, 1997.

[12] J. H. Conway, *Regular Algebra and Finite Machines.* Chapman and Hall, Ltd., London, 1971.

[13] S. Owens, J. H. Reppy, and A. Turon, "Regular-expression derivatives re-examined," *J. Funct. Program.*, vol. 19, no. 2, 2009, pp. 173–190.

[14] K. Thompson, "Regular expression search algorithm," *Commun. ACM*, vol. 11, no. 6, 1968, pp. 419–422. DOI: 10.1145/363347.363387.

[15] V. M. Antimirov, "Partial derivatives of regular expressions and finite automaton constructions," *Theor. Comput. Sci.*, vol. 155, no. 2, 1996, pp. 291–319. DOI: 10.1016/0304-3975(95)00182-4.

[16] V. M. Glushkov, "The abstract theory of automata," *Russian Mathematical Surveys*, vol. 16, 3 1961, pp. 1–53. DOI: 10.1070/RM1961v016n05ABEH004112.

[17] G. Berry and R. Sethi, "From regular expressions to deterministic automata," *Theor. Comput. Sci.*, vol. 48, no. 3, 1986, pp. 117–126. DOI: 10.1016/0304-3975(86)90088-5.

[18] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IRE Trans. Electron. Comput.*, vol. 9, no. 1, 1960, pp. 39–47. DOI: 10.1109/TEC.1960.5221603.

[19] B. W. Watson, "A taxonomy of finite automata construction algorithms," Technische Universiteit Eindhoven, Tech. Rep., 1993. [Online]. Available: https://research.tue.nl/files/2482472/9313452.

[20] D. Kozen, "Myhill-Nerode relations on automatic systems and the completeness of Kleene algebra," in *STACS*, pp. 27–38, 2001. DOI: 10.1007/3-540-44693-1_3.

[21] B. Jacobs, “A bialgebraic review of deterministic automata, regular expressions and languages,” in *Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, pp. 375–404, 2006. DOI: 10.1007/11780274_20.

[22] D. Kozen, “A completeness theorem for Kleene algebras and the algebra of regular events,” *Inf. Comput.*, vol. 110, no. 2, 1994, pp. 366–390. DOI: 10.1006/inco.1994.1037.

[23] D. Kozen and F. Smith, “Kleene algebra with tests: Completeness and decidability,” in *CSL*, pp. 244–259, 1996. DOI: 10.1007/3-540-63172-0_43.

[24] D. Kozen, “On Hoare logic and Kleene algebra with tests,” *ACM Trans. Comput. Log.*, vol. 1, no. 1, 2000, pp. 60–76. DOI: 10.1145/343369.343378.

[25] A. Angus and D. Kozen, “Kleene algebra with tests and program schematology,” Cornell University, Tech. Rep. TR2001-1844, 2001. [Online]. Available: http://hdl.handle.net/1813/5831.

[26] D. Kozen, “Nonlocal flow of control and Kleene algebra with tests,” in *LICS*, pp. 105–117, 2008. DOI: 10.1109/LICS.2008.32.

[27] C. J. Anderson, N. Foster, A. Guha, J. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, “NetKAT: Semantic foundations for networks,” in *POPL*, pp. 113–126, 2014. DOI: 10.1145/2535838.2535862.

[28] D. M. Kaplan, “Regular expressions and the equivalence of programs,” *J. Comput. Syst. Sci.*, vol. 3, no. 4, 1969, pp. 361–386. DOI: 10.1016/S0022-0000(69)80027-9.

[29] D. Kozen, “Automata on guarded strings and applications,” *Matemática Contemporânea*, vol. 24, 2003, pp. 117–139.

[30] D. Kozen, “On the coalgebraic theory of Kleene algebra with tests,” in *Rohit Parikh on Logic, Language and Society*, ser. Outstanding Contributions to Logic, C. Başkent, L. S. Moss, and R. Ramanujam, Eds., vol. 11, Springer, 2017, pp. 279–298.

[31] E. Cohen, “Hypotheses in Kleene algebra,” Bellcore, Tech. Rep., 1994.

[32] A. Doumane, D. Kuperberg, D. Pous, and P. Pradic, “Kleene algebra with hypotheses,” in *FOSSACS*, pp. 207–223, 2019. DOI: 10.1007/978-3-030-17127-8_12.

[33] D. Kozen, “On the complexity of reasoning in Kleene algebra,” *Inf. Comput.*, vol. 179, no. 2, 2002, pp. 152–162. DOI: 10.1006/inco.2001.2960.

[34] D. Kozen and K. Mamouras, “Kleene algebra with equations,” in *ICALP*, pp. 280–292, 2014. DOI: 10.1007/978-3-662-43951-7_24.

[35] T. Kappé, P. Brunet, A. Silva, J. Wagemaker, and F. Zanasi, “Concurrent Kleene algebra with observations: From hypotheses to completeness,” in *FOSSACS*, pp. 381–400, 2020. DOI: 10.1007/978-3-030-45231-5_20.

[36] D. Pous, J. Rot, and J. Wagemaker, “On tools for completeness of Kleene algebra with hypotheses,” in *RAMICS*, pp. 378–395, 2021. DOI: 10.1007/978-3-030-88701-8_23.

[37] D. Pous, J. Rot, and J. Wagemaker, “On tools for completeness of Kleene algebra with hypotheses,” *Log. Methods Comput. Sci.*, vol. 20, no. 2, 2024. DOI: 10.46298/LMCS-20(2:8)2024.

[38] S. C. Kleene, “Representation of events in nerve nets and finite automata,” in *Automata Studies*, C. E. Shannon and J. McCarthy, Eds., Princeton University Press, 1956, pp. 3–41.

[39] A. Salomaa, “Two complete axiom systems for the algebra of regular events,” *J. ACM*, vol. 13, no. 1, 1966, pp. 158–169. DOI: 10.1145/321312.321326.

[40] D. Krob, “A complete system of b-rational identities,” in *ICALP*, pp. 60–73, 1990. DOI: 10.1007/BFb0032022.

[41] M. Boffa, “Une remarque sur les systèmes complets d’identités rationnelles,” *ITA*, vol. 24, 1990, pp. 419–428.

[42] D. Kozen, “Kleene algebra with tests and commutativity conditions,” in *TACAS*, pp. 14–33, 1996. DOI: 10.1007/3-540-61042-1_35.

[43] F. Bonchi and D. Pous, “Checking NFA equivalence with bisimulations up to congruence,” in *POPL*, pp. 457–468, 2013. DOI: 10.1145/2429069.2429124.

[44] T. Braibant and D. Pous, “An efficient Coq tactic for deciding Kleene algebras,” in *ITP*, pp. 163–178, 2010. DOI: 10.1007/978-3-642-14052-5_13.

[45] D. Kozen and F. Smith, “Kleene algebra with tests: Completeness and decidability,” in *CSL*, pp. 244–259, 1996. DOI: 10.1007/3-540-63172-0_43.

[46] D. Pous, “Kleene algebra with tests and Coq tools for while programs,” in *ITP*, pp. 180–196, 2013. DOI: 10.1007/978-3-642-39634-2_15.

[47] T. Kappé, P. Brunet, J. Rot, A. Silva, J. Wagemaker, and F. Zanasi, “Kleene algebra with observations,” in *CONCUR*, 41:1–41:16, 2019. DOI: 10.4230/LIPIcs.CONCUR.2019.41.

[48] N. B. B. Grathwohl, D. Kozen, and K. Mamouras, “KAT + B!,” 44:1–44:10, ACM, 2014. DOI: 10.1145/2603088.2603095.

[49] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson, “A coalgebraic decision procedure for NetKAT,” in *POPL*, pp. 343–355, 2015. DOI: 10.1145/2676726.2677011.

[50] S. Smolka, N. Foster, J. Hsu, T. Kappé, D. Kozen, and A. Silva, “Guarded Kleene algebra with tests: Verification of uninterpreted programs in nearly linear time,” 2020, 61:1–61:28. DOI: 10.1145/3371129.

[51] J. Desharnais, B. Möller, and G. Struth, “Kleene algebra with domain,” *ACM Trans. Comput. Log.*, vol. 7, no. 4, 2006, pp. 798–833. DOI: 10.1145/1183278.1183285.

[52] J. Desharnais, B. Möller, and G. Struth, “Modal Kleene algebra and applications - a survey,” *Journal on Relational Methods in Computer Science*, vol. 1, 2004, pp. 93–131.

[53] B. Möller and G. Struth, “Modal Kleene algebra and partial correctness,” in *AMAST*, pp. 379–393, 2004. DOI: 10.1007/978-3-540-27815-3_30.

[54] D. Pous and J. Wagemaker, “Completeness theorems for Kleene algebra with top,” in *CONCUR*, 26:1–26:18, 2022. DOI: 10.4230/LIPICS.CONCUR.2022.26.

[55] D. Pous and J. Wagemaker, “Completeness theorems for Kleene algebra with tests and top,” *Log. Methods Comput. Sci.*, vol. 20, no. 3, 2024. DOI: 10.46298/LMCS-20(3:27)2024.

[56] C. Zhang, A. A. de Amorim, and M. Gaboardi, “On incorrectness logic and Kleene algebra with top and tests,” in *POPL*, pp. 1–30, 2022. DOI: 10.1145/3498690.

[57] K. Mamouras, "Equational theories of abnormal termination based on Kleene algebra," in *FOSSACS*, pp. 88–105, 2017. DOI: 10.1007/978-3-662-54458-7_6.

[58] P. Brunet, "A note on commutative Kleene algebra," 2019. arXiv: 1910.14381.

[59] D. Kozen, "Kleene algebra with tests and commutativity conditions," in *TACAS*, pp. 14–33, 1996. DOI: 10.1007/3-540-61042-1_35.

[60] R. Milner, "A complete inference system for a class of regular behaviours," *J. Comput. Syst. Sci.*, vol. 28, no. 3, 1984, pp. 439–466. DOI: 10.1016/0022-0000(84)90023-0.

[61] C. A. Grabmayer, "Milner's proof system for regular expressions modulo bisimilarity is complete: Crystallization: Near-collapsing process graph interpretations of regular expressions," in *LICS*, 34:1–34:13, 2022. DOI: 10.1145/3531130.3532430.

[62] T. Schmid, T. Kappé, and A. Silva, "A complete inference system for skip-free guarded Kleene algebra with tests," in *ESOP*, pp. 309–336, 2023. DOI: 10.1007/978-3-031-30044-8_12.

[63] A. McIver and T. Weber, "Towards automated proof support for probabilistic distributed systems," in *LPAR*, pp. 534–548, 2005. DOI: 10.1007/11591191_37.

[64] A. K. McIver, C. Gonzalía, E. Cohen, and C. C. Morgan, "Using probabilistic Kleene algebra pKA for protocol verification," *J. Log. Algebraic Methods Program.*, vol. 76, no. 1, 2008, pp. 90–111. DOI: 10.1016/J.JLAP.2007.10.005.

[65] A. McIver, T. M. Rabehaja, and G. Struth, "On probabilistic Kleene algebras, automata and simulations," in *RAMICS*, pp. 264–279, 2011. DOI: 10.1007/978-3-642-21070-9_20.

[66] W. Rozowski, T. Kappé, D. Kozen, T. Schmid, and A. Silva, "Probabilistic guarded KAT modulo bisimilarity: Completeness and complexity," in *ICALP*, 136:1–136:20, 2023. DOI: 10.4230/LIPICS.ICALP.2023.136.

[67] C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman, "Foundations of concurrent Kleene algebra," in *RelMICS/AKA*, pp. 166–186, 2009. DOI: 10.1007/978-3-642-04639-1_12.

[68] T. Hoare, B. Möller, G. Struth, and I. Wehrman, "Concurrent Kleene algebra and its foundations," *J. Log. Algebraic Methods Program.*, vol. 80, no. 6, 2011, pp. 266–296. DOI: 10.1016/J.JLAP.2011.04.005.

[69] G. Struth, "Trimming the hedges: An algebra to tame concurrency," in *Theories of Programming: The Life and Works of Tony Hoare*, ser. ACM Books, vol. 39, ACM / Morgan & Claypool, 2021, pp. 317–346. DOI: 10.1145/3477355.3477370.

[70] M. R. Laurence and G. Struth, "Completeness theorems for pomset languages and concurrent Kleene algebras," 2017. arXiv: 1705.05896.

[71] T. Kappé, P. Brunet, A. Silva, and F. Zanasi, "Concurrent Kleene algebra: Free model and completeness," in *ESOP*, pp. 856–882, 2018. DOI: 10.1007/978-3-319-89884-1_30.

[72] M. R. Laurence and G. Struth, "Completeness theorems for bi-Kleene algebras and series-parallel rational pomset languages," in *RAMICS*, pp. 65–82, 2014. DOI: 10.1007/978-3-319-06251-8_5.

[73] T. Kappé, P. Brunet, A. Silva, J. Wagemaker, and F. Zanasi, "Concurrent Kleene algebra with observations: From hypotheses to completeness," in *FOSSACS*, pp. 381–400, 2020. DOI: 10.1007/978-3-030-45231-5_20.

[74] J. Wagemaker, P. Brunet, S. Docherty, T. Kappé, J. Rot, and A. Silva, "Partially observable concurrent Kleene algebra," in *CONCUR*, 20:1–20:22, 2020. DOI: 10.4230/LIPICS.CONCUR.2020.20.

[75] J. Wagemaker, N. Foster, T. Kappé, D. Kozen, J. Rot, and A. Silva, "Concurrent NetKAT - modeling and analyzing stateful, concurrent networks," in *ESOP*, pp. 575–602, 2022. DOI: 10.1007/978-3-030-99336-8_21.

[76] G. Caltais, H. Hojjat, M. R. Mousavi, and H. C. Tunç, "DyNetKAT: An algebra of dynamic networks," in *FOSSACS*, pp. 184–204, 2022. DOI: 10.1007/978-3-030-99253-8_10.

[77] C. Prisacariu, "Synchronous Kleene algebra," *J. Log. Algebraic Methods Program.*, vol. 79, no. 7, 2010, pp. 608–635. DOI: 10.1016/J.JLAP.2010.07.009.

[78] J. Wagemaker, M. M. Bonsangue, T. Kappé, J. Rot, and A. Silva, "Completeness and incompleteness of synchronous Kleene algebra," in *MPC*, pp. 385–413, 2019. DOI: 10.1007/978-3-030-33636-3_14.

[79] A. McIver, T. M. Rabehaja, and G. Struth, "Probabilistic concurrent Kleene algebra," in *QAPL*, pp. 97–115, 2013. DOI: 10.4204/EPTCS.117.7.

[80] A. McIver, T. M. Rabehaja, and G. Struth, "An event structure model for probabilistic concurrent Kleene algebra," in *LPAR*, pp. 653–667, 2013. DOI: 10.1007/978-3-642-45221-5_43.