Automata Closure Constructions for Kleene Algebra with Hypotheses

MSc Thesis *(Afstudeerscriptie)*

written by

Liam Chung
(born April 3, 1999 in New York, United States of America)

under the supervision of **Dr. Tobias Kappé** and **Prof. Dr. Yde Venema**, and submitted to the Examinations Board in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:
August 27, 2024

Members of the Thesis Committee:
Dr. Katia Shutova (chair)
Dr. Tobias Kappé
Prof. Dr. Yde Venema
Dr. Malvin Gattinger
Dr. Jana Wagemaker

Institute for Logic, Language and Computation

Abstract

In this thesis, we will develop automaton constructions as a tool for studying the equational theories of Kleene algebra with arbitrary hypotheses. Kleene algebra with hypotheses is a framework for assigning language-based semantics to Kleene algebra-based systems where extra features are formulated as axioms in the equational logic.

These semantics are defined by a closure operation on languages called "hypothesis closure". Completeness of the "hypothesis-closed" semantics can be shown via reductions to known systems, such as Kleene algebra with tests. However, doing so requires that the hypothesis closure can be effectively computed. This thesis proposes a general approach to calculating the hypothesis closure in terms of standard Kleene algebra, for arbitrary hypotheses.

We define two constructions on automata and prove they are correct, when they terminate. We examine cases where they terminate, and present a strategy for improving how often they do. In cases where the constructions terminate, proving completeness becomes much simpler via reductions, and we even obtain a decision procedure for equality, using automaton equivalence. We will therefore work extensively with automata-theoretic machinery to define the proposed constructions, and prove their correctness.

Acknowledgements

Naturally I want thank my supervisors, Tobias Kappé and Yde Venema. Thank you Tobias for all of your guidance and patience throughout this process. Your enthusiasm for every new topic we discuss is always infectious. Thank you Yde for your feedback and support, always having a ready supply of notes and ideas. It was an honour to work with both of you. Thanks also to Malvin Gattinger and Jana Wagemaker for being on my thesis committee.

Now, some thanks to my family and friends. To Mom, Dad, and Avery, for pushing me to take care of myself, a task arguably even more difficult than writing this thesis; to Lingyuan, for always being around to enable my categorical nonsense; to Brendan, for the long walks and long talks.

Last but not least: to 一一. Thank you, as always, for holding me together.

Contents

Chapter 1

Introduction

Kleene algebra, originally known as the "algebra of regular events" [7] is an algebraic structure used to study equivalences between so-called *regular expressions*. Regular expressions, not to be confused with those known to programmers, are syntactic terms that can abstractly represent computer programs. A fundamental result of Kleene algebra is that its equational theory is both *decidable* and *complete*, opening the door to automated program equivalence and formal verification in a restricted class of cases.

In practice, however, this computational pleasantness comes at a cost: Kleene algebra lacks a great deal of the expressibility. Recent years have seen a proliferation of work studying Kleene algebra augmented to model more complex program features. Some examples include booleans in Kleene algebra with tests [9], parallelism in concurrent Kleene algebra [3], and even a fully-featured network programming language in NetKAT [1]. Each of these takes the form of Kleene algebra altered with new rules or assumptions, so as to effectively represent some computational concept.

When working with any proof system, completeness is a central concern: that is, can we prove everything that is true, and so be confident that if I can not prove something, it is not true? For each of the systems mentioned above, work introducing it defines an appropriate class of models for which it is complete, and proves that this is the case. The choice of semantics and corresponding completeness proof is often specific to the system in question, but these proofs follow patterns, raising the question of if the situation can be generalised to all systems based on Kleene algebra.

Recent work by Doumane et al. [2] developed the *hypothesis closure*, a closure operation on languages that associates canonical, language-based models with Kleene algebras extended with extra axioms. By language-based, we mean that the semantics of the system are simply a certain sub-class of languages, for example "all languages that contain the word `abc`", "all languages that are closed under replacing `b` with `c` in every word", etc.

There is major boon imparted by working purely with language-based semantics across various extensions of Kleene algebra: results in one system can

be easily interpreted in another, and when the class of languages for a system is a sub-class of regular languages, it is grounded in the *complete* and *decidable* theory of Kleene algebra for regular languages. These nice properties of Kleene algebra create a very favorable environment for studying these systems.

The semantic framework developed by Doumane et al. is also useful because it is defined purely in terms of the new axioms. In practice, this provides an alternative semantics for variants of Kleene algebra like KAT, so long as the additions to the system can be formulated as extra axioms in the equational theory. This is far easier than designing bespoke semantics for which the system will be complete; Pous et al. [12] have developed a suite of tools for accomplishing this task, and even show that the standard semantics associated with KAT are indeed equivalent to those generated by the hypothesis closure.

Unfortunately, equational theories of Kleene algebra extended with hypotheses are undecidable in general, even for quite simple hypotheses like commutativity $ab = ba$ [9]. However, if one can effectively interpret the semantics of a Kleene algebra with hypotheses in terms of "standard" Kleene algebra, then decidability of Kleene algebra can be leveraged to obtain decidability and take a meaningful step towards completeness for the system.

The objective of this thesis will be to define, and prove the correctness of, automaton constructions that can be used to calculate the hypothesis closure of an arbitrary regular expression, for an arbitrary hypothesis. This construction will not always terminate in finite time, and so its termination will be an important topic as well.

The work of this thesis can be summed up as follows:

1. Specifications of two constructions on automata, $H^*_\bullet$ and H^*;
2. Proofs that when the constructions terminate, they produce the correct automaton;
3. Strategies for finding hypotheses that produce equivalent closures, so that when the construction does not terminate, we seek equivalent hypotheses for which it does.

In chapter 2, we will review important notions for the thesis, in particular those for regular languages and some order theory. In chapter 3, we will motivate and formally build the notion of hypothesis and hypothesis closure, compiling recent work on the subject, and setting our objective for the thesis more formally. In chapter 4, we put forward the first version of our construction: what we call the singleton version. It is defined for a simpler sub-class of hypotheses, and will serve to acquaint the reader with the techniques that will be used in the more general version, which will be covered in chapter 6. In chapter 5, we discuss a technique for improving the termination conditions of the singleton construction, and prove its effectiveness. In chapter 6, we will generalise the construction given in chapter 4, to work for arbitrary hypotheses; and we will again prove its correctness. Finally, in chapter 7, we will conclude with a review of the major results, and remarks on future work.

Chapter 2

Preliminaries

We assume that the reader has a passing familiarity with regular expressions, finite automata, and regular languages. The main purpose of this section will not be to educate on these topics, but rather serve as a refresher, and fix the notations and terminology that will be in use for this thesis. For a more detailed treatment, [4] is highly recommended.

2.1 Languages, Regular Expressions, Automata

Definition 2.1.
An *alphabet* Σ is a finite set of *letters*, for which we will use the teletype letters $\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \ldots$

- A *word* w over the alphlabet Σ is a finite ordered sequence of elements from Σ, for example $\mathtt{aaa}, \mathtt{cab}, \mathtt{dad}$, and so on. The *empty word*, denoted ε, is the empty sequence of letters.
- The *set of all words* over the alphabet Σ is written Σ^*.
- A *language* L over the alphabet Σ is a (potentially infinite) set of words over Σ, or more formally, $L \in \mathcal{P}(\Sigma^*)$.

Definition 2.2.
A *regular expression* over the alphabet Σ is any term obtained in one of the following ways:

- for all $\mathtt{a} \in \Sigma$, $\mathtt{a}$ is a regular expression;
- 0 and 1 are regular expressions;
- if e and f are regular expressions, so is $e \cdot f$;
- if e and f are regular expressions, so is $e + f$;
- if e is a regular expression, so is e^*.

We will refer to $\cdot$ as *composition*, $+$ as *choice*, and $*$ as *Kleene star*. When writing expressions, we often omit $\cdot$, so that $\mathtt{abc} = \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c}$, and the binding order is $* > \cdot > +$, so for example:

$$\mathtt{a}^*\mathtt{bc} + \mathtt{de}^*\mathtt{f} = \big((\mathtt{a}^*) \cdot \mathtt{b} \cdot \mathtt{c}\big) + \big(\mathtt{d} \cdot (\mathtt{e}^*) \cdot \mathtt{f}\big)$$

Lastly, we will refer to the set of all regular expressions over an alphabet Σ as $\mathcal{T}(\Sigma)$.

We now define a semantics for regular expression in terms of languages, where a regular expression r will represent some language $[\![r]\!] \in \mathcal{P}(\Sigma^*)$.

Definition 2.3.
The language-based semantics for regular expressions $[\![-]\!] : \mathcal{T}(\Sigma) \to \mathcal{P}(\Sigma^*)$, is defined:

$$\begin{aligned}
&[\![0]\!] := \emptyset && [\![e \cdot f]\!] := \{uv : u \in [\![e]\!], v \in [\![f]\!]\} \\
&[\![1]\!] := \{\varepsilon\} && [\![e + f]\!] := [\![e]\!] \cup [\![f]\!] \\
&[\![\mathtt{a}]\!] := \{\mathtt{a}\} \quad \forall \mathtt{a} \in \Sigma && [\![e^*]\!] := \{\varepsilon\} \cup \bigcup_{n \in \mathbb{N}} [\![e^n]\!]
\end{aligned}$$

where e^n denotes $e \cdot e \cdot \ldots \cdot e$, n many times. We define $\cdot$ for languages as exactly the operation corresponding to $\cdot$ on regular expressions above, as this will sometimes be convenient notationally.

If for a language $L \in \mathcal{P}(\Sigma^*)$ there is some regular expression r such that $[\![r]\!] = L$, we call L a *regular language*. In such a case we call r a regular expression *representing* L.

Also note that in the context of *expressions*, we use 0 and 1, where in the context of *languages*, we use $\emptyset$ and ε. With this semantics in hand, we make two observations that will be central to our understanding of regular expressions:

1. given a regular language, there can be more than one regular expression representing it;
2. there exist languages with no regular expression representing them. In other words, not all languages are regular.

Example 2.4.
Using our newly defined semantics, we can easily find two regular expressions whose semantics are the same:

$$[\![\mathtt{a} + \mathtt{b}]\!] = \{\mathtt{a}, \mathtt{b}\} = [\![\mathtt{b} + \mathtt{a}]\!].$$

On the other hand, the following language is *non-regular*: there is no regular language representing it.

$$\{\mathtt{a}^n\mathtt{b}\mathtt{a}^n : n \in \mathbb{N}\} = \{\mathtt{aba}, \mathtt{aabaa}, \ldots\}$$

it is well known that this language is non-regular, but we refer to [4], section 4.1 for details.

We also refer to [4] for some important properties of regular languages, namely their closure properties and decidability.

Lemma 2.5.
Let L, M be regular languages. The following are also regular languages:

- $L \cup M$
- $L \cap M$
- $L \setminus M$

Let L_i for $i \in I$ be a finite set of regular languages. The following are also regular languages:

- $\bigcup_{i \in I} L_i$
- $\bigcap_{i \in I} L_i$

Theorem 2.6.
Let r, s be regular expressions, $w \in \Sigma^*$ a word. The following problems are all decidable:

1. is $w \in [\![r]\!]$?
2. is $[\![r]\!] = [\![s]\!]$?
3. is $[\![r]\!]$ non-empty?

Now we define automata, fixing our particular notation for it. This thesis will mostly deal with finite, non-deterministic automata, so we will define this as our "default" form for an automaton.

Definition 2.7.
An *automaton* $\mathcal{X}$ on an alphabet Σ is a tuple $(X, \delta_{\mathcal{X}}, \nu_{\mathcal{X}})$ consisting of a finite set X, called the *state set*, a map $\delta_{\mathcal{X}} : X \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(X)$, called the *transition function*, and a subset $\nu_{\mathcal{X}} \subseteq X$, called the subset of *accepting states*. We will often exclude the subscripts when the automaton we are dealing with is clear. We call the class of all automata NA.

We also define the *extended transition function* $\delta^*_{\mathcal{X}} : X \times \Sigma^* \to \mathcal{P}(X)$. Here we let $E_{\mathcal{X}}(x)$ for a state x refer to the "epsilon closure" of x, that is, the set of all states reachable from x in $\mathcal{X}$ via only ε transitions. We now define δ^* recursively:

$$\delta^*(x, \varepsilon) = E_{\mathcal{X}}(x)$$
$$\delta^*(x, \mathtt{a} \cdot w) = \bigcup_{y \in \delta^*(x,w)} E_{\mathcal{X}}(\delta(y, \mathtt{a}))$$

namely, the extended transition function simply applies the normal transition function one letter at a time, starting with the state x. We will say that x *accepts* a word w if there exists some state $z \in \delta^*(x, w)$ such that $z \in \nu$.

We will often overload the transition function notation δ to refer to either the normal letter version, or the extended word version.

Automata are a more concrete tool for reasoning about simple programs and computational processes. Their simplicity (compared to a model like the Turing Machine) does impose restrictions on its expressivity: for example the notions of memory and state are intertwined, since possible states for the program must be finite.

A state in a given automaton indeed has a *set* of *words* that it accepts, and so naturally we can fit this into our existing framwork of languages:

Definition 2.8.
Let $\mathcal{X}$ be an automaton. For a state $x \in X$, *the language accepted* by x is defined:

$$l_{\mathcal{X}}(x) := \{w \in \Sigma^* : \exists z \in \delta_{\mathcal{X}}(w, x)\ z \in v_{\mathcal{X}}\}$$

When using an automaton to discuss a language, we will often want to highlight a particular state. A *pointed* automaton is simply a pair $(\mathcal{X}, x_0)$ where $x_0 \in X_{\mathcal{X}}$ is called the *initial state*. We may refer to $(\mathcal{X}, x_0)$ as just $\mathcal{X}$ when it is established what the highlighted state is.

Unlike a regular expression, which (under the standard semantics given) represents exactly one language, an automaton yields a *set* of related languages, one for each state. On the other hand, much like regular expressions, a state in an automaton does not uniquely represent its language, and there are usually many automata one can define to accept a given language.

We will be proving some fairly involved results about constructions on automata, in particular the languages that are accepted by them. We also define the notion of *trace* so that we can talk about valid strings of inputs on an automaton.

Definition 2.9.
Given an automaton $\mathcal{X}$, an $\mathcal{X}$ *trace* is a finite sequence of states and transitions in $\mathcal{X}$:

$$x_1 \overset{l_1}{\to} x_2 \overset{l_2}{\to} \cdots \overset{l_{n-1}}{\to} x_n$$

where each $x_i \in X$ and $l_i \in \Sigma$, such that for all $i < n$, $x_{i+1} \in \delta(l_i, x_i)$. We will call a trace *accepting* if $x_n \in \nu$.

When describing traces, we will often use this diagrammatic notation. In particular we will use the $\bullet$ to represent an arbitrary state in the automaton, and $\circledcirc$ to represent an arbitrary accepting state in the automaton.

Deterministic automata will not play as significant of a role in this thesis as non-deterministic automata, but we will use them for some smaller proofs.

Definition 2.10.
An automaton is *deterministic* if for every $\mathtt{a} \in \Sigma$ and $x \in X$, $\delta_{\mathcal{X}}(x, \mathtt{a}) = \{y\}$ for some $y \in X$, and $\delta_{\mathcal{X}}(x, \varepsilon) = \emptyset$. In other words, the transition function can be rewritten as a function $X \times \Sigma \to X$.

It is also worth introducing the power set construction; for the proof of its correctness as well as a more in-depth treatment, see [4], section 2.3.5.

Theorem 2.11.
Let $\mathcal{X} = (X, \delta, \nu)$ be an automaton. Then for any $w \in \Sigma^*, S \subseteq \Sigma^*$, let:

$$X' := \mathcal{P}(X) \quad \delta'(S, w) = \bigcup_{x \in S} \delta(x, w) \quad \nu' = \{S \in \mathcal{P}(X) : S \cap \nu \neq \emptyset\}$$

and we let $\mathcal{X}' := (X', \delta', \nu')$. We call this automaton the output of the *power set construction* applied to $\mathcal{X}$. Now:

$$\forall x \in X,\ l_{\mathcal{X}}(x) = l_{\mathcal{X}'}(\{x\}).$$

2.2 Regular Languages

Theorem 2.12 (Kleene's Theorem)**.**
The set of languages represented by regular expressions and finite automata correspond; that is, for every regular expression r there is a pointed automaton $(\mathcal{X}, x)$ such that $l_{\mathcal{X}}(x) = [\![r]\!]$. Dually, for every state y in an automaton $\mathcal{Y}$, there is a regular exprssion s such that $[\![s]\!] = l_{\mathcal{Y}}(y)$.

This theorem can be witnessed with constructions in either direction. In particular, the construction taking regular expressions to pointed automata is of interest to us, which is called *Thompson's construction*. We do not describe it in detail here, but a curious reader can see [4] section 3.2.3.

Lemma 2.13.
For a regular expression r, Thompson's construction outputs a pointed automaton $(\mathcal{Z}, z_0)$ such that:

$$l_{\mathcal{Z}}(z_0) = [\![r]\!]$$

as per Kleene's theorem. We will sometimes use $\mathcal{Z}_r$ to refer to $\mathcal{Z}$, to make clear what regular expression it represents.

In later sections of the thesis, we will use pseudocode to describe constructions on automata and regular expressions. In these contexts, we will use the function name RegToAut to refer to Thompson's construction.

The correspondence between automata and regular expressions also has a very important use: showing that it is decidable whether a regular expression's language is empty.

Theorem 2.14.
Let r be a regular expression. Then it is decidable whether $[\![r]\!]$ is empty.

Now we introduce an important tool for working with regular languages: the Brzozowski derivative.

Definition 2.15 (Brzozowski Derivative)**.**
Let L be a regular language, and $w \in \Sigma^*$ a word. The *Brzozowski derivative* of L with respect to w is defined:

$$w^{-1}L := \{u \in \Sigma^* : wu \in L\}.$$

If r is a regular expression, we can define a Brzozowski derivative for r as well in the obvious way:

$$w^{-1}r := w^{-1}[\![r]\!].$$

Note that the Brzozowski derivative of a regular expression is a language.

Example 2.16.
An easy way to understand the Brzozowski derivative can be to just look at some examples:

- $(\mathtt{a})^{-1}\{\mathtt{a}, \mathtt{ab}, \mathtt{aaba}\} = \{\varepsilon, \mathtt{b}, \mathtt{aba}\}$
- $(\mathtt{a})^{-1}\{\mathtt{b}, \mathtt{ba}, \mathtt{aba}\} = \{\mathtt{ba}\}$
- $(\mathtt{ab})^{-1}\{\mathtt{bac}, \mathtt{abb}, \mathtt{aba}\} = \{\mathtt{b}, \mathtt{a}\}$
- $(\mathtt{a})^{-1}((\mathtt{a} + \mathtt{b})\mathtt{c}) = \{\mathtt{c}\}$

A useful intuition for the Brzozowski derivative is to think of an automaton with a state accepting L: "apply" w to this state (by which we mean $\delta(-, w)$), and the Brzozowski derivative is the union of all languages accepted by the resulting states.

Proposition 2.17.
Let $\mathcal{X}$ be an automaton, and $x \in X$ a state in it. Let $w \in \Sigma^*$ a word. Then:

$$w^{-1}l_{\mathcal{X}}(x) = \bigcup_{y \in \delta(x,w)} l_{\mathcal{X}}(y).$$

Proof. Let $u \in w^{-1}l_{\mathcal{X}}(x)$. Then by definition, $wu \in l_{\mathcal{X}}(x)$. So there must exist an $\mathcal{X}$ trace for the word wu that starts at x, and ends at an accepting state z. By definition there must exist a state y such that $y \in \delta(x, w)$ and $z \in \delta(y, u)$. However z is an accepting state, and so $u \in l_{\mathcal{X}}(y)$ where $y \in \delta(x, w)$ as desired.

Now let $v \in \bigcup_{y \in \delta(x,w)} l_{\mathcal{X}}(y)$. So there exists a state y such that $y \in \delta(x, w)$ and $v \in l_{\mathcal{X}}(y)$. This latter fact means that there must exist an $\mathcal{X}$ trace beginning at y, ending at an accepting state z, for the word v. Now because $y \in \delta(x, w)$, there must also exist an $\mathcal{X}$ trace starting at x and ending at y for the word w. Concatenating these two traces, we obtain an $\mathcal{X}$ trace starting at x and ending at z for the word wv, and because z is an accepting state, we can conclude that $wv \in l_{\mathcal{X}}(x)$. So by defition, it must be that $v \in w^{-1}l_{\mathcal{X}}(x)$, as desired. 完

Now, we state some more important facts about Brzozowski derivatives. We defer to [4] for proofs of these facts.

Lemma 2.18.
Let $L = [\![r]\!]$ be a regular language, $w, u \in \Sigma^*$ words, $\mathcal{X}$ an automaton, and $x \in X$ a state in it. Then:

1. $w^{-1}L$ is a regular language
2. it is decidable whether $w^{-1}e$ is empty
3. $(wu)^{-1}L = u^{-1}(w^{-1}L)$
4. if $\varepsilon \in w^{-1}l_{\mathcal{X}}(x)$ then $w \in l_{\mathcal{X}}(x)$

2.3 Equational Theory of Kleene Algebra

We have now established a theory of regular languages, characterised by both regular expressions and automata. Recall that both regular expressions and automata do not generally refer to a regular language uniquely: there are usually many equivalent ways to represent a language. As such, we will define a formal system, called *Kleene algebra*, that can reason about these equivalences. This system is *equational* in that it studies what expressions are equal to what others. A more comprehensive treatment can be found in [13], chapter II, section 14.

Definition 2.19.
An *equation* of regular expressions is $e = f$, where e, f are regular expressions. An *inequation* is $f \geq e$ (also written $e \leq f$), and is simply shorthand for an equation:

$$f \geq e \overset{(\text{def})}{\Leftrightarrow} f = e + f.$$

A *quasi-equation* is an implication $A \to B$ where A, B are equations (or inequations).

Let Ax be a set of equations and quasi-equations, which we will also call the set of *axioms*. We will say that Ax proves an equality $e = f$, written $\mathsf{Ax} \vdash e = f$, if there is a finite sequence of equations:

$$p_1 = q_1, p_2 = q_2, \dots \ , e = f$$

where each $p_i = q_i$, $e = f$ is an axiom, or follows from prior statements and axioms by the standard inference rules of equational logic. We call such a sequence of statement a *proof* in equational logic.

Given a set of axioms Ax, we may choose to adopt a set of extra axioms H (for *hypotheses*) with which to reason. In such a case, we may use the following notation:

$$\mathsf{Ax} + H \vdash e = f \quad \iff (\mathsf{Ax} \cup H) \vdash e = f.$$

We will define KA as a finite set of axioms that can be used to reason about regular expressions, and a central topic of this thesis will be the study of KA as an axiom set when augmented with some set of extra hypotheses H.

We also need to say what it means for a choice of axioms to be *"correct"* for our interpretation of regular expressions. This will mainly be that they are both *sound* and *complete* for our semantics $[\![-]\!]$.

Definition 2.20.
Let Ax be a set of axioms, and let $[\![-]\!]_A : \mathcal{T}(\Sigma) \to \mathcal{P}(\Sigma^*)$ be a language-based semantics for regular expressions.

- We say that Ax is *sound* for $[\![-]\!]_A$ if for all regular expressions e, f:

$$\mathsf{Ax} \vdash e = f \Rightarrow [\![e]\!]_A = [\![f]\!]_A.$$

- We say that Ax is *complete* for $[\![-]\!]$ if for all regular expressions e, f:

$$\mathsf{Ax} \vdash e = f \Leftarrow [\![e]\!]_A = [\![f]\!]_A.$$

Defining a finite set of sound and complete axioms for $[\![-]\!]$ of Kleene algebra was an open problem for a number of years, but a set was finally put forward by Kozen in 1991 [8]. The axioms as they appear here come from [9].

Definition 2.21.
The axiom set KA consists of:

1. $(e + f) + g = e + (f + g)$
2. $e + f = f + e$
3. $e + e = e$
4. $e + 0 = e = 0 + e$
5. $e(fg) = (ef)g$
6. $1e = e = e1$
7. $0e = 0 = e0$
8. $e(f + g) = ef + eg$
9. $(e + f)g = eg + fg$
10. $1 + ee^* = e^*$
11. $1 + e^*e = e^*$
12. $f + eg \leq g \to e^*f \leq g$
13. $f + ge \leq g \to fe^* \leq g$

where $e, f, g \in \mathcal{T}(\Sigma)$.

Soundness is straightforward to verify, and Kozen proved that these axioms are complete in the same paper [8].

Theorem 2.22.
KA is both sound and complete on the set of regular languages over a finite alphabet Σ: for all $e, f \in \mathcal{T}(\Sigma)$, $[\![e]\!] = [\![f]\!]$ if and only if $\mathsf{KA} \vdash e = f$.

Let us look at some examples of what proofs using these axioms look like.

Example 2.23.
For all $e, f \in \mathcal{T}(\Sigma)$:

1. $\mathsf{KA} \vdash e^*e = ee^*$
2. $\mathsf{KA} + \{e \leq f, f \leq e\} \vdash e = f$.
3. $\mathsf{KA} \vdash e \leq e + f$

Proof. In practice, we will not write proofs at this level of detail, as more complex proofs can become lengthy very quickly. Nonetheless, we do these in their full formality.

1. Each of the following equalities are exactly axioms of KA:

$$ee^* \overset{(11)}{=} e(1 + e^*e) \overset{(8)}{=} e + ee^*e \overset{(9)}{=} (1 + ee^*)e \overset{(10)}{=} e^*e$$

2. Suppose that $e \leq f$ and $f \leq e$. Recall that these are equivalent to $f = e+f$ and $e = f + e$ respectively. Therefore:

$$e = f + e \overset{(2)}{=} e + f = f$$

where the middle equality is an axiom of KA, and the other two equalities are exactly our assumptions.

3. In order to prove that $e \leq e+f$, we will prove the equivalent (by definition) statement that $e + f = (e + f) + e$:

$$e + f \overset{(3)}{=} (e + e) + f \overset{(1)}{=} e + (e + f) \overset{(2,1)}{=} (e + f) + e.$$

完

2.4 Order Theory and Lattices

This thesis will deal heavily with operations that close or augment existing structures. For example, constructions on automata that add new states, transitions, and accepting states. In order to make proofs about these operations easier, we discuss some order theory here.

Definition 2.24.
A *partial order* is a relation $\leq$ on a set X that is reflexive, antisymmetric, and transitive. In this case, the pair $(X, \leq)$ is called a *partially ordered set*, or *poset*.

As an example of a poset, and one that we will use later in this thesis, we introduce a poset on automata.

Definition 2.25.
Let "$\sqsubseteq$" be the order on NA such that, for two automata $\mathcal{X} = (X, \delta, \nu)$ and $\mathcal{Y} = (Y, \delta', \nu')$:

$$\mathcal{X} \sqsubseteq \mathcal{Y} \quad \text{if and only if} \quad X \subseteq Y \text{ and } \forall x \in X,\ l_{\mathcal{X}}(x) \subseteq l_{\mathcal{Y}}(x).$$

It is straightforward to see that $\sqsubseteq$ is a partial order on automata, since $\subseteq$ is a partial order on Σ^*. It will be useful to also prove a sufficient (but not necessary) condition that is more "physical" with respect to automata:

Lemma 2.26.
Let $\mathcal{X}$, $\mathcal{Y}$ be automata. If the following all hold,

- $X_{\mathcal{X}} \subseteq X_{\mathcal{Y}}$
- $\forall x \in X_{\mathcal{X}}, \forall w \in \Sigma^*,\ \delta_{\mathcal{X}}(x, w) \subseteq \delta_{\mathcal{Y}}(x, w)$
- $\nu_{\mathcal{X}} \subseteq \nu_{\mathcal{Y}}$

then $\mathcal{X} \sqsubseteq \mathcal{Y}$.

Throughout this thesis we will be working extensively with *functions* on posets, and proving facts about them. To this end, we define two important properties:

Definition 2.27.
Let $(X, \leq)$ and $(Y, \leq')$ be posets, and let $f : X \to Y$. We say that f is *monotone* with respect to $\leq, \leq'$ if for all $x_1, x_2 \in X$,

$$x_1 \leq x_2 \Rightarrow f(x_1) \leq' f(x_2).$$

Now let $g : X \to X$. We say that g is *inflationary* with respect to $\leq$ if for all $x \in X$:

$$x \leq g(x).$$

A particular kind of order that we will also use in this thesis is a *lattice*, which will be particularly important as we study languages.

Definition 2.28.
For some subset $S \subseteq X$, a *lower bound* (*upper bound*) on S is some element $z \in X$ such that $z \leq s$ ($z \geq s$) for all $s \in S$. We call z the *greatest lower bound*, or *meet* (*least upper bound*, or *join*) of S if for all lower bounds (upper bounds) $a \in X$, $z \geq a$ ($z \leq a$).

A poset $(X, \leq)$ is called a *lattice* if for every pair of elements $x, y \in X$, the meet and join of $\{x, y\}$ both exist.

Our interest in lattices will largely arise from application of the *Knaster-Tarski fixed point theorem*, an important result about monotone functions on complete lattices. To do so, we first define:

Definition 2.29.
A lattice $(X, \leq)$ is called *complete* if for all subsets of elements $S \subseteq X$, S has both a meet and join.

An important example of a complete lattice is the power set of some set X; the set $\mathcal{P}(\Sigma^*)$ is of this form. Now we can state the theorem.

Theorem 2.30 (Knaster-Tarski Fixed Point Theorem)**.**
Let $(X, \leq)$ be a complete lattice, and let $f : X \to X$ be a monotone function for $\leq$. Then there exist $x^\flat, x^\sharp \in X$ such that:

- $f(x^\flat) = x^\flat,\ f(x^\sharp) = x^\sharp$
- $\forall x \in X$ such that $f(x) = x$, $x^\flat \leq x \leq x^\sharp$

that is, $x^\flat$ and $x^\sharp$ are respectively the *greatest* and *least fixed points* of f.

Perhaps one of the most important consequences of this theorem is that the fixed points $x^\flat$ and $x^\sharp$ *exist at all*. The existence of fixed points for a function is non-trivial in general, but in this setting (f monotone and X a complete lattice), the theorem tells us at least one exists. Note that while $x^\flat$ and $x^\sharp$ need to exist, they need not be distinct: it is valid for f to have just one fixed point. For further reading on the Knaster-Tarski Fixed Point theorem, see [15].

Chapter 3

Reasoning with Hypotheses

3.1 Kleene Algebra for Computer Programs

An important question to ask about Kleene algebra is: why? Why represent a computer program in this formal system? What can it do to help us practically reason about programs? Before addressing *why*, we will discuss *how*.

The goal is to use letters, words, and languages to reason about programs *algebraically*. How can these things meaningfully represent computer programs? We propose the following:

- $\mathtt{a} \in \Sigma$ represents an atomic program: e.g. "PRINT$(-)$" or "$n \leftarrow n + 1$"
- $\emptyset$ represents abrupt program termination
- ε represents doing nothing, effectively "SKIP", "WAIT", etc.
- $w \in \Sigma^*$ is a list of atomic programs, like a program trace: a sequence of actions taken by the computer one after the other
- $L \in \mathcal{P}(\Sigma^*)$ is a set of traces, akin to a program specification: "these are the acceptable ways for our program to run"

Given this list of correspondences, the natural next question is: what is a program? Naturally, it is a regular expression. Abstractly, composition (or $\cdot$) such as $\mathtt{a} \cdot \mathtt{b}$ executes $\mathtt{a}$, and then $\mathtt{b}$. Choice $\mathtt{a} + \mathtt{b}$ "does" either $\mathtt{a}$ or $\mathtt{b}$. This usually manifests as **if** x **then** $\mathtt{a}$ **else** $\mathtt{b}$, where x is some condition. Lastly, the Kleene star represents *iteration*: $\mathtt{a}^*$ does $\mathtt{a}$ some number of times, effectively **while** x **do** $\mathtt{a}$ for some condition x.

Notice that the regular expressions lack the ability to talk about the boolean conditions governing the control flow: namely the condition x in both the **if** and **while** statements. We will return to the discussion of Kleene algebra lacking in expressiveness later in this section.

We can now return to the original question. Why represent a program in Kleene algebra? Assume that we have a program that we have converted into

a regular expression that is meant to somehow "algebraically" represent the program. Now what? What kind of reasoning can we do with this expression?

The short answer is, decide if it is equivalent to other programs. Say that a programmer is approaching a problem, and writes some sloppy code that is quite slow or resource-intensive. They do, however, find that what they have written *works* in the way they intend it to. However, when they attempt to optimise, refactor, and clean up the code, it suddenly stops working; or worse, looks as though it still works, only to break later. How can that problem be avoided? Ideally, the programmer would have some way to verify that the original program and their revised version are *equivalent*.

Unfortunately, deciding if two programs are equivalent is undecidable, due to a reduction from the halting problem [4]. Nevertheless, program equivalence is a high-value problem, and it being undecidable in the general setting does not mean there is nothing to be done. Below, we can see that the two (sub)programs are equivalent, without knowing in any detail what they do:

```
   ⋮                                  ⋮
1: if conditionA(n) then          1: n ← n + 1
2:     n ← n + 1                  2: if conditionA(n) then
3:     operationB(n)              3:     operationB(n)
4: else                           4: else
5:     n ← n + 1                  5:     operationC(n)
6:     operationC(n)              6: end if
7: end if
   ⋮                                  ⋮
```

That is, if one of these patterns appeared within a program, we know it can be swapped out for the other without changing the behaviour of the program. This is true regardless of the compiler, the programming language, the particular functions and variables involved, and so on.

Many details have been stripped away in this representation, for example what the conditionA is; but it is not necessary to know. Examining possible traces of both programs, we see that for both there are only two:

- $n \leftarrow n + 1$
- operationB(n)

- $n \leftarrow n + 1$
- operationC(n)

To align this with our earlier proposal that we think of words as traces, and languages as regular expressions, we would want regular expressions representing these programs to be equivalent. This is the kind of reasoning that Kleene algebra is used for. Suppose that we name some parts of our program like so:

- a refers to $n \leftarrow n + 1$

- `b` refers to OPERATIONB
- `c` refers to OPERATIONC

then we can describe the equivalence above as an instantiation of the following, familiar-looking principle:

$$\mathtt{ab} + \mathtt{ac} = \mathtt{a}(\mathtt{b} + \mathtt{c}). \tag{3.1}$$

As we mentioned to above, CONDITIONA is not a part of the program description. The syntax uses + to represent that *either* one thing or the other will happen, but says nothing about the conditions under which one or the other might happen.

By representing real computer programs in this abstracted semantics, we lose some meaning. However, we also gain the ability to decide equivalence in the system KA described in section 2. Nonetheless, we should explore what this loss of expressiveness looks like.

Example 3.1.
Consider the following program:

```
1: if a∧ ⁄a then
2:     PRINT("yes")
3: else
4:     PRINT("no")
5: end if
```

Obviously, the program will never print *"yes"*. However, using our *current* conception of converting programs into regular expressions, we lack the ability to show this fact. We would like to show that the above program is equivalent to simply:

```
1: PRINT("no")
```

However, if we let `a` refer to the atomic program PRINT("*yes*") and `b` refer to PRINT("*no*"), then the first program is $\mathtt{a} + \mathtt{b}$, and the second program is `b`. These expressions are obviously not equivalent in the standard semantics of Kleene algebra, and so by soundness KA cannot prove that they are equivalent.

Regardless, we would *like* to prove that these programs are equivalent, and an obvious way to do so would be to factor the booleans used by **if** and **while** statements into our represenations of programs. Kleene algebra by itself cannot manage this; so we turn to adding such capabilities into Kleene algebra.

3.2 Extended Kleene Algebra

While Kleene algebra is a useful tool for reasoning about simple programs, it (or rather, finite automata) lack expressive power. This has already been seen just above in example 3.1, where we wished to show that two programs were

equivalent, but Kleene algebra lacked the machinery to represent its structure in enough detail to do so.

To fix this issue, we could use a more expressive model, for example a Turing machine, but its extra expressivity also brings with it extreme computational difficulty. An attractive approach, then, is to modify Kleene algebra, while carefully maintaining its decidability, so it can be used to reason about a wider class of computational situations. We will briefly discuss one such example now, Kleene algebra with tests, or KAT [9]. KAT is a historically significant system, but we will not cover it in detail, as a deep understanding of this particular system is not necessary for the work of this thesis.

Kleene algebra with tests (KAT) distinguishes a subset of the alphabet's programs as *tests*. Tests can be thought of as programs that do not alter the state of the program, only representing some sort of boolean check. If the program $\mathtt{a}$ is a test, then running $\mathtt{a}$ means we check if $\mathtt{a}$ is true, and if it is, continue, and if it is not, terminate immediately.

The following **if** statement:

$$\textbf{if } a \textbf{ then } c \textbf{ else } d$$

can be represented as:

$$\mathtt{ac} + \widetilde{\mathtt{a}}\mathtt{d},$$

where $\widetilde{\mathtt{a}}$ is the *negation* of $\mathtt{a}$, a test that returns true if and only if $\mathtt{a}$ does not. We encourage the reader to think of a program in terms of its possible traces. Here, if we run the **if** statement above, depending on if a is true of false, the resulting run of the program will be a, c or $\widetilde{a}, d$. Without tests, this program would ignore the condition a, and be represented by the expression $\mathtt{c} + \mathtt{d}$. So as we can see, KAT represents more of the program structure than KA can.

However, in order to reason about Kleene aglebra with tests, it will not be enough to use just the axioms of Kleene algebra. For example, if $\mathtt{a}$ and $\mathtt{b}$ are tests, then it is clear that:

$$\mathtt{ab} = \mathtt{ba} \tag{3.2}$$

because there is no difference between checking the condition $\mathtt{a}$ first or checking the condition $\mathtt{b}$ first. This rule is *not* true in normal Kleene algebra; but if we know we are in the setting of Kleene algebra with tests, and $\mathtt{a}, \mathtt{b}$ are tests, then we know that (3.2) will always be true. If we know it will always be true, we ought to take (3.2) as another axiom in our equational logic.

To restate, if we want to reason about programs where an assumption like $\mathtt{ab} = \mathtt{ba}$ always holds, we will need to find a different semantics to correspond to such proofs. For example, the regular expression $\mathtt{abc}$ would normally be interpreted as the singleton set $\{\mathtt{abc}\}$. Soundness and completeness of Kleene algebra (theorem 2.22) tell us that *any* other regular expression that KA can prove is equivalent to $\mathtt{abc}$ will also be interpreted as $\{\mathtt{abc}\}$. If we adopt the assumption $\mathtt{ab} = \mathtt{ba}$ in our proofs, we will be able to prove the following equivalence:

$$\mathsf{KA} + (\mathtt{ab} = \mathtt{ba}) \vdash \mathtt{abc} = \mathtt{bac}$$

which our standard semantics of regular expressions does not support.

Our standard semantics of regular expressions were purely language-based in that an expression corresponded to a set of words. KAT, on the other hand, was originally proven to be complete for the semantics of *guarded strings* [11], essentially strings of the form $B_1 \mathtt{a} B_2 \mathtt{b} B_3$, where B_1, B_2, B_3 are *sets* of tests, and $\mathtt{a}$ and $\mathtt{b}$ are non-test programs. Sets are used because, as we remarked above, concatenation of tests is commutative, and so any consecutive run of tests can be executed in any order.

The guarded string semantics for KAT is undoubtedly useful, but developing new semantics tailored for each new system is a tall task, that can indeed be just as difficult as showing completeness. Recent work by Doumane et al. [2] presents an operation on languages, called the "*hypothesis closure*". Given a set of hypotheses H, the hypothesis closure is a map $H^* : \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ used to define a semantics for the system $\mathsf{KA} + H$:

$$[\![-]\!]_H := H^*([\![-]\!]).$$

In theorem 2 of [2], it is shown that these semantics are always sound in the sense of 2.20, essentially by construction. In practice, this means that given some Kleene algebra-based system like KAT, if we can formulate KAT's extra features in terms of extra assumptions, we obtain a sound semantics by default. These assumptions are formally called *hypotheses*: statements like $\mathtt{ab} = \mathtt{ba}$ that act as extra axioms in the equational logic.

While many Kleene algebra-based systems like KAT have their own bespoke semantics, that need not even be based on languages (regular or non) at all, this strategy takes a unified approach: the semantics of an expression in $\mathsf{KA} + H$ are determined purely from H. A later paper by Pous et al. [12] explored how completeness results for $\mathsf{KA}+H$ systems could be re-proven using the hypothesis closure; for example KAT in section 4.2. They even prove that KAT's standard "guarded string" semantics are equivalent to the new, hypothesis closure semantics (section 4.1). The core strategy of the completeness proof was to construct a "reduction" from KAT terms to standard Kleene algebra, and leverage KA's completeness to claim that KAT is complete as well. Unfortunately, designing this reduction is non-trivial in general, and no general strategy for creating one has been proposed. The objective of this thesis will be to propose such a strategy.

We now proceed to formally define what is meant by hypotheses and hypothesis closure.

3.3 Hypothesis Closure

Definition 3.2.
A *hypothesis* H is an inequation $f \geq e$ where $f, e \in \mathcal{T}(\Sigma)$ are regular expressions. We call Kleene algebra *extended* by H the system resulting from taking the axioms of KA along with H; we call this sytem $\mathsf{KA} + H$.

Remark 3.3.
While we have defined hypotheses as inequations, we consider equality hypotheses as well by taking a pair of inequalities, for example $f \geq e$ and $e \geq f$ for $e = f$.

Most work on hypotheses works with *sets* of hypotheses, not just one at a time. However, generalising the work of this thesis to finite sets of hypotheses does not change much, while resulting in much more notationally complex bookkeeping. In any case, there is already extensive work discussing the composition and decomposition of sets of hypotheses. For example, [12] section 3 studies reductions between hypotheses (where one set of hypotheses follows from another), and [2] proposition 3 gives a method of decomposing closure under a set of hypotheses into a sequence of closures.

Throughout the thesis, we will focus on a type of hypothesis that is both simple to understand, and significant in literature: *contraction*. When we want to reason with contraction, we will be taking the following as an extra axiom:

$$\mathtt{aa} \geq \mathtt{a}$$

for some $\mathtt{a} \in \Sigma$. In [6], Kappé et al. define and use contraction to show that the system KAO is complete with respect to the language-based semantics $[\![-]\!]_{\downarrow}$, which we would calculate as the hypothesis closure with respect to the hypothesis $\mathtt{aa} \geq \mathtt{a}$, for all programs $\mathtt{a}$ of a particular kind. Namely, those called (in that system) *observations*, a variant of the *tests* that we saw earlier. We can intuitively see why the hypothesis $\mathtt{aa} \geq \mathtt{a}$ is reasonable for observations: making the same observation twice is contained in doing so just once.

Example 3.4.
If we want to reason about programs with observations, we know that $\mathtt{aa} \geq \mathtt{a}$ will always be true if $\mathtt{a}$ is an observation. So what if we extend the axioms of KA with the hypothesis $\mathtt{aa} \geq \mathtt{a}$? Recall this inequation is equivalent to $\mathtt{aa} + \mathtt{a} = \mathtt{aa}$. Then we can prove:

$$\mathtt{aaaa}^* = \mathtt{aaa}^* = \mathtt{aa}^* \tag{3.3}$$

and of course, looking to our standard semantics:

$$\begin{aligned} [\![\mathtt{aaaa}^*]\!] &= \{\mathtt{a}^n : n \geq 3\} \\ [\![\mathtt{aa}^*]\!] &= \{\mathtt{a}^n : n \geq 1\} \\ \Rightarrow [\![\mathtt{aaaa}^*]\!] &\neq [\![\mathtt{aa}^*]\!] \end{aligned}$$

So our (admittedly short) proof 3.3 is unsound for our standard semantics, $[\![-]\!]$. Indeed, we will need a new semantics for our proofs using the axioms of KA with $\mathtt{aa} \geq \mathtt{a}$ added to be sound.

We now move to formally define the hypothesis closure. In [2] where it was originally defined (definition 2), and in subsequent papers such as [12], the hypothesis closure is defined directly. For the purposes of this thesis, we will split that definition into two parts: first we define the function H, which

represents "one step" of the closure. Then we will define the closure operation H^*, which is H applied repeatedly until it stabilises (potentially at infinity). In [2], H is called ψ, and $H^*_\bullet$ is called cl_H. To begin, we will define H.

Definition 3.5.
Let H be a (finite) set of hypotheses of the form $f \geq e$, and let $L \in \mathcal{P}(\Sigma^*)$ be a language. The *language hypothesis function* of H is defined as:

$$H(L) := L \cup \{uw_ev : \exists f \geq e \in H \text{ such that } w_e \in [\![e]\!], \text{ and } u[\![f]\!]v \subseteq L\}$$

This is an important definition and it is quite dense, so we take some time to unpack it here. For the moment, we focus on just one hypothesis at a time. Given a language L and a hypothesis $f \geq e$, a word w is in $H(L)$ if and only if it is in L, or we can break the word into three subwords u, w_e, v that have the following properties:

- $w = uw_ev$
- $u[\![f]\!]v \subseteq L$, that is, for all $w_f \in [\![f]\!]$, $uw_fv \in L$
- $w_e \in [\![e]\!]$

To calculate the language hypothesis function of $f \geq e$ applied to a language L, we take L as a starting point, and iterate through pairs of words u, v. For each, we check if $u[\![f]\!]v \subseteq L$; we will often call u, v a *prefix/suffix pair* for this reason. If u, v are indeed such a pair of words, then the function will add in the words of $u[\![e]\!]v$, that is words of the form uw_ev where $w_e \in [\![e]\!]$. Due to the flow of this process, we will refer to f in this hypothesis as the *assumption*, and e as the *conclusion*.

Example 3.6.
Take the hypothesis $\texttt{aa} \geq \texttt{a}$, and consider the following language:

$$L := \{\texttt{aabaa, abcaabb}\}$$

if we apply H to this language, we will obtain:

$$H(L) = \{\texttt{aabaa, abaa, aaba, abcaabb, abcabb}\}.$$

How? Firstly, we look at our hypothesis. Both of the expressions in our hypothesis represent just one word; we will call such expressions *singleton expressions*. Working with singleton expressions makes the calculation of H much simpler, because the condition of $u[\![f]\!]v \subseteq L$ is actually just $u \cdot \texttt{aa} \cdot v \in L$. With that in mind, we need only look for anywhere in words that $\texttt{aa}$ appears. For example, if we take $u = \varepsilon$ and $v = \texttt{baa}$, then we can see that:

$$u \cdot \texttt{aa} \cdot v = \varepsilon \cdot \texttt{aa} \cdot \texttt{baa} = \texttt{aabaa} \in L$$

and so we conclude that $u \cdot \mathtt{a} \cdot v = \mathtt{abaa} \in H(L)$. We apply similar reasoning to find the other two new words in $H(L)$:

$$\mathtt{aab} \cdot \mathtt{aa} \cdot \varepsilon \mapsto \mathtt{aab} \cdot \mathtt{a} \cdot \varepsilon$$
$$\mathtt{abc} \cdot \mathtt{aa} \cdot \mathtt{bb} \mapsto \mathtt{abc} \cdot \mathtt{a} \cdot \mathtt{bb}$$

Observe that in some sense there is more to do, after one application of H: we can apply it again, and that would expand the language further, obtaining the word $\mathtt{aba}$. At that point, applying H would do nothing. That language we obtain, when there is nothing more to do, is the language we are seeking to define as the hypothesis closure.

Example 3.7.
Before proceeding, we treat more complex examples that foreshadow an important distinction for later: the hypothesis $\mathtt{aa} \geq \mathtt{a}$ has only one word in both its assumption language ($\mathtt{aa}$) and its conclusion language ($\mathtt{a}$); as stated above, this makes the calculation of H much simpler. Let us now take the hypothesis $(\mathtt{a} + \mathtt{b})\mathtt{c} \geq \mathtt{d} + \mathtt{e}$, and consider the following language:

$$L_1 := \{\mathtt{dbce}, \mathtt{dace}\}$$

To check if a word is in $H(L_1)$, we need a few things. Firstly, some choice of prefix/suffix such that for every word in the assumption $(\mathtt{a} + \mathtt{b})\mathtt{c}$, we can wrap the word in the prefix and suffix, and the resulting word is in L_1. In this case, our assumption has only two words in it: $\mathtt{ac}$ and $\mathtt{bc}$. We choose for our prefix $\mathtt{d}$, and for our suffix we choose $\mathtt{e}$. Then, observing that both $(\mathtt{d})(\mathtt{bc})(\mathtt{e})$ and $(\mathtt{d})(\mathtt{ac})(\mathtt{e})$ are in L_1, we can conclude that for any word in the conclusion $\mathtt{d} + \mathtt{e}$, we can wrap it with our prefix and suffix, and that word is in $H(L_1)$. Explicitly:

$$H(L_1) = \{\mathtt{dbce}, \mathtt{dace}, \mathtt{dde}, \mathtt{dee}\}.$$

Now consider these two languages:

$$L_2 := \{\mathtt{dace}\} \quad L_3 := \{\mathtt{dbce}, \mathtt{eacd}\}$$

We can immediately say that $H(L_2) = L_2$, for the simple reason that our assumption $(\mathtt{a} + \mathtt{b})\mathtt{c}$ has two words in it, and this language has only one. No matter what prefix/suffix pair u, v we choose, we can not have that both $u \cdot \mathtt{ac} \cdot v$ and $u \cdot \mathtt{bc} \cdot v$ are in L_2.

We will also see that $H(L_3) = L_3$, but for a more nuanced reason. While we do have words with both $\mathtt{bc}$ and $\mathtt{ac}$ as infixes, we cannot make a consistent choice for the prefix/suffix pair. By this we mean, if we choose $u = \mathtt{d}$ and $v = \mathtt{e}$, then:

$$(\mathtt{d})(\mathtt{bc})(\mathtt{e}) \in L_3 \quad (\mathtt{d})(\mathtt{ac})(\mathtt{e}) \notin L_3$$

On the other hand, if we try instead $u = \mathtt{e}$ and $v = \mathtt{d}$:

$$(\mathtt{e})(\mathtt{bc})(\mathtt{d}) \notin L_3 \quad (\mathtt{e})(\mathtt{ac})(\mathtt{d}) \in L_3$$

We need to not only have all words from the assumption be infixes, we need to be able to wrap them all in a fixed prefix and suffix for the hypothesis function to add any new words.

Notice that each time we find a prefix/suffix pair that allows H to add new words, we include one new word for each of the words in the conclusion. One can imagine how much more complex this whole process becomes when the assumption can be much larger and more complex, even infinite.

By definition, H is inflationary: for all L, $L \subseteq H(L)$. So, we can imagine that by applying this function to language L over and over, H can be used to define a "closure" operation on languages. To do so, we need to apply this function to the language until it stabilises, or proceed infinitely if it does not, reaching the answer "at infinity". Recall the Knaster-Tarski fixed point theorem, which guarantees a least fixed point for monotone functions on complete lattices, by applying the function to the bottom element. We will use this theorem to guarantee the existence of our desired closure.

Proposition 3.8.
Let H be a fixed hypothesis. $H : \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ is monotone with respect to $\subseteq$.

Proof. Given languages $L \subseteq L'$, we want to show that $H(L) \subseteq H(L')$. Let $w \in H(L)$. If $w \in L$, then $w \in L'$, and so $w \in H(L')$, since $L \subseteq L'$, and H is inflationary. On the other hand, if $w \notin L$, then by the definition of H, it must be that $w = uw_ev$ where $w_e \in [\![e]\!]$, and u, v are such that $u[\![f]\!]v \subseteq L$. Therefore, $u[\![f]\!]v \subseteq L'$ as well, and so $uw_ev \in H(L')$. 完

Since H is monotone on $(\mathcal{P}(\Sigma^*), \subseteq)$, we can apply the Knaster-Tarski fixed point theorem to learn that H must have a least fixed point, but for most hypotheses, that will just be $\emptyset$. What we would really like to do is to start from L, applying H until we reach a fixed point. To achieve this, we create a new function, and build the language L into the function itself:

$$\begin{aligned} H_L : \mathcal{P}(\Sigma^*) &\to \mathcal{P}(\Sigma^*) \\ L' &\mapsto L \cup H(L') \end{aligned}$$

By definition, there can be no fixed point of H_L that is below L itself. We can use this least fixed point to claim that our notion of using H for a closure operation is meaningful:

Proposition 3.9.
Let H be a hypothesis, and L a language. There exists a least language (with respect to $\subseteq$) $\overline{L}$ such that:

1. $L \subseteq \overline{L}$
2. $H(\overline{L}) = \overline{L}$

Proof. The objective is to show existence of a language $\overline{L}$ that is a fixed point of H. We will aim to use the Knaster-Tarski fixed point theorem, as (with the right strategy) it will give us exactly this $\overline{L}$.

Firstly, we note that the lattice of languages over a given alphabet is complete, as it is the power set of the set Σ^*. Now we will show that H_L is

monotone. Let L_1,$L_2 \in \mathcal{P}(\Sigma^*)$ such that $L_1 \subseteq L_2$. We want to show that $H_L(L_1) \subseteq H_L(L_2)$. By definition:

$$H_L(L_1) = L \cup H(L_1) \subseteq L \cup H(L_2) = H_L(L_2)$$

where the middle inclusion follows by monotonicity of H. We can now apply the Knaster-Tarski fixed point theorem to H_L, and obtain its least fixed point: a language $\overline{L}$ such that $H_L(\overline{L}) = \overline{L}$, and the least such that this is the case.

Now we claim this is the desired language. By the definition of $\overline{L}$ and the definition of H_L:

$$\overline{L} = H_L(\overline{L}) = L \cup H(\overline{L}). \tag{3.4}$$

Therefore, $L \subseteq \overline{L}$, fulfilling condition (1). Now we need to show that $H(\overline{L}) = \overline{L}$ as well. Note that equation 3.4 also tells us that $H(\overline{L}) \subseteq \overline{L}$, and the inclusion in the other direction follows because H is inflationary, fulfilling condition (2).

Lastly, we need to show that $\overline{L}$ is the least language such that these conditions hold. Let $\hat{L}$ be a language such that $L \subseteq \hat{L}$ and $H(\hat{L}) = \hat{L}$. We want to show that $\overline{L} \subseteq \hat{L}$. We need only observe that by definition, $\overline{L}$ is the least fixed point of H_L, and show that $\hat{L}$ is also a fixed point of H_L:

$$H_L(\hat{L}) = L \cup H(\hat{L}) = L \cup \hat{L} = \hat{L} \qquad \text{完}$$

Now, we can use one of two equivalent definitions to finally define the desired closure operator:

Definition 3.10.
Let H be a hypothesis. The *language hypothesis closure* H^* of a language L is the least language $\overline{L}$ such that $L \subseteq \overline{L}$ and $H(\overline{L}) = \overline{L}$. Alternately, it is the least fixed point of the function:

$$H_L : L' \mapsto L \cup H(L')$$

We refer to [2] for proofs of many simple properties about this function, for example that it is indeed a closure operator. For the purposes of this thesis, we name some of the properties about H^* that we will use, again from [2]:

Lemma 3.11.
Let H be a hypothesis of the form $f \geq e$, and let $L \in \mathcal{P}(\Sigma^*)$ be arbitrary.

- $L \subseteq H^*(L)$
- H^* is monotone
- $u \cdot H(L) \cdot v \subseteq H(uLv)$, for all $u, v \in \Sigma^*$

It is important to note that while the H function *may* stabilise in finitely many steps, in general it will not.

Example 3.12.
For example, with the hypothesis $\mathtt{a} \geq \mathtt{aa}$, the function H will never stabilise on the language $\{\mathtt{a}\}$:

$$\begin{aligned} H^0(\{\mathtt{a}\}) &= \{\mathtt{a}, \mathtt{aa}\} \\ H^1(\{\mathtt{a}\}) &= \{\mathtt{a}, \mathtt{aa}, \mathtt{aaa}\} \\ H^2(\{\mathtt{a}\}) &= \{\mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \mathtt{aaaa}\} \\ &\vdots \end{aligned}$$

With H^* defined, we can now officially define the hypothesis closure semantics we have been pursuing for much of this section:

Definition 3.13.
Let H be a hypothesis. We define the *hypothesis closed semantics* for a regular expression r:

$$[\![r]\!]_H := H^*([\![r]\!])$$

A hypothesis H is called *complete* if the proof system $\mathsf{KA} + H$ is complete for the hypothesis closed semantics $[\![-]\!]_H$. That is, for all regular expressions e, f:

$$[\![e]\!]_H = [\![f]\!]_H \implies \mathsf{KA} + H \vdash e = f$$

It is shown in [2] theorem 2 that for any set of hypotheses H, $\mathsf{KA} + H$ is sound for the semantics $[\![-]\!]_H$. Now, to show completeness (if at all possible), it is very effective to find a way to calculate the language $H^*([\![r]\!])$ for an arbitrary regular expression r; this relates the semantics back to KA, and its completeness can be extended to $\mathsf{KA} + H$. For more detail, see section 5.1, and [12].

When endeavouring to define a transformation on regular expressions, a fruitful approach is a construction on the automata that can then be converted back into the desired regular expression. This will be the approach that we take. So, we can at last state our objective in more detail:

> Given a hypothesis H and a regular expression r, we will define an operation that, when applied to an automaton accepting $[\![r]\!]$, produces an automaton that accepts $[\![r]\!]_H$.

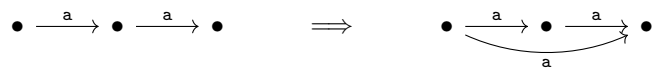
Another way of phrasing the goal is that we will *extend the hypothesis closure operation to automata.*

3.4 Examples of Automaton Closure

We now look at some examples to illustrate what we mean by extending hypothesis closure to automata. We begin with a familiar example, contraction.

Example 3.14.
Let $H := \{\mathtt{aa} \geq \mathtt{a}\}$. It is fairly straightforward to look at contraction and realise

it as a transformation on automata: we want to take any instance of two a's and allow there to be just one, so we just look at any sequence of two a transitions in the automaton and add an transition bridging the distance. Ostensibly, a sort of "transitive closure":

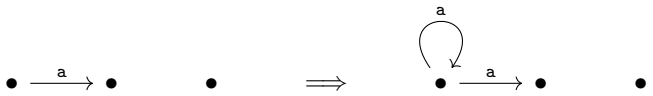


Remark 3.15.
For any set of hypotheses, it is important to specify how the letters are being quantified. As we see above for contraction, the hypothesis $\mathtt{aa} \geq \mathtt{a}$ by itself applies only to the *specific* letter $\mathtt{a} \in \Sigma$, and no others. On the other hand, a "general" contraction is one such hypothesis for every letter in Σ, and this would correspond to transitive closure across the automaton.

Example 3.16.
We now give more examples of hypotheses, and for each, we will informally describe how to realise the closure as a transformation on automata. These examples are meant to give the reader a sense of what we are doing: given a hypothesis, discern how it can be realised on automata.

- Inflation: $\mathtt{a} \geq \mathtt{aa}$

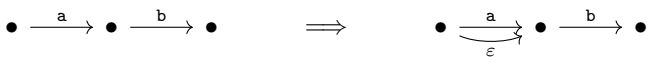
  A dual to contraction: if a program executes a once, it can then execute a again without affecting program behaviour. On automata, it is sort of a half-measure to reflexive closure (see *insertion*, below) where any state with an a arrow out of it must have a reflexive a arrow.



  Notice that the both the middle and rightmost states are unchanged, in that they have no more outgoing transitions, because they have no outgoing a transitions.

- Removal: $\mathtt{a} \geq \varepsilon$

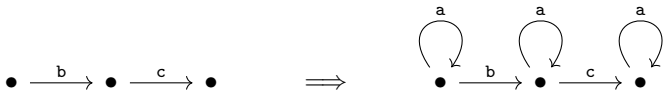
  The atomic program a can be removed from a program, and it will not affect the semantics. On automata, each a arrow will have to "reflect transitions", meaning any transition valid in the image of an a arrow must also be valid from the domain of the a arrow. Alternately, any a transition can be "skipped":



- Insertion: $\varepsilon \geq \mathtt{a}$.

  The atomic program a can be inserted anywhere in a nonempty program. Use of this hypothesis might correspond to the program a being labeled

somehow unrelated structurally to those we care to reason about; for example, inserting a print statement somewhere in the program, as programmers often do for debugging. In a literal sense this *does* change the behaviour of the program, but maybe not in a way that we care about. On automata, it will correspond to reflexive closure under the specified letters.



For each of the above hypotheses, we described a corresponding construction on automata; but we relied heavily on intuition, and went mostly by eye for each. We also did not formally verify that they are correct. How might we approach this for a more complex hypothesis or set of hypotheses, and be sure we have obtained a correct answer? This is the purpose that a general automaton closure will serve.

3.5 The Objective: Automaton Closure

To review, it is an open problem how to compute a regular expression corresponding to the "hypothesis closure" semantics of a regular expression for arbitrary sets of hypotheses. Working with automata to represent regular expressions is intuitive and effective, so that will be our approach.

As a result, we would like to generalise the "hypothesis closure" operation defined on languages to automata as well. The result will be a construction on automata:

$$H^* : \mathsf{NA} \rightharpoonup \mathsf{NA}$$

where $\rightharpoonup$ indicates that the function is partial. Naturally we would like this to be a total function, but as we will see, this is not possible. Chapter 5 is primarily concerned with this problem.

Note that we are overloading notation: we have the *language* hypothesis function H, which we used to define the *language* hypothesis closure function H^*. We are now aiming to define an *automaton* hypothesis function H, and use it to define an *automaton* hypothesis closure H^*.

Before describing the construction, let us be more explicit about the objective. We need to define what it means for H^* to *work* as a construction on automata. One possible condition, given that the motivation was simply to compute the hypothesis closure of a regular expression, is the following:

$$l_{H^*(\mathcal{X})}(x) = H^*(l_{\mathcal{X}}(x)) \quad \text{where } X, x \text{ are such that: } [\![r]\!] = l_{\mathcal{X}}(x). \tag{3.5}$$

This one is in some sense quite "economical", as it focuses on a single state and requests nothing of the others. For the purposes of just computing the closure of an expression, such a condition would work; but in practice it will be difficult

to generalise, as we will see in example 3.17. It will be easier to work with the following condition which is ostensibly the same, but for all states in $\mathcal{X}$, rather than just one.

$$\forall x \in X, \quad l_{H^*(\mathcal{X})}(x) = H^*(l_{\mathcal{X}}(x)). \tag{3.6}$$

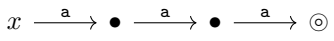
However, as we will discuss in section 4.18, we can actually do a bit better. Notice that the condition 3.6 quantifies over all states in X, the state set of $\mathcal{X}$, rather than the state set of the expanded automaton, $H^*(\mathcal{X})$. We are claiming to define a hypothesis closure operation on automata, so a reasonable goal would be that the languages of all states x in $H^*(\mathcal{X})$ are closed under the hypotheses.

$$\forall x \in H^*(\mathcal{X}), \quad l_{H^*(\mathcal{X})}(x) = H(l_{H^*(\mathcal{X})}(x)). \tag{3.7}$$

This is in a sense a slightly weaker condition, as it only requires that the languages of all states *be* closed, not that those languages be the hypothesis closures of the languages they had in $\mathcal{X}$; if given the hypothesis $\mathtt{a} \geq \mathtt{b}$, the empty language is closed under this hypothesis, so removing all transitions and accepting states from an automaton would be a solution consistent with condition 3.7.

However, requiring a version of condition 3.6 for all states is not reasonable, because if a state is in $H^*(\mathcal{X})$ and not in $\mathcal{X}$, there is no "original" language to compare its language in $H^*(\mathcal{X})$ to.

Example 3.17.
Consider the following automaton, and let $H := \mathtt{aa} \geq \mathtt{a}$ be a hypothesis.



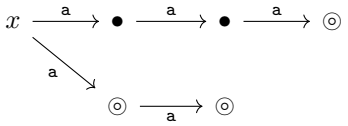
Now, the language accepted by the state x is:

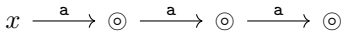
$$l_{\mathcal{X}}(x) = \{\mathtt{aaa}\}$$

it is straightforward to compute the language we want to make x accept in the output automaton:

$$H^*(l_{\mathcal{X}}(x)) = H^*(\{\mathtt{aaa}\}) = \{\mathtt{aaa}, \mathtt{aa}, \mathtt{a}\}$$

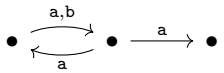
The following automata both adhere to condition (3.5):





The state x in both of these automata accepts exactly $H^*(l_{\mathcal{X}}(x))$; so they could both be valid outputs for condition (3.5). However, it is not at all clear how we can generalise the strategies used to obtain these automata, nor how we can be

sure those strategies will work on a more non-trivial automaton, such as this one:



For the time being, we will set condition (3.6) as our goal. We will see, however, that for our construction it will be sufficient to show that condition (3.7) holds; see corollary 4.18.

Chapter 4

Singleton Hypothesis Closure, $H^*_\bullet$

To review, given a hypothesis H, the objective is to define a map on automata $H^* : \mathsf{NA} \to \mathsf{NA}$ to act as a "hypothesis closure" on automata. We will approach the construction by doing a simpler version first, for a restricted form of hypothesis.

Definition 4.1.
Let $H := f \geq e$ be a hypothesis. We say that H is a *singleton hypothesis* if the assumption f represents a singleton language. That is,

$$[\![f]\!] = \{w_f\}$$

for some word $w_f \in \Sigma^*$.

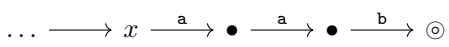
The construction when working with a singleton hypothesis is simpler, and we will call this version the *singleton* version of the construction, $H^*_\bullet$.

Why is it worth separating the singleton case from the general case? A counterexample can be seen in example 6.1, but intuitively: if f represents only one word w_f, then the closure of L is computed by finding every word w in the language that has w_f as a subword, so $w = uw_fv$, and adding a new word uw_ev, where w_e is any word in $[\![e]\!]$. One can imagine why finding subwords across the language, one word at a time, is simpler than finding prefix/suffix pairs that must consider all words in L simultaneously.

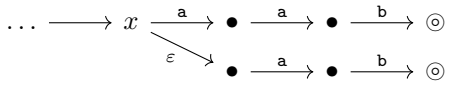
In a similar fashion to how we defined $H : \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ as a stepping stone to defining $H^* : \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$, we will start by defining a construction $H_\bullet : \mathsf{NA} \to \mathsf{NA}$ that will serve as "one step" in the larger construction.

Now we informally describe what $H_\bullet$ will do. For each state in the automaton, we will check: should this state accept any more words than it currently does? If not, we do not need to do anything, so we move on. If the state ought to accept more words than it currently does, we will force it to do so, by "pasting" an automaton onto it accepting the desired words.

Example 4.2.
Let $H := \texttt{aa} \geq \texttt{a}$. Consider the following (sub)automaton:



We can quickly observe that the state x accepts the word `aab`, and so according to the hypothesis, ought to accept `ab` too. Meanwhile, the other three states accept `ab`, `b`, and ε respectively, and so nothing needs to change for these states. So how do we ensure x also accepts `ab`? We propose the following:



This process of adding in the lower row of states and transitions, adding an ε transition from x into the newly added states, is what we will come to call "pasting".

Remark 4.3.
Importantly, when we paste in the new automaton, we are adding new states: this will mean more states that need to be checked. As a result, the construction could operate on the initial set of states, add some new states, then loop over the new states, add some new states, and proceed infinitely. We will offset this problem by not considering newly added states in one application of $H_\bullet$, considering only those that were there when it started.

Of course, we need to investigate the new states eventually, which will happen in subsequent application(s) of $H_\bullet$; and so still there is still significant risk of non-termination. Indeed, we will not expect the construction to terminate in a great deal of cases, but we defer any further comments on this topic to chapter 5.

We begin by defining some important notions that will be central to the work that follows: a generalised Brzozowski derivative, and automaton pasting.

4.1 Generalised Brzozowski Derivative

In chapter 2 we defined the Brzozowski derivative (definition 2.15), an operation on languages that can be used to check what words in a language have a particular prefix. For that version, we only defined the derivative with respect to a *single word*; in this section we will expand the Brzozowski derivative so that we can take a derivative with respect to *languages* as well.

Definition 4.4 (Generalised Brzozowski Derivative)**.**
Let L, M be regular languages. The *generalised Brzozowski derivative* of L with respect to M is defined:

$$M^{-1}L := \bigcap_{w \in M} w^{-1}L = \{v \in \Sigma^* : wv \in L, \text{ for all } w \in M\}$$

as with the word-based Brzozowski derivative, we can also lift the definition to regular expressions r, s:

$$s^{-1}r := (\llbracket s \rrbracket)^{-1}\llbracket r \rrbracket$$

We now want to ensure that this generalised form maintains some of the nice properties that we introduced for standard Brzozowski derivatives:

Proposition 4.5.
Let L, M, N be regular languages.

1. $M^{-1}L$ is a regular language
2. it is decidable whether $M^{-1}L$ is empty
3. $(M \cdot N)^{-1}L = N^{-1}(M^{-1}L)$

Proof. We again let L, M, N be regular languages.

1. Recall from lemma 2.5 that the intersection of finitely many regular languages is as well a regular language. To overcome the fact that $M^{-1}L$ is defined as an infinite intersection, we will show that it is the intersection of an at-most finite number of different languages. We will use Thompson's construction and proposition 2.17 for this purpose.

   L is a regular language, so using Thompson's construction there is a pointed automaton $(\mathcal{X}, x_0)$ such that $L = l_{\mathcal{X}}(x_0)$. Now using proposition 2.17 we can rewrite:

   $$\begin{aligned} M^{-1}L = M^{-1}l_{\mathcal{X}}(x_0) &= \bigcap_{w \in M} w^{-1}l_{\mathcal{X}}(x_0) \\ &= \bigcap_{w \in M} \left(\bigcup_{y \in \delta(x_0, w)} l_{\mathcal{X}}(y) \right) \end{aligned}$$

   We will now argue that despite there being infinitely many words in Σ^*, there are only finitely many values that the union:

   $$\bigcup_{y \in \delta(x_0, w)} l_{\mathcal{X}}(y) \tag{4.1}$$

   can take. Of course, $l_{\mathcal{X}}(y)$ is the language accepted by a state y in $\mathcal{X}$, and $\mathcal{X}$ is a finite automaton, so there are finitely many such languages; we also note that all of these are regular. So given a word w, the union described in (4.1) is the union of the languages for some finite number of states, even if there are infinitely many words in M. Recall from proposition 2.5 that the union of finitely many regular languages is also regular. Therefore, the language in (4.1) is indeed regular.

   We can identify each union of the form in (4.1) with a subset of X (namely $\delta(x_0, w)$). Again because $\mathcal{X}$ is finite, there are finitely many such subsets

possible: $2^{|X|}$ many. Therefore, the intersection:

$$M^{-1}L = \bigcap_{w \in M} \left(\bigcup_{y \in \delta(x_0, w)} l_{\mathcal{X}}(y) \right)$$

is an intersection of finitely many regular languages, so it is itself regular.

2. Follows from part (1) of this lemma and decidability of KA (theorem 2.6).

3. We now want to prove that $(M \cdot N)^{-1}L = N^{-1}(M^{-1}L)$. Again using lemma 2.18, this is straightforward with some manipulation:

$$\begin{aligned}
(M \cdot N)^{-1}L &= \{v \in \Sigma^* : \forall w \in (M \cdot N), wv \in L\} \\
&= \{v \in \Sigma^* : \forall w_1 \in M, \forall w_2 \in N, (w_1 w_2)v \in L\} \\
&= \{v \in \Sigma^* : \forall w_2 \in N, \big(\forall w_1 \in M, w_1(w_2 v) \in L\big)\} \\
&= \{v \in \Sigma^* : \forall w_2 \in N, w_2 v \in M^{-1}L\} \\
&= N^{-1}(M^{-1}L) \qquad \text{完}
\end{aligned}$$

4.2 Automaton Pasting

Now we formalise the notion of *automaton pasting*. It will be our main tool for expanding the language accepted by a state, and will be central to defining our proposed construction.

Definition 4.6.
Let $\mathcal{X}$ be an automaton, and $x \in X$ one of its states. Let $(\mathcal{Y}, y_0)$ be a pointed automaton. We define a *pasting operation* that produces a new automaton $\mathcal{X}[x \to \mathcal{Y}] := (X', \delta_{\mathcal{X}'}, \nu_{\mathcal{X}'})$ such that:

$$\begin{aligned}
X' &:= X \sqcup Y \\
\delta_{\mathcal{X}'} &:= \delta_{\mathcal{X}} \sqcup \delta_{\mathcal{Y}} \sqcup \{x \xrightarrow{\varepsilon} y_0\} \\
\nu_{\mathcal{X}'} &:= \nu_{\mathcal{X}} \sqcup \nu_{\mathcal{Y}}
\end{aligned}$$

We may also say that we "paste $\mathcal{Y}$ onto $\mathcal{X}$ at x" to refer to the automaton $\mathcal{X}[x \to \mathcal{Y}]$, with no reference to y_0 when it is clear from context. We also introduce some convenient notations for doing multiple pastes onto the same automaton:

$$\begin{aligned}
\big(\mathcal{X}[x \to \mathcal{Y}]\big)[x' \to \mathcal{Y}'] &:= \mathcal{X}[x \to \mathcal{Y}, x' \to \mathcal{Y}'] \\
\mathcal{X}[x \to \mathcal{Y}_1, x \to \mathcal{Y}_2, \dots x \to \mathcal{Y}_n] &:= \mathcal{X}[x \to \mathcal{Y}_1, \dots \mathcal{Y}_n] \\
\mathcal{X}[x_1 \to \mathcal{Y}, x_2 \to \mathcal{Y}, \dots x_n \to \mathcal{Y}] &:= \mathcal{X}[x_1, \dots x_n \to \mathcal{Y}]
\end{aligned}$$

Remark 4.7.
Recall that in Thompson's construction (see [4] for details), when (recursively) defining an automaton accepting the expression $e \cdot f$, we would find automata

accepting e and f, which we will call $\mathcal{Z}_e$ and $\mathcal{Z}_f$, and define the automaton $\mathcal{Z}_{e \cdot f}$ by putting epsilon transitions from the accepting states of $\mathcal{Z}_e$ into the initial state of $\mathcal{Z}_f$, and taking only $\mathcal{Z}_f$'s accepting states (and not $\mathcal{Z}_e$'s) as accepting in the new automaton.

Pasting is extremely similar, but it keeps *all* of the accepting states of both $\mathcal{X}$ and $\mathcal{Y}$. This has the effect of strictly *expanding* the language accepted by states that are having automata pasted onto them.

Lemma 4.8.
Let $\mathcal{X}$ be an automaton, and $x \in X$ one of its states. Let $(\mathcal{Y}, y_0)$ be a pointed automaton.

1. Pasting is inflationary with respect to $\sqsubseteq$ (def. 2.25):

$$\mathcal{X} \sqsubseteq \mathcal{X}[x \to \mathcal{Y}].$$

2. If an $\mathcal{X}[x \to \mathcal{Y}]$ trace visits a state y in $\mathcal{Y}$, then all states the trace visits *after* y are also in $\mathcal{Y}$. In particular, if a trace visits a state in $\mathcal{Y}$, then it ends in $\mathcal{Y}$.

3. Let r be a regular expression, and let $\mathcal{Z}_r$ be the output of Thompson's expression for r. Then:

$$l_{\mathcal{X}[x \to \mathcal{Z}_r]}(x) = l_{\mathcal{X}}(x) \cup \mathit{loop}_{\mathcal{X}}(x) \cdot [\![r]\!]$$

   where $\mathit{loop}_{\mathcal{X}}(x) := \{w \in \Sigma^* : x \in \delta(x, w)\}$ is the set of all words such that there is an $\mathcal{X}$ trace going from x to x.

4. Let $x_1, x_2 \in X$ be $\mathcal{X}$ states, and $\mathcal{Y}_1, \mathcal{Y}_2$ be pointed automata. Then:

$$\mathcal{X}[x_1 \to \mathcal{Y}_1, x_2 \to \mathcal{Y}_2] = \mathcal{X}[x_2 \to \mathcal{Y}_2, x_1 \to \mathcal{Y}_1].$$

Proof. Let $\mathcal{X} = (X, \delta, \nu)$, $\mathcal{X}[x \to \mathcal{Y}] = (X', \delta', \nu')$.

1. In order to show that $\mathcal{X} \sqsubseteq \mathcal{X}[x \to \mathcal{Y}]$, by lemma 2.26 it is sufficient to show that $X \subseteq X'$, $\nu \subseteq \nu'$, and for all $x \in X$, $w \in \Sigma^*$, $\delta(x, w) \subseteq \delta'(x, w)$. All three statements follow quite easily from the definition of pasting, as each of X', δ', and ν' is defined as part of a disjoint union from X, δ, and ν.

2. We consider the automaton $\mathcal{X}[x \to \mathcal{Y}]$. We can think of it as the disjoint union of $\mathcal{X}$ and $\mathcal{Y}$, with an ε transition $x \xrightarrow{\varepsilon} y_0$ added. So we can observe that this automaton has no transitions that begin at a state in $\mathcal{Y}$ and end at a state in $\mathcal{X}$. This means that if there is an $\mathcal{X}[x \to \mathcal{Y}]$ trace that visits a state $y \in X_{\mathcal{Y}}$, each step after must follow a transition to another state in $\mathcal{Y}$.

3. Recall that implicit in our notation is that the output of Thompson's construction for r is the pointed automaton $(\mathcal{Z}_r, z_0)$ such that:

$$l_{\mathcal{X}_r}(z_0) = [\![r]\!].$$

This automaton is pasted onto $\mathcal{X}$ at x, and we want to show that this alters the language accepted by x in particular way; namely,

$$l_{\mathcal{X}[x \to \mathcal{Z}_r]}(x) = l_{\mathcal{X}}(x) \cup [\![r]\!].$$

Let $w \in l_{\mathcal{X}[x \to \mathcal{Z}_r]}(x)$. This means that there exists an accepting $\mathcal{X}[x \to \mathcal{Z}_r]$ trace for w. If every state in the trace is a state in $\mathcal{X}$, then similarly to (2) of this lemma, we observe that the trace can not have traversed any of the transitions that were newly added in the pasting operation (as they all *end* at a state in $\mathcal{Z}_r$). This means that all of the states and transitions in the trace were already in $\mathcal{X}$, and the trace ends at an accepting state in $\mathcal{X}$. Therefore, in the case that all states in the trace were already in $\mathcal{X}$, the trace must also be an accepting $\mathcal{X}$ trace, and therefore $w \in l_{\mathcal{X}}(x)$.

On the other hand, if the trace for w visits any states in $\mathcal{Z}_r$, we know by (2) of this lemma that the trace must then end at an accepting state z in $\mathcal{Z}_r$. As we have observed, the only way to start in $\mathcal{X}$ and reach a state in $\mathcal{Z}_r$ is via the added transition $x \xrightarrow{\varepsilon} z_0$. We break the trace into two pieces: before it traverses this transition, and after. Let $w = w_1 w_2$ such that:

$$x \xrightarrow{w_1} x \xrightarrow{\varepsilon} z_0 \xrightarrow{w_2} z$$

We observe that the latter half is simply an accepting $\mathcal{Z}_r$ trace for w_2. This means that $w_2 \in [\![r]\!]$. On the other hand, the first half of the trace goes from x to x, meaning that $x \in \delta(x, w)$, so by definition, $w \in \mathit{loop}_{\mathcal{X}}(x)$.

4. The claim largely follows by commutativity of the disjoint union operation on the constituent parts of the automata. We let $\mathcal{X} = (X, \delta, \nu)$, $\mathcal{Y}_1 = (Y_1, \delta_1, \nu_1)$, and $\mathcal{Y}_2 = (Y_2, \delta_2, \nu_2)$. The state sets of $MX[x_1 \to \mathcal{Y}_1, x_2 \to \mathcal{Y}_2]$ and $\mathcal{X}[x_2 \to \mathcal{Y}_2, x_1 \to \mathcal{Y}_1]$ are defined as, respectively,

$$(X \sqcup Y_1) \sqcup Y_2 = (X \sqcup Y_2) \sqcup Y_1.$$

The situation is quite similar for the transitions and accepting states.

$$(\delta \sqcup \delta_1) \sqcup \delta_2 = (\delta \sqcup \delta_2) \sqcup \delta_1$$
$$(\nu \sqcup \nu_1) \sqcup \nu_2 = (\nu \sqcup \nu_2) \sqcup \nu_1 \qquad \text{完}$$

4.3 Definitions of $H_\bullet$, $H_\bullet^*$

Now we can move into defining the automaton closure operation. The operation will, for each state in the automaton, check if the state needs to be altered, and

if it does, paste an automaton onto the state. Otherwise, it does nothing to the state. From here on, we will fix a singleton hypothesis $H := f \geq e$.

We start by describing the process at a particular state: let $\mathcal{X}$ be an automaton, and fix a state $x \in X$. Now we want to check if this state needs to be altered. We propose the condition:

$$f^{-1}l_{\mathcal{X}}(x) \stackrel{?}{\subseteq} e^{-1}l_{\mathcal{X}}(x). \tag{4.2}$$

If this is the case, then we do nothing to x. We know this condition is decidable by theorem 2.6. Recall by definition that in a hypothesis closed language, for all u, v,

$$u[\![f]\!]v \subseteq L \quad \Rightarrow \quad u[\![e]\!]v \subseteq L. \tag{4.3}$$

Condition 4.2 checks if the implication 4.3 holds "locally" (in that we ignore the prefix u). If a word w is in $f^{-1}l_{\mathcal{X}}(x)$, it means that $w_f w \in l_{\mathcal{X}}(x)$, where $[\![f]\!] = \{w_f\}$. Can we replace w_f with any word from e? In other words, is $w_e w \in l_{\mathcal{X}}(x)$ for every $w_e \in [\![e]\!]$?

As stated, if condition 4.2 is met, then (for now) there is nothing to do for the state x and we move on. If it is false, however, we need to alter the state x, extending its language. To achieve this, we let $\mathcal{Z}$ be the output of Thompson's construction for the regular expression $[\![e]\!] \cdot f^{-1}l_{\mathcal{X}}(x)$, and paste it onto $\mathcal{X}$ at x.

Now we will repeat this operation over all of the states in X.

Definition 4.9 ($H_\bullet$)**.**
Let $H := f \geq e$ be a singleton hypothesis, and let $\mathcal{X}$ be an automaton. Let $x_1, x_2, \ldots x_n \in X$ be states of $\mathcal{X}$ such that for all $i \leq n$,

$$f^{-1}l_{\mathcal{X}}(x_i) \not\subseteq e^{-1}l_{\mathcal{X}}(x_i).$$

Now let $\mathcal{Z}_i$ be the output of Thompson's construction for the regular language:

$$[\![e]\!] \cdot f^{-1}l_{\mathcal{X}}(x_i).$$

Then we define:

$$H_\bullet(\mathcal{X}) = \mathcal{X}[x_1 \to \mathcal{Z}_1, \ldots x_n \to \mathcal{Z}_n].$$

For a more algorithmic presentation, see figure 1.

In words, given an automaton $\mathcal{X}$, we iterate through all of the states in $\mathcal{X}$ doing the procedure at the beginning of the section, and propagate the results in the automaton $\mathcal{X}'$. Importantly, we *do not* iterate through the states in $\mathcal{X}'$, so newly added states are not treated in one application of $H_\bullet$.

Another important detail is that in both line 4 and line 5, we use $l_{\mathcal{X}}(x)$, not $l_{\mathcal{X}'}(x)$. In other words, the process applies to all states in X "simultaneously", not considering how the alterations it is making might affect the languages of other states. Why define $H_\bullet$ in this way? In short, to ensure that $H_\bullet$ is well defined, producing the same output regardless of what order we iterate through the states. If we do not do this, the order that the states are visited *does* matter, as the following example will show.

Algorithm 1 $H_\bullet$ is defined by the function AutCloseSingle.

```
1: function AutCloseSingle(f ≥ e, 𝒳)
2:     𝒳' ← 𝒳
3:     for all x ∈ X do
4:         if f^{-1} l_𝒳(x) ⋢ e^{-1} l_𝒳(x) then
5:             𝒵 ← regToAut([[e]] · f^{-1} l_𝒳(x))
6:             𝒳' ← 𝒳'[x → 𝒵]
7:         end if
8:     end for
9:     return 𝒳'
10: end function
```

Example 4.10.
Let $H := \mathtt{ba} \geq \mathtt{a}$, and consider the following automaton:

$$\mathcal{X} := \qquad x_1 \xrightarrow{\mathtt{b}} x_2 \xrightarrow{\mathtt{b}} x_3 \xrightarrow{\mathtt{a}} \circledcirc$$

We now calculate $H_\bullet(\mathcal{X})$, but will at each step use the new language obtained in the prior step, contrary to how we defined the construction above. Now we have to decide, *which state do we look at first?*

Suppose we visit x_2 first. Then we calculate that it accepts the language $\{\mathtt{ba}\}$. So we have the Brzozowski derivatives:

$$\begin{aligned} f^{-1} l_{\mathcal{X}}(x_2) =& (\mathtt{ba})^{-1}\{\mathtt{ba}\} = \{\varepsilon\} \\ e^{-1} l_{\mathcal{X}}(x_2) =& \ (\mathtt{a})^{-1}\{\mathtt{ba}\} = \emptyset \end{aligned}$$

Since $\{\varepsilon\} \not\subseteq \emptyset$, we will paste an automaton at x_2. Namely, we have to paste on the automaton for the regular expression $\mathtt{a} \cdot \varepsilon = \mathtt{a}$, which we call $\mathcal{Z}_\mathtt{a}$:

$$\mathcal{X}[x_2 \to \mathcal{Z}_\mathtt{a}] = \qquad \begin{array}{l} x_1 \xrightarrow{\mathtt{b}} x_2 \xrightarrow{\mathtt{b}} x_3 \xrightarrow{\mathtt{a}} \circledcirc \\ \qquad\qquad\ \ \varepsilon \big\downarrow \\ \qquad\qquad\ \ \bullet \xrightarrow[\mathtt{a}]{} \circledcirc \end{array} \tag{4.4}$$

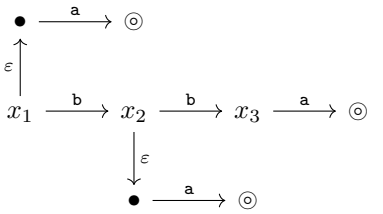
Now we move on in 4.4, say we look to x_1 next. We calculate that it accepts the language $\{\mathtt{bba}, \mathtt{ba}\}$, giving the Brzozowski derivatives:

$$\begin{aligned} (\mathtt{ba})^{-1}\{\mathtt{bba}, \mathtt{ba}\} &= \{\varepsilon\} \\ (\mathtt{a})^{-1}\{\mathtt{bba}, \mathtt{ba}\} &= \emptyset \end{aligned}$$

as was the case for x_2, $\{\varepsilon\} \not\subseteq \emptyset$, so we have to operate on x_1 as well. In fact, we

have to paste on the same automaton, $\mathcal{Z}_{\mathtt{a}}$:

$$\mathcal{X}[x_2 \to \mathcal{Z}_{\mathtt{a}}, x_1 \to \mathcal{Z}_{\mathtt{a}}] = \tag{4.5}$$



Here lies the problem: if we had chosen to visit x_1 first, we would have found its language to be $\{\mathtt{bba}\}$ (not $\{\mathtt{bba}, \mathtt{ba}\}$), and so we would have found that:

$$(\mathtt{ba})^{-1}\{\mathtt{bba}\} = \emptyset$$
$$(\mathtt{a})^{-1}\{\mathtt{bba}\} = \emptyset$$

In this case, we would have done nothing to x_1. This means that depending on if we had looked at x_2 or x_1 first, we have two potential output automata; (4.5), with an automaton pasted onto x_1, and (4.4), without.

We circumvent this problem by always considering the language of x_1 *as it was* before we did anything; specifically, by using $l_{\mathcal{X}}(x_1)$ rather than $l_{\mathcal{X}'}(x_1)$ as we did in 4.9. Therefore, the output for $H_\bullet$ as we have defined it would be (4.4).

We now intend to use $H_\bullet$ as the step for a closure operation, similar to how we defined H on languages. In order for this to be meaningful, we first prove that this function is inflationary.

Lemma 4.11.
$H_\bullet$ is inflationary with respect to the order $\sqsubseteq$ on automata (def. 2.25). By definition, this means it is also inflationary on languages.

Proof. Let $\mathcal{X}$ be an automaton. Then we want to show that $\mathcal{X} \sqsubseteq H_\bullet(\mathcal{X})$. By lemma 2.26, it is sufficient to show that:

- $X_{\mathcal{X}} \subseteq X_{\mathcal{Y}}$
- $\forall x \in X_{\mathcal{X}}, \forall w \in \Sigma^*,\ \delta_{\mathcal{X}}(x, w) \subseteq \delta_{\mathcal{Y}}(x, w)$
- $\nu_{\mathcal{X}} \subseteq \nu_{\mathcal{Y}}$

Let X be the state set of $\mathcal{X}$, and we say that $\overline{X}$ is the state set for $H_\bullet(\mathcal{X})$.

We break down an application of $H_\bullet$: there is some subset of X which are *the states that are acted on*, those states that have an automaton pasted to them. We let $x_1, x_2, \ldots x_n$ be these states, and $\mathcal{Z}_1, \mathcal{Z}_2, \ldots \mathcal{Z}_n$ be the automata that are pasted onto them. Recall from lemma 4.8 that pasting is inflationary.

So we reason:

$$\begin{aligned}\mathcal{X} &\sqsubseteq \mathcal{X}[x_1 \to \mathcal{Z}_1] \\ &\sqsubseteq \mathcal{X}[x_1 \to \mathcal{Z}_1, x_2 \to \mathcal{Z}_2] \\ &\dots \\ &\sqsubseteq \mathcal{X}[x_1 \to \mathcal{Z}_1, x_2 \to \mathcal{Z}_2, \dots x_n \to \mathcal{Z}_n] = H_\bullet(\mathcal{X})\end{aligned}$$

Lemma 4.8 also proved that the order of $x_1, \dots x_n$ is unimportant. So $\mathcal{X} \sqsubseteq H_\bullet(\mathcal{X})$, as desired. 完

$H_\bullet$ is monotone as well, but the proof is quite involved, and the result will follow nicely from one of our main results. See corollary 4.17 for details.

Now that we know $H_\bullet$ is inflationary, we can use it to define our desired closure operation. We define it in a way that should be reminiscent of our definition of H^* for languages:

Definition 4.12 ($H^*_\bullet$)**.**
Let H be a singleton hypothesis and $\mathcal{X}$ an automaton such that there exists some finite stabilisation point of $H_\bullet$ on $\mathcal{X}$; that is, some $n \in \mathbb{N}$ such that:

$$H^n_\bullet(\mathcal{X}) = H^{n+1}_\bullet(\mathcal{X}).$$

Then the *automaton singleton hypothesis closure* $H^*_\bullet$ is defined:

$$H^*_\bullet(\mathcal{X}) := H^n_\bullet(\mathcal{X}).$$

In situations where H is a singleton hypothesis and $H^*_\bullet$ is defined, we will say that $H^*_\bullet$ *terminated* on $\mathcal{X}$.

Equivalently, the automaton singleton hypothesis closure is the least automaton (with respect to $\sqsubseteq$) $\overline{\mathcal{X}}$ such that $\mathcal{X} \sqsubseteq \overline{\mathcal{X}}$ and $H_\bullet(\overline{\mathcal{X}}) = \overline{\mathcal{X}}$, if such an automaton exists.

There is a key difference between this definition and that for hypothesis closure of languages: we can not use Knaster-Tarski to guarantee it exists by phrasing it as a least fixed point. While (as we stated above) $H_\bullet$ is indeed monotone, the lattice of finite automata is not complete: we can very easily obtain non-finite (infinite) automata as the union of some set of finite automata, for example an infinite disjoint union of singleton automata. Knaster-Tarski only applies to complete lattices, so it cannot be used here. Indeed, $H^*_\bullet$ often does not terminate, and should not do so: see example 5.3 and section 5.1 for more details.

The fact that termination of $H^*_\bullet$ is not guaranteed reflects an important fact about hypothesis closure, namely that it does not always produce a regular language/finite automaton. The hypothesis closure operation produces is a language, but not necessarily a regular one. Similarly, the closure operation on automata will produce an automaton, but not necessarily a finite one. As we do

not concern ourselves with infinite automata in this thesis, we simply defined the operation partially.

With the closure operation defined, we still need to show that it *works*: in the situations where it terminates, does it produce an automaton that is correct according to the goal we set in section 3.5? Namely, does $H_\bullet^*$ correspond in a meaningful way to the hypothesis closure operation on languages, H^*? As a preview of that proof, the objective will be to show that applications of $H_\bullet$ to the automaton correspond to applications of H to the language.

Before moving to the proof, we work out some examples and justify some decisions made in the definition of $H_\bullet$.

Example 4.13.
The reader might wonder why we reapply the construction to all states in the automaton at every step, rather than only applying to the newly added states. As it stands, we are likely doing a lot of redundant work. The reason for that can already be seen in example 4.10. We recall the automaton used here:

$$\mathcal{X} := \qquad x_1 \xrightarrow{\texttt{b}} x_2 \xrightarrow{\texttt{b}} x_3 \xrightarrow{\texttt{a}} \circledcirc$$

Let $H := \texttt{ba} \geq \texttt{a}$, and we now calculate successive applications of $H_\bullet$; see example 4.10 for details on how they were calculated.

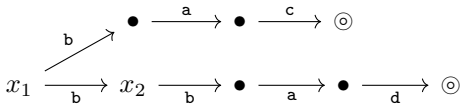
$$H_\bullet(\mathcal{X}) = \qquad \begin{array}{ccccccc} x_1 & \xrightarrow{\texttt{b}} & x_2 & \xrightarrow{\texttt{b}} & x_3 & \xrightarrow{\texttt{a}} & \circledcirc \\ & & \downarrow{\scriptstyle \varepsilon} & & & & \\ & & \bullet & \xrightarrow[\texttt{a}]{} & \circledcirc & & \end{array}$$

$$H_\bullet^2(\mathcal{X}) = \qquad \begin{array}{ccccccc} & & \bullet & \xrightarrow{\texttt{a}} & \circledcirc & & \\ & & \uparrow{\scriptstyle \varepsilon} & & & & \\ x_1 & \xrightarrow{\texttt{b}} & x_2 & \xrightarrow{\texttt{b}} & x_3 & \xrightarrow{\texttt{a}} & \circledcirc \\ & & \downarrow{\scriptstyle \varepsilon} & & & & \\ & & \bullet & \xrightarrow[\texttt{a}]{} & \circledcirc & & \end{array}$$

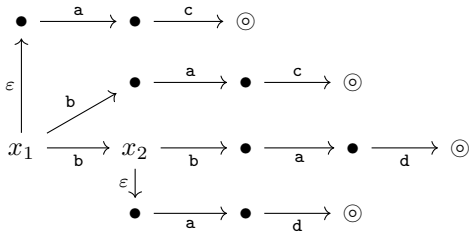
Notice that in the second application of $H_\bullet$, we needed to paste an automaton to the state x_1; despite the fact that it was already examined in the prior application, and that it was determined then that it did not need to have an automaton pasted to it. Therefore, we needed to check it again in this step, and in general, we *cannot* assume that a state will not need an automaton pasted to it simply because it was checked in a prior application of $H_\bullet$.

Example 4.14.
The reader may *also* wonder if, to solve the problem of "re-checking" states we have already examined, we could keep track of which states have had an automaton pasted to them, and so not have to check them again. For example, in the automaton $\mathcal{X}$ given above in example 4.13, we did need to check x_1 again in the second application of $H_\bullet$, but we did not need to check x_2, as it already had an automaton pasted to it. Unfortunately, this does not work either. We

can construct a counterexample by just "stacking" instances of the hypothesis used in example 4.13.

$\mathcal{X} :=$ 

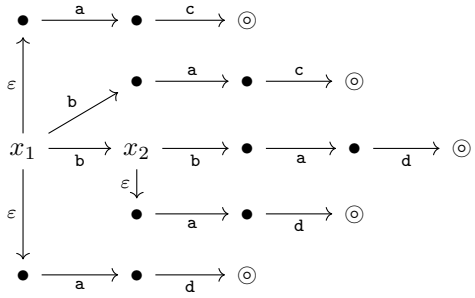
Examining each state, we see that both x_1 and x_2 need an automaton pasted to them, because they both accept a word beginning with `ba`. Pasting on the appropriate automata:

$H_\bullet(\mathcal{X}) =$ 

We see that both states have an automaton pasted to them; we might at this moment say that they are both "marked" and so no longer need to be checked. Unfortunately, this is not the case either. The language accepted by x_1 is now:

$$l_{H_\bullet(\mathcal{X})}(x_1) = \{\texttt{ac}, \texttt{bac}, \texttt{bbad}, \texttt{bad}\}$$

note that this language contains `bad`, but does not contain `ad`. Pasting on the appropriate automaton finally gives us the correct output automaton:

$H^*_\bullet(\mathcal{X}) = H^2_\bullet(\mathcal{X}) =$ 

We now proceed to the correctness proof of $H^*_\bullet$.

4.4 Correctness proof

The construction $H^*_\bullet$ is a way to "close" an automaton with respect to a singleton hypothesis. Two important questions remain, however:

- In what cases will $H^*_\bullet$ terminate?

▶ When it does terminate, does it produce a correct automaton?

For this section, we will focus on the latter question; discussion of the first will come in chapter 5. To show that $H^*_\bullet$ produces a "correct" automaton, we will show a correspondence between it and the closure operation H^* defined for languages. In other words, for all $x \in X$:

$$H^*(l_{\mathcal{X}}(x)) = l_{H^*_\bullet(\mathcal{X})}(x). \tag{4.6}$$

Both the language closure and automaton closure are obtained by a possibly infinite number of applications of an intermediary function, H and $H_\bullet$ respectively. We will prove an intermediary result, and the desired correspondence (equation 4.6) will follow as a consequence. We make a stronger claim, that not only are the end results of these closures the same, but they are also the same at every intermediary step.

Theorem 4.15 (Singleton Correspondence)**.**
Let H be a singleton hypothesis, and $\mathcal{X}$ an automaton. Then for all $x \in X$:

$$H(l_{\mathcal{X}}(x)) = l_{H_\bullet(\mathcal{X})}(x).$$

Naturally, the function H on languages will feature heavily in this proof. We refer the reader back to definition 3.10 for the details of its definition.

Proof. We will approach this proof by showing that each language is a subset of the other. We begin with the "$\subseteq$" direction.

Let $w \in H(l_{\mathcal{X}}(x))$. If $w \in l_{\mathcal{X}}(x)$, then $w \in l_{H_\bullet(\mathcal{X})}(x)$ as well, because $H_\bullet$ is inflationary on languages in its automaton, as discussed in lemma 4.11. So we assume that w was newly added in the application of H, that is $w \notin l_{\mathcal{X}}(x)$. Because $H := f \geq e$, according to the definition of H on languages, $w = uw_ev$ where $w_e \in [\![e]\!]$, and u, v are words such that $uw_fv \in l_{\mathcal{X}}(x)$ for some $w_f \in [\![f]\!]$. However, because H is a singleton hypothesis, we will use w_f to refer to the singular word in $[\![f]\!]$.

Therefore, there must be an $\mathcal{X}$ trace for the word uw_fv that runs from x to an accepting state z. We can decompose such a trace as follows:

$$x \xrightarrow{u} y_1 \xrightarrow{w_f} y_2 \xrightarrow{v} z \tag{4.7}$$

Now examining the sub-trace running from y_1 to z, we see that this is an accepting $\mathcal{X}$ trace for w_fv and so:

$$\begin{aligned} & w_fv \in l_{\mathcal{X}}(y_1) \\ & \Rightarrow v \in w_f^{-1} l_{\mathcal{X}}(y_1) \end{aligned}$$

At the same time, because $w = uw_ev \notin l_{\mathcal{X}}(x)$, we can conclude:

$$\begin{aligned} & w_ev \notin l_{\mathcal{X}}(y_1) \\ & \Rightarrow v \notin w_e^{-1} l_{\mathcal{X}}(y_1) \\ & \Rightarrow v \notin e^{-1} l_{\mathcal{X}}(y_1) \\ & \Rightarrow f^{-1} l_{\mathcal{X}}(y_1) = w_f^{-1} l_{\mathcal{X}}(y_1) \not\subseteq e^{-1} l_{\mathcal{X}}(y_1) \end{aligned}$$

Recall that this is precisely the condition for which $H_\bullet(-)$ will paste in an automaton at y_1, if we run it on $\mathcal{X}$. Formally, we can say that:

$$\mathcal{X} \;\sqsubseteq\; \mathcal{X}[y_1 \to \mathcal{Z}] \;\sqsubseteq\; H_\bullet(\mathcal{X})$$

where $\mathcal{Z}$ is the output of Thompson's construction for the regular expression $[\![e]\!] \cdot w_f^{-1} l_{\mathcal{X}}(x)$. By the definition of automaton pasting, we know that the following is an $\mathcal{X}[y_1 \to \mathcal{Z}]$ trace:

$$y_1 \xrightarrow{\varepsilon} \bullet \xrightarrow{w_e} \bullet \xrightarrow{v} z$$

for any word $w_e \in [\![e]\!]$. Since $w_e \in [\![e]\!]$, we have that:

$$\begin{aligned} & w_e v \in l_{\mathcal{X}[y_1 \to \mathcal{Z}]}(y_1) \\ \Rightarrow\; & u w_e v \in l_{\mathcal{X}[y_1 \to \mathcal{Z}]}(x) \quad \text{(see trace 4.7)} \\ \Rightarrow\; & u w_e v \in l_{H_\bullet(\mathcal{X})}(x) \end{aligned}$$

where the last line follows from the fact that $\mathcal{X}[y_1 \to \mathcal{Z}] \sqsubseteq H_\bullet(\mathcal{X})$. From this we conclude that $w \in l_{H_\bullet(\mathcal{X})}(x)$, as desired.

Now we proceed to the "$\supseteq$" direction. Let $w \in l_{H_\bullet(\mathcal{X})}(x)$. Similarly to the other direction, if $w \in l_{\mathcal{X}}(x)$, then $w \in H(l_{\mathcal{X}}(x))$ immediately because H is inflationary (lem. 4.11). So we assume that $w \notin l_{\mathcal{X}}(x)$, so w is not accepted by x in $\mathcal{X}$. But, it is accepted by x in $H_\bullet(\mathcal{X})$. This means that all accepting traces for w must traverse some transition(s) added by $H_\bullet$, and that there is at least one such trace.

Recall from lemma 4.8 that if a trace visits a state in a pasted-on automaton, it must also end in the pasted-on automaton; because the transitions added by $H_\bullet$ all *end* in pasted-on automata, we know that all accepting traces for w must visit a state newly added by $H_\bullet$, and so also end in a newly added state.

Fixing some accepting trace of w on $\mathcal{X}$, we know it must end at a state z that was added by $H_\bullet$, meaning that it is an accepting state in one of the automata that was pasted onto $\mathcal{X}$. Let $\mathcal{Z}$ be the automaton that z is in, and $y \in X$ be the state onto which $\mathcal{Z}$ was pasted. Then by the definition of $H_\bullet$:

$$\mathcal{X} \;\sqsubseteq\; \mathcal{X}[y \to \mathcal{Z}] \;\sqsubseteq\; H_\bullet(\mathcal{X})$$

where $(\mathcal{Z}, z_0)$ is the output of Thompson's construction for $[\![e]\!] \cdot w_f^{-1} l_{\mathcal{X}}(y)$. So we can dissect the trace, breaking down the word w such that $w = uw'$ and:

$$x \xrightarrow{u} y \xrightarrow{\varepsilon} z_0 \xrightarrow{w'} z$$

is an $X[y \to \mathcal{Z}]$ trace. In particular, $w' \in l_{\mathcal{Z}}(z_0)$, and so by lemma 2.13, it must be that $w' \in [\![e]\!] \cdot w_f^{-1} l_{\mathcal{X}}(y)$. Therefore, we can decompose the word $w' = w_e v$ such that $w_e \in [\![e]\!]$ and:

$$\begin{aligned} & v \in (w_f)^{-1} l_{\mathcal{X}}(y) \\ \Rightarrow\; & w_f v \in l_{\mathcal{X}}(y) \\ \Rightarrow\; & u w_f v \in l_{\mathcal{X}}(x) \\ \Rightarrow\; & u[\![e]\!] v \subseteq H(l_{\mathcal{X}}(x)) \end{aligned}$$

and because $w_e \in [\![e]\!]$, we conclude that $w = uw_ev \in H(l_{\mathcal{X}}(x))$. 完

With the theorem proven, we can now use it to reach our ultimate goal: a correspondence between the closure operations on language and automata, H^* and $H^*_\bullet$.

Corollary 4.16.
Let H be a singleton hypothesis, $\mathcal{X}$ an automaton with $x \in X$ such that $H^*_\bullet$ terminates on $\mathcal{X}$. Then:

$$H^*(l_{\mathcal{X}}(x)) = l_{H^*_\bullet(\mathcal{X})}(x).$$

Proof. $H^*_\bullet$ terminates on $\mathcal{X}$, meaning that there is some $n \in \mathbb{N}$ such that:

$$H^n_\bullet(\mathcal{X}) = H^*_\bullet(\mathcal{X}).$$

By theorem 4.15, we know that

$$H(l_{\mathcal{X}}(x)) = l_{H_\bullet(\mathcal{X})}(x).$$

So it is straightforward to also conclude:

$$\begin{aligned} H^n(l_{\mathcal{X}}(x)) &= H^{n-1}(l_{H_\bullet(\mathcal{X})}(x)) \\ &= H^{n-2}(l_{H^2_\bullet(\mathcal{X})}(x)) \\ &= \cdots \\ &= l_{H^n_\bullet(\mathcal{X})}(x) \end{aligned}$$ 完

It was mentioned in section 4.3 that it is possible to prove that $H_\bullet$ is monotone directly from definition, but it involves a fair amount of bookkeeping, and was not necessary as we did not use Knaster-Tarski to define $H^*_\bullet$. Nonetheless, it follows nicely a result of theorem 4.15.

Corollary 4.17.
Let H be a singleton hypothesis; then $H_\bullet$ is monotone.

Proof. Let $\mathcal{X} = (X, \delta, \nu)$ and $\mathcal{Y} = (Y, \delta', \nu')$ be automata such that $\mathcal{X} \sqsubseteq \mathcal{Y}$. Then we want to show that $H_\bullet(\mathcal{X}) \sqsubseteq H_\bullet(\mathcal{Y})$. Now for all $x \in X$:

$$\begin{aligned} & l_{\mathcal{X}}(x) \subseteq l_{\mathcal{Y}}(x) \quad \text{(def. 2.25)} \\ \Rightarrow & H(l_{\mathcal{X}}(x)) \subseteq H(l_{\mathcal{Y}}(x)) \quad \text{(prop. 3.8)} \\ \Rightarrow & l_{H_\bullet(\mathcal{X})}(x) \subseteq l_{H_\bullet(\mathcal{Y})}(x) \quad \text{(thm. 4.15)} \\ \Rightarrow & H_\bullet(\mathcal{X}) \sqsubseteq H_\bullet(\mathcal{Y}) \quad \text{(def. 2.25)} \end{aligned}$$ 完

As discussed in section 3.5, there are multiple criteria by which we could judge that $H^*_\bullet$ succeeded, in that it produced the correct automaton. We revisit one such criterion now, showing it follows from our existing result:

Corollary 4.18.
Fix an automaton $\mathcal{X}$, and a singleton hypothesis $H := f \geq e$ (with $[\![f]\!] = \{w_f\}$) such that $H^*_\bullet$ terminates on $\mathcal{X}$. Then:

$$\forall x \in H^*_\bullet(\mathcal{X}),\ H(l_{H^*_\bullet(\mathcal{X})}(x)) = l_{H^*_\bullet(\mathcal{X})}(x).$$

that is, *all languages in a hypothesis closed automaton are hypothesis closed.*

Proof. Since $H^*_\bullet$ terminated on $\mathcal{X}$, we know there is some finite $n \in \mathbb{N}$ such that:

$$H^n_\bullet(\mathcal{X}) = H^*_\bullet(\mathcal{X}).$$

Therefore, our objective is to show that, for every $x \in H^*_\bullet(\mathcal{X})$,

$$H(l_{H^n_\bullet(\mathcal{X})}(x)) = l_{H^n_\bullet(\mathcal{X})}(x). \tag{4.8}$$

This is quite straightforward to see, invoking theorem 4.15 and definition 4.12:

$$H(l_{H^n_\bullet(\mathcal{X})}(x)) = l_{H^{n+1}_\bullet(\mathcal{X})}(x)) = l_{H^n_\bullet(\mathcal{X})}(x). \qquad \text{完}$$

In this chapter we have covered the construction $H_\bullet$ and its closure operation $H^*_\bullet$, as well as correctness proofs in the cases where it terminates. In the next chapter, we will discuss what termination cases are, why they are desirable, and what we can do to improve the construction to create more such cases.

Chapter 5

Termination

In this section we discuss some corollaries of theorem 4.15, show how it can be used for decidability and completeness proofs of Kleene algebra with hypotheses in the case that it terminates, and present some strategies for growing the class of hypotheses for which it can terminate.

5.1 Applications

Recall that the entire purpose of using Kleene algebra as a starting point was to leverage its decidability (theorem 2.6) and completeness (theorem 2.22), and enhance its expressiveness without losing those nice properties. Now that we have proposed the construction $H^*_\bullet$ for automata, we discuss how to leverage those properties to prove decidability and completeness, if we can show that $H^*_\bullet$ terminates. These ideas are common throughout literature (for example in [5] section 4), although they are not always explicitly stated. We incorporate them here so as to form a coherent picture of how $H^*_\bullet$ can be effectively used.

Proposition 5.1.
Let e, f be regular expressions, and let H be a singleton hypothesis such that $H^*_\bullet$ terminates for all input automata. Then it is decidable whether

$$[\![e]\!]_H = [\![f]\!]_H.$$

Proof. Note that by definition 3.13:

$$[\![e]\!]_H := H^*([\![e]\!]),$$

Now using Thompson's construction we can obtain an automaton $\mathcal{X}_e$ with an initial state x_e such that:

$$l_{\mathcal{X}_e}(x_e) = [\![e]\!].$$

By assumption $H^*_\bullet$ terminates for $\mathcal{X}_e$, so by theorem 4.15 we have that:

$$H^*([\![e]\!]) = H^*(l_{\mathcal{X}_e}(x_e)) = l_{H^*_\bullet(\mathcal{X}_e)}(x_e)$$

and by Kleene's theorem we can create a regular expression $\bar{e}$ such that $[\![\bar{e}]\!] = l_{H^*_\bullet(\mathcal{X}_e)}(x_e)$, and therefore:

$$[\![\bar{e}]\!] = H^*([\![e]\!]).$$

Of course we can repeat this whole process to obtain a regular expression $\bar{f}$ with a similar result. Now, because Kleene algebra is decidable (thm. 2.6) we can decide whether:

$$[\![e]\!]_H = H^*([\![e]\!]) = [\![\bar{e}]\!] \stackrel{?}{=} [\![\bar{f}]\!] = H^*([\![f]\!]) = [\![f]\!]_H$$

so we are done. 完

Note that, because KA' is simply KA with extra axioms, the proof of $\bar{e} = \bar{f}$ is also valid in KA'.

Proposition 5.2.
Given any regular expression r, let $\bar{r}$ be its "hypothesis closed" regular expression as defined in the proof of theorem 5.1. Let e, f be regular expressions, and let H be a singleton hypothesis such that $H^*_\bullet$ terminates for all input automata. Now assume that $\mathsf{KA} + H \vdash r = \bar{r}$ for every r. Then $\mathsf{KA} + H$ is complete.

Proof. Suppose that $[\![e]\!]_H = [\![f]\!]_H$. Then by completeness of KA (thm 2.22) and our reasoning in the proof of proposition 5.1, we will have that:

$$\mathsf{KA} \vdash \bar{e} = \bar{f}.$$

Note that because $\mathsf{KA} + H$ is simply KA with more axioms, any proof valid in KA is also valid in $\mathsf{KA} + H$. As a result, we can use the KA proof of $\bar{e} = \bar{f}$ in $\mathsf{KA} + H$ as well:

$$\mathsf{KA} + H \vdash \bar{e} = \bar{f}.$$

On the other hand, we have assumed that:

$$\mathsf{KA} + H \vdash \bar{e} = e \qquad \mathsf{KA} + H \vdash \bar{f} = f.$$

So by transitivity, we have that:

$$\mathsf{KA} + H \vdash e = f$$

as desired. 完

With these results, it becomes clear why having that $H^*_\bullet$ terminates for all input automata is a desirable situation.

Recall that in remark 4.3 we stated that we do not expect $H^*_\bullet$ to terminate in general. This remains true, and theorem 4.15 illuminates why—we also do not expect the language hypothesis function to stabilise in finitely many steps in a great deal of cases.

It is important to say: there are many cases in which $H^*_\bullet$ *should not* terminate. If H is a hypothesis that does not preserve regularity of the input language, then we would not expect to be able to produce a finite automaton accepting it, because if a language is non-regular, then there is no such finite automaton.

Example 5.3.
Let $H := \mathtt{b} \geq \mathtt{aba}$, and let $L := \{\mathtt{b}\}$. Then we can observe:

$$\begin{aligned} H^0(L) &= \{\mathtt{b}\} \\ H^1(L) &= \{\mathtt{b}, \mathtt{aba}\} \\ H^2(L) &= \{\mathtt{b}, \mathtt{aba}, \mathtt{aabaa}\} \\ &\cdots \\ H^*(L) &= \{\mathtt{a}^n\mathtt{ba}^n | n \in \mathbb{N}\} \end{aligned}$$

$\mathtt{a}^n\mathtt{ba}^n$ is well-known to be a non-regular language; there is no finite automaton that accepts it. So $H^*_\bullet$ will not terminate on any automaton with a state accepting the language $\{\mathtt{b}\}$, because that would produce a finite automaton accepting a non-regular language.

Theorem 4.15 showed that one application of H on a language is equivalent to one application of $H_\bullet$ to automata. We can use this correspondence to relate the conditions under which they will stabilise:

Corollary 5.4.
Let $\mathcal{X}$ be an automaton, and $H := f \geq e$ a singleton hypothesis such that $H^*_\bullet$ terminates on $\mathcal{X}$. Then the following are equivalent:

1. for every $x \in X$, there exists an $n_x \in \mathbb{N}$ such that:

$$H^*(l_\mathcal{X}(x)) = H^{n_x}(l_\mathcal{X}(x))$$

2. there exists an $n \in \mathbb{N}$ such that:

$$H^*_\bullet(\mathcal{X}) = H^n_\bullet(\mathcal{X})$$

Intuitively, this makes sense: each of the states in $\mathcal{X}$ has a language that will be closed after some number of steps. There are finitely many, so we just take the highest of these, and apply $H_\bullet$ that many times. On the other hand, since we proved in corollary 4.18 that all languages in a hypothesis-closed automaton are closed, we know they were all closed after at most as many times as $H_\bullet$ was applied.

Proof. For the direction of $1 \Rightarrow 2$, assume that 1 holds: for every $x \in X$ there is an $n_x \in \mathbb{N}$ such that $H^*(l_\mathcal{X}(x)) = H^{n_x}(l_\mathcal{X}(x))$. $\mathcal{X}$ is a finite automaton, so X is a finite set, meaning there are finitely many n_x; we let n be the largest one. We now claim that:

$$H_\bullet(H^n_\bullet(\mathcal{X})) = H^n_\bullet(\mathcal{X}) \quad (\star)$$

from which we could conclude that:

$$H^*_\bullet(\mathcal{X}) = H^n_\bullet(\mathcal{X}).$$

To prove ($\star$), we observe that the languages of all states in $\mathcal{X}$ are hypothesis-closed, because for every x, $n_x \leq n$ and H has been applied to each language n times. Formally, for every $x \in X$:

$$\begin{aligned} H(H^n(l_{\mathcal{X}}(x))) &= H(H^{n-n_x}(H^{n_x}(l_{\mathcal{X}}(x)))) \\ &= H(H^{n-n_x}(H^*(l_{\mathcal{X}}(x)))) \\ &= H^{n-n_x}(H(H^*(l_{\mathcal{X}}(x)))) \\ &= H^{n-n_x}(H^*(l_{\mathcal{X}}(x))) \\ &= H^{n-n_x}(H^{n_x}(l_{\mathcal{X}}(x))) \\ &= H^n(l_{\mathcal{X}}(x)) \end{aligned} \tag{5.1}$$

The question now is, is this sufficient to say that $H^*_\bullet$ has terminated? To check, we want to know if applying $H_\bullet$ will do anything to any state; if not, then it has indeed terminated. Recalling the condition from definition 4.9, we want to show that for every $x \in H^n_\bullet(\mathcal{X})$,

$$f^{-1} l_{H^n_\bullet(\mathcal{X})}(x) \subseteq e^{-1} l_{H^n_\bullet(\mathcal{X})}(x).$$

Note that since f is assumed to be a singleton hypothesis, we have that $[\![f]\!] = \{w_f\}$ for some $w_f \in \Sigma^*$. We rephrase the goal:

$$(w_f)^{-1} l_{H^n_\bullet(\mathcal{X})}(x) \subseteq (w_e)^{-1} l_{H^n_\bullet(\mathcal{X})}(x), \text{ for every } w_e \in [\![e]\!].$$

Let $w \in (w_f)^{-1} l_{H^n_\bullet(\mathcal{X})}(x)$, and let $w_e \in [\![e]\!]$ arbitrary. Then:

$$\begin{aligned} & w_f w \in l_{H^n_\bullet(\mathcal{X})}(x) && \text{(def. of Brzozowski derivative, 2.15)} \\ \Rightarrow & w_f w \in H^n(l_{\mathcal{X}}(x)) && \text{(thm. 4.15)} \\ \Rightarrow & w_e w \in H\big(H^n(l_{\mathcal{X}}(x))\big) && \text{(def. of } H\text{, 3.5)} \\ \Rightarrow & w_e w \in H^n(l_{\mathcal{X}}(x)) && \text{(eq. 5.1)} \\ \Rightarrow & w \in (w_e)^{-1} H^n(l_{\mathcal{X}}(x)) && \text{(def. of Brzozowski derivative, 2.15)} \\ \Rightarrow & w \in (w_e)^{-1} l_{H^n_\bullet(\mathcal{X})}(x) && \text{(thm 4.15)} \end{aligned}$$

Therefore, for each state $x \in X$, $H_\bullet$ will not paste an automaton to it, meaning that $H^*_\bullet$ has terminated, as desired.

As for the $2 \Rightarrow 1$ direction, if $H^*_\bullet$ terminates in n steps, we know that for all $x \in H^*_\bullet(\mathcal{X}) = H^n_\bullet(\mathcal{X})$:

$$H^*(l_{H^n_\bullet(\mathcal{X})}(x)) = l_{H^n_\bullet(\mathcal{X})}(x)$$

by corollary 4.18. In words, that in the hypothesis-closed automaton, the languages of all the states are also hypothesis-closed. We also know by theorem 4.15 that:

$$l_{H^n_\bullet(\mathcal{X})}(x) = H^n(l_{\mathcal{X}}(x))$$

so for each $x \in H^*_\bullet(\mathcal{X})$, we let $n_x := n$, and we are done. 完

We have remarked throughout the thesis that we do often do not expect $H^*_\bullet$ to terminate, and that in general it *should not*. We already saw one such example in example 5.3: a hypothesis that, when applied to a regular language, can produce a non-regular language. Here we define an important notion for discussing when $H^*_\bullet$ will terminate: whether the hypothesis in question preserves regularity, when used for closure.

Definition 5.5.
Let H be a hypothesis. If for all regular languages L, $H(L)$ is also a regular language, we say that H *preserves regularity*.

When defining the hypothesis closure operation for languages, it was not of interest if, or when, it would terminate: the definition just used the stabilisation point of H, when applied to the language L. The correspondence established by 4.15, however, tells us that we can use termination of the H^* on languages to study termination of $H^*_\bullet$ on automata. We see now how this might be of interest: if we know for a hypothesis H that $H^*_\bullet$ will always terminate, then we know that H preserves regularity.

Remark 5.6.
Unfortunately, deciding whether a hypothesis preserves regularity is quite non-trivial: in fact, in general, we believe it to be undecidable. The most likely avenue of proof would use Greibach's theorem, a result stating that it is undecidable whether a context-free grammar generates a regular language. One can easily construct a set of hypotheses to mimic a context-free grammar, such that the closure under those hypotheses is the sentential form of the grammar. One would then have to prove that the sentential forms of a grammar are regular if and only if the context-free language of that grammar are regular, and also that the hypothesis preserves regularity if and only if the sentential forms are regular. Both are reasonable claims, but fall outside the scope of this thesis.

Nonetheless, we briefly sketch the conversion from grammar to hypotheses. Let G be an arbitrary context-free grammar, with V the set of non-terminal symbols, $S \in V$ the initial symbol, and P be our set of production rules. We now define a set of hypotheses for the alphabet $V \cup \Sigma$:

$$\{A \geq w : A \to w \text{ is a rule in } P\}.$$

Now if we take the language $\{S\}$ and close it under this set of hypotheses, we obtain exactly the sentential forms of G.

It is still a significant avenue for future research to find classes of hypotheses that can be guaranteed to preserve regularity. Indeed, there are already some results in this direction: in 2014, Kozen and Mamouras [10] showed that a particular sub-class of hypotheses can be thought of as *inverse context-free rewrite systems*, which are known to preserve regularity.

Remark 5.7.
Throughout the thesis up to this point, it has largely been left vague when we

want $H_\bullet^*$ to terminate. Mostly, it has just been said that we would like it to terminate "as often as possible". We have also observed that $H_\bullet^*$ will clearly not terminate when the hypothesis closure does not preserve regularity (example 5.3).

This raises the question of what the precise goal *should* be. We stated earlier we set the goal that $H_\bullet^*$ should terminate if and only if the hypothesis H preserves regularity. It is not clear if this is possible, in the sense that there may be hypotheses that preserve regularity, but somehow cannot be captured by a finite construction on automata. See 5.14 for more discussion.

However, there are also some situations where we would like $H_\bullet^*$ to terminate, but it does not.

Example 5.8.
Consider the following automaton:

$$\mathcal{X} := \qquad x \xrightarrow{\mathtt{a}} \circledcirc$$

If we take the hypothesis $H := \mathtt{a} \geq \mathtt{aa}$, we can deduce in a fairly straightforward way the hypothesis closure of $l_{\mathcal{X}}(x)$:

$$H^*(l_{\mathcal{X}}(x)) = H^*(\{\mathtt{a}\}) = \{\mathtt{a}^n : n > 0\} = [\![\mathtt{aa}^*]\!]$$

We can also observe that a finite solution to the automaton closure problem (with no regard for how $H_\bullet$ works) exists:

$$\mathcal{X}' := \qquad x \xrightarrow{\mathtt{a}} \circledcirc \quad \text{with a loop } \mathtt{a} \text{ on } \circledcirc$$

Unfortunately, however, $H_\bullet^*$ will proceed infinitely, never stabilising, as we will now see. Looking at the languages of our states (and the relevant Brzozowski derivatives):

$$f^{-1}l_{\mathcal{X}}(x) = (\mathtt{a})^{-1}\{\mathtt{a}\} = \{\varepsilon\} \qquad f^{-1}l_{\mathcal{X}}(\circledcirc) = (\mathtt{a})^{-1}\{\varepsilon\} = \emptyset$$
$$e^{-1}l_{\mathcal{X}}(x) = (\mathtt{aa})^{-1}\{\mathtt{a}\} = \emptyset \qquad e^{-1}l_{\mathcal{X}}(\circledcirc) = (\mathtt{aa})^{-1}\{\varepsilon\} = \emptyset$$

From this we conclude that, since $\{\varepsilon\} \not\subseteq \emptyset$, we need to operate on x, but we do not need to do anything for $\circledcirc$. So we paste the automaton representing the language $\mathtt{aa} \cdot \varepsilon = \mathtt{aa}$ on at x:

$$H_\bullet(\mathcal{X}) = \qquad x \xrightarrow{\mathtt{a}} \circledcirc, \quad x \xrightarrow{\varepsilon} x_1 \xrightarrow{\mathtt{a}} x_2 \xrightarrow{\mathtt{a}} \circledcirc$$

The language of both $\circledcirc$ states is empty, so we know we will not have to operate on those. Unfortunately, x still meets the condition to have an automaton

pasted onto it:

$$\begin{aligned} l_{H_\bullet(\mathcal{X})}(x) &= \{\mathtt{a}, \mathtt{aa}\} \\ f^{-1}l_{H_\bullet(\mathcal{X})}(x) &= (\mathtt{a})^{-1}\{\mathtt{a}, \mathtt{aa}\} = \{\varepsilon, \mathtt{a}\} \\ e^{-1}l_{H_\bullet(\mathcal{X})}(x) &= (\mathtt{aa})^{-1}\{\mathtt{a}, \mathtt{aa}\} = \{\varepsilon\} \end{aligned}$$

and we can extrapolate to further applications of $H_\bullet$:

$$\begin{aligned} l_{H^2_\bullet(\mathcal{X})}(x) &= \{\mathtt{a}, \mathtt{aa}, \mathtt{aaa}\} \\ f^{-1}l_{H^2_\bullet(\mathcal{X})}(x) &= (\mathtt{a})^{-1}\{\mathtt{a}, \mathtt{aa}, \mathtt{aaa}\} = \{\varepsilon, \mathtt{a}, \mathtt{aa}\} \\ e^{-1}l_{H^2_\bullet(\mathcal{X})}(x) &= (\mathtt{aa})^{-1}\{\mathtt{a}, \mathtt{aa}, \mathtt{aaa}\} = \{\varepsilon, \mathtt{a}\} \end{aligned}$$

No matter how many times we apply $H_\bullet$, we will keep needing to operate on x, and new states (x_1, x_2 so far) will need to be operated on as well.

As an aside, the fact that $H_\bullet$ will proceed without terminating on this automaton also follows from theorem 4.15, by instead examining the hypothesis language function H applied to the language $\{\mathtt{a}\}$; we can observe that $H^n := \{\mathtt{a}^m : m \leq n\}$ which will never stabilise.

The state x initially only accepts $\mathtt{a}$, and the desired outcome is that it accepts $\mathtt{aa}^*$, which is obviously regular. But as we can see, the construction operating only on one application of the hypothesis at a time does not have the "foresight" to know we could achieve the same goal in one step, by just adding a loop.

As such, we will look at a way to modify $H^*_\bullet$ so that there are more hypotheses and automata for which it will terminate.

5.2 Saturation

We saw in example 5.8 a situation where we would like $H^*_\bullet$ to terminate, but it does not. It is also not hard to imagine more examples of a similar type. If applying the hypothesis to a language adds new words that themselves allow us to apply the hypothesis again, the construction will proceed infinitely. This is the case with $\mathtt{a} \geq \mathtt{aa}$: given a word of the form $u\mathtt{a}v$, we add the word $u\mathtt{aa}v$, from which we can produce $u\mathtt{aaa}v$, and then $u\mathtt{aaaa}v$, and so on, and so on...

Example 5.9.
In example 5.8 we took our hypothesis to be $\mathtt{a} \geq \mathtt{aa}$. Imagine we had instead taken our hypothesis to be $\mathtt{a} \geq \mathtt{aaa}^*$. Then we can calculate:

$$H^*(\{\mathtt{a}\}) = \{\mathtt{a}^n | n > 0\} = [\![\mathtt{aa}^*]\!]$$

So $\mathtt{a} \geq \mathtt{aaa}^*$ is another hypothesis that is equivalent to $\mathtt{a} \geq \mathtt{aa}$ in that they produce the same language hypothesis closure, meaning that a solution to one is a solution to the other so far as automaton closure is concerned. However, we can observe that closing the automaton from example 5.8 under $\mathtt{a} \geq \mathtt{aaa}^*$ actually terminates. Recall that the automaton was:

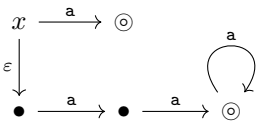
$$\mathcal{X} := \qquad x \xrightarrow{\mathtt{a}} \circledcirc$$

In our first application, we find that $l_{\mathcal{X}}(\circledcirc) = \{\varepsilon\}$ so we do nothing to it, and we find that:

$$f^{-1}l_{\mathcal{X}}(x) = (\mathtt{a})^{-1}\{\mathtt{a}\} = \{\varepsilon\}$$
$$e^{-1}l_{\mathcal{X}}(x) = (\mathtt{aaa}^*)^{-1}\{\mathtt{a}\} = \emptyset$$

So we paste the automaton representing $\mathtt{aaa}^* \cdot \varepsilon = \mathtt{aaa}^*$ onto x:

$H_\bullet(\mathcal{X}) =$



while not literally identical to the automaton we were aiming for, we now can examine each of the states and find there is nothing to do: if a state accepts $\mathtt{a}$, it can accept up to infinitely many.

We briefly remark on why this is a worthwhile thing to do. Recall from propositions 5.1 and 5.2 that guaranteeing $H^*_\bullet$ will terminate for any automaton gives decidability of equality for $[\![-]\!]_H$, and is a meaningful step in the direction of completeness of $\mathsf{KA} + H$ with respect to $[\![-]\!]_H$. So any situation for which we can show that $H^*_\bullet$ will terminate is a significant result for the system $\mathsf{KA} + H$.

When presented with a hypothesis, we would like to find a way that we can find a corresponding equivalent hypothesis for which the automaton closure will terminate, in general. What is the relationship between the hypothesis $\mathtt{a} \geq \mathtt{aa}$ and the improved version we found, $\mathtt{a} \geq \mathtt{aaa}^*$? Observe that if we let $H := \mathtt{a} \geq \mathtt{aa}$,

$$[\![\mathtt{aaa}^*]\!] = H^*([\![\mathtt{aa}]\!])$$

that is, $\mathtt{aaa}^*$ is the closure of H's conclusion under H itself!

Definition 5.10.
For a hypothesis $H := f \geq e$, the *saturation* of H is defined $f \geq \overline{e}$, where $\overline{e}$ is a regular expression representing the (regular) language $H^*([\![e]\!])$. We will often refer to the saturation of a hypotheses H as $\overline{H}$.

We say that a hypothesis $f \geq e$ is *saturated* if $H^*([\![e]\!]) = [\![e]\!]$.

We saw (in example 5.9) the hypothesis $\mathtt{a} \geq \mathtt{aa}$, if it exists, which can be represented by a construction on automata in a fairly straightforward way, but for whom our proposed construction $H^*_\bullet$ will generally not terminate. Saturation can be a useful tool for finding another "equivalent" hypothesis for which $H^*_\bullet$ will be able to terminate. In order for this strategy to be useful, however, we need to specify what is meant by "equivalent" and show that the newly acquired hypothesis will be equivalent to the old one.

Theorem 5.11 (Saturation Invariance)**.**
Let H be a singleton hypothesis with saturation $\overline{H}$, and let L be a regular language. Then:

$$H^*(L) = \overline{H}^*(L). \tag{5.2}$$

Proof. Let $H = f \geq e$ be our hypothesis, and $\overline{H} = f \geq \bar{e}$ the saturation of H, where $\bar{e}$ is such that:

$$[\![\bar{e}]\!] = [\![e]\!]_H.$$

First, we set out to prove equation 5.2. For the "$\subseteq$" direction, note first that by definition:

$$[\![e]\!] \subseteq H^*([\![e]\!]) = [\![e]\!]_H = [\![\bar{e}]\!]. \tag{5.3}$$

Now we simply claim that for every L, $H(L) \subseteq \overline{H}(L)$. Recall the definition of H (def. 3.5) here:

$$H(L) := L \cup \{uw_e v : \text{ such that } w_e \in [\![e]\!], \text{ and } u[\![f]\!]v \subseteq L\}.$$

We also write the definition when we use $\overline{H}$ as our hypothesis:

$$\overline{H}(L) := L \cup \{uw_e v : \text{ such that } w_e \in [\![\bar{e}]\!], \text{ and } u[\![f]\!]v \subseteq L\}.$$

Suppose that $w \in H(L)$. If $w \in L$ then by definition $w \in \overline{H}(L)$. If not, then $w = uw_e v$, such that $w_e \in [\![e]\!]$ and u, v are such that $u[\![f]\!]v \subseteq L$. Now by (5.3), $[\![e]\!] \subseteq [\![\bar{e}]\!]$ and so $w_e \in [\![\bar{e}]\!]$ as well. Therefore, all of the conditions are met for us to say that $w = uw_e v \in \overline{H}(L)$.

As for the "$\supseteq$" direction, let $w \in \overline{H}^*(L)$. We observe that there is an $n \in \mathbb{N}$ such that $w \in \overline{H}^n(L)$. Our objective will be to replace each application of $\overline{H}$ with a finite (but probably large) number of applications of H. To do this, we claim that if $w \in \overline{H}(L)$, there exists some $m \in \mathbb{N}$ such that:

$$w \in H^m(L) \subseteq H^*(L).$$

So let $w \in \overline{H}(L)$. If $w \in L$, then as usual we are done because $\overline{H}$ is inflationary by definition. So we assume $w \notin L$, so $w = uw_{\bar{e}}v$ such that:

1. $w_{\bar{e}} \in [\![\bar{e}]\!]$
2. $u \cdot [\![f]\!] \cdot v \subseteq L$

Now recall that by definition:

$$[\![\bar{e}]\!] = [\![e]\!]_H = H^*([\![e]\!]).$$

Using item (1), we observe that there is some $k \in \mathbb{N}$ such that $w_{\bar{e}} \in H^k([\![e]\!])$. Meanwhile item (2) tells us that $u \cdot [\![e]\!] \cdot v \subseteq H(L)$. Therefore:

$$\begin{aligned} & u \cdot [\![e]\!] \cdot v \subseteq H(L) \\ \Rightarrow\ & H^k(u \cdot [\![e]\!] \cdot v) \subseteq H^{k+1}(L). \end{aligned}$$

We can now apply lemma 3.11 to say that:

$$u \cdot H([\![e]\!]) \cdot v \subseteq H(u \cdot [\![e]\!] \cdot v)$$

and combining these claims, we have that:

$$u \cdot H^k([\![e]\!]) \cdot v \subseteq H^k(u \cdot [\![e]\!] \cdot v) \subseteq H^{k+1}(L).$$

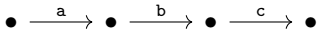
Now because $w_{\bar{e}} \in H^k([\![e]\!])$, we at last have that $w = uw_{\bar{e}}v \in H^{k+1}(L)$, as desired. 完

Remark 5.12.
An observant reader may notice that employing the strategy of saturating hypotheses before closing under them is circular, in a sense. One of our objectives is to find a way to calculate the hypothesis closure of a regular expression in terms of another regular expression. However, in order to saturate a hypothesis $f \geq e$, we need to be able to calculate the hypothesis closure of e as a regular expression $\bar{e}$.

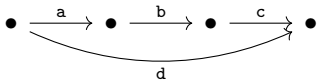
This is still a reduction in complexity, however. In relevant situations where saturation can guarantee termination of $H^*_\bullet$ (such as $H := \mathtt{a} \geq \mathtt{aa}$), we have reduced the problem of computing the closure by H of an arbitrary regular expression to that of computing the closure by H of the expression $\mathtt{aa}$.

Intuition might lead us to believe that saturating hypotheses can guarantee termination, maybe even in one step as was the case in example 5.9. Sadly, this is not the case. We present one counterexample here:

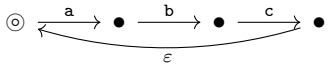
Example 5.13.
Let $H := \mathtt{abc} \geq \mathtt{d}$. We can observe informally that, similar to contraction, this hypothesis can be realised on automata by seeking out the following situation in the automaton:



and adding a transition:



This is with no regard for the actual construction $H^*_\bullet$; we are observing that our desired goal (automaton closure) can be achieved in a fairly straightforward way, similar to the other hypotheses given in example 3.16. We now compare this to how the given construction, $H^*_\bullet$, fares. Observe that H is saturated: closing $\mathtt{d}$ under H just gives $\mathtt{d}$. Therefore, we are closing under a saturated hypothesis. Now consider the following automaton:

$\mathcal{X} :=$ 

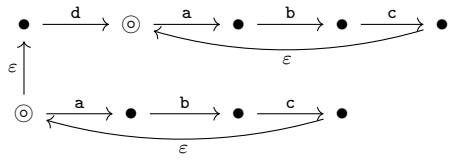
The state ◎ accepts the language $(\mathtt{abc})^*$; let's set the goal by finding the closure

of this language:

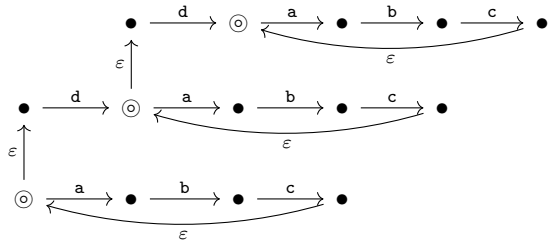
$$\begin{aligned}
[\![(\mathtt{abc})^*)]\!] =& \{\varepsilon, \mathtt{abc}, \mathtt{abcabc}, \mathtt{abcabcabc}, \ldots\} \\
H^*([\![(\mathtt{abc})^*]\!]) =& \{\varepsilon, \mathtt{abc}, \mathtt{abcabc}, \ldots \\
& \quad \mathtt{d}, \mathtt{dabc}, \mathtt{dabcabc}, \ldots \\
& \quad\quad \mathtt{dd}, \mathtt{ddabc}, \ldots\} \\
=& [\![(\mathtt{abc} + \mathtt{d})^*]\!]
\end{aligned}$$

However, as we try to close the corresponding automaton...

$H_\bullet(\mathcal{X}) =$



$H^2_\bullet(\mathcal{X}) =$



we can extrapolate and see that at each step, the newly added ◎ state will need to be operated, and so $H^*_\bullet$ will proceed without terminating.

Remark 5.14.
The fact that $H^*_\bullet$ does not terminate for the simple hypothesis $\mathtt{abc} \geq \mathtt{d}$ (as seen in example 5.13) is unfortunate, particularly because we do not study any further means of improving it in this thesis. This means that we must officially abandon the goal mentioned in 5.7.

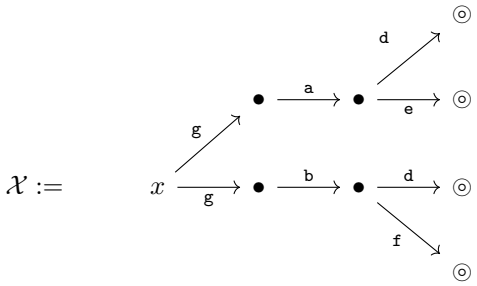
We now move on to the generalised version of $H^*_\bullet$, for non-singleton hypotheses.

Chapter 6

General Hypothesis Closure, H

Now that we have seen a simpler version of the construction for simpler hypotheses, we move on to the case where hypotheses $f \geq e$ can have both f, e as arbitrary regular expressions. As a starting point, we should ask: why is the construction we have just defined *not sufficient*?

Example 6.1.
Let $H := \mathtt{a} + \mathtt{b} \geq \mathtt{c}$. Note that H is *not* a singleton hypothesis, since $[\![\mathtt{a} + \mathtt{b}]\!] = \{\mathtt{a}, \mathtt{b}\}$. Now consider the following automaton:



We now deduce that:

$$l_{\mathcal{X}}(x) = \{\mathtt{gad}, \mathtt{gae}, \mathtt{gbd}, \mathtt{gbf}\}$$

Recall from the definition of the language hypothesis function (definition 3.5) that in order to calculate $H(l_{\mathcal{X}}(x))$, we have to find all pairs of words $u, v \in \Sigma^*$ such that:

> Both $u \cdot \mathtt{a} \cdot v$ and $u \cdot \mathtt{b} \cdot v$ are in $l_{\mathcal{X}}(x)$.

By observation, we see that the only such pair is $\mathtt{g}, \mathtt{d}$. Therefore,

$$H(l_{\mathcal{X}}(x)) = \{\mathtt{gad}, \mathtt{gae}, \mathtt{gbd}, \mathtt{gbf}, \mathtt{gcd}\}.$$

If we look again, we do not find any prefix/suffix pairs that 6.1 holds; so we conclude that:

$$H^*(l_{\mathcal{X}}(x)) = \{\mathtt{gad}, \mathtt{gae}, \mathtt{gbd}, \mathtt{gbf}, \mathtt{gcd}\}.$$

However, according to $H_\bullet$, there is nothing to do for this automaton. Recall that we are checking, for each state $y \in X$:

$$\begin{aligned} f^{-1}l_{\mathcal{X}}(y) &\overset{?}{\subseteq} e^{-1}l_{\mathcal{X}}(y) \\ f^{-1}l_{\mathcal{X}}(y) &= \bigcap_{w \in [\![f]\!]} w^{-1}l_{\mathcal{X}}(y) \\ &= (\mathtt{a} + \mathtt{b})^{-1}l_{\mathcal{X}}(y) \\ &= (\mathtt{a})^{-1}l_{\mathcal{X}}(y) \cap (\mathtt{b})^{-1}l_{\mathcal{X}}(y) \\ e^{-1}l_{\mathcal{X}}(y) &= (\mathtt{c})^{-1}l_{\mathcal{X}}(y) \end{aligned} \tag{6.1}$$

It is straightforward to check that for all of the states, both of these derivatives are empty; meaning that:

$$H_\bullet(\mathcal{X}) = \mathcal{X} \quad \Rightarrow \quad H_\bullet^*(\mathcal{X}) = \mathcal{X}.$$

As we can see, this violates the condition we set for $H_\bullet$'s correctness:

$$l_{H_\bullet^*(\mathcal{X})}(x) = l_{\mathcal{X}}(x) \neq \{\mathtt{gad}, \mathtt{gae}, \mathtt{gbd}, \mathtt{gbf}, \mathtt{gcd}\} = H^*(l_{\mathcal{X}}(x)).$$

So our construction $H_\bullet$ will not be sufficient for this situation. The reason is that the states from which the `a` and `b` transitions originate are separate (despite being reached using the same prefix), and $H_\bullet$ will not operate on either. We would like for it to operate on x, but for it to do that, it would have to adopt the prefix `g` as part of the Brzozowski derivative it uses to decide if it will operate on a state or not.

The reason that this is a problem now, and not with singleton hypotheses, can be seen in (6.1) above. In the case where H is a singleton hypothesis, $[\![f]\!]$ is a singleton, and so the intersection in $f^{-1}l_{\mathcal{X}}(y)$ will always be just one state. With us allowing $[\![f]\!]$ to have more than one word in it, this intersection can be non-trivial.

A further reason for the problem is that we are dealing with non-deterministic automata; we could of course require determinism, or even just determinise before starting, but every application of the construction introduces (by design) a great deal of non-determinism. One could determinise after every step, but that would make the continuity between states much harder to track as the construction proceeded.

In this section, we will define and verify a construction $H : \mathsf{NA} \rightharpoonup \mathsf{NA}$ on automata, for general hypotheses. We will define H using $H_\bullet$ as a foundation, adding machinery to deal with the problem of prefixes discussed above.

6.1 Bounded Prefixes with $\sim_{\mathcal{X},x}$

We have established that the construction has to somehow account for arbitrary prefix words when closing a state in an automaton. However, words are of unbounded length: if we naively check all possible prefixes, our construction will never terminate. Therefore, we need a way to bound the words that are necessary to check. To this end, we will define an equivalence relation:

Definition 6.2.
Given an automaton $\mathcal{X}$ and a state $x \in X$, we define an equivalence relation $\sim_{\mathcal{X},x} \subseteq \Sigma^* \times \Sigma^*$ on words as follows:

$$w \sim_{\mathcal{X},x} w' \quad \Leftrightarrow \quad \delta(x, w) = \delta(x, w')$$

It is straightforward to observe that $\sim_{\mathcal{X},x}$ is an equivalence relation, because it is based on the equality of $\delta(x, -)$. Intuitively, two words are equivalent under this relation if the set of states they can reach from x in $\mathcal{X}$ is the same. The objective will be to show that if two words are equivalent in this way, then we can regard them as equivalent with respect to the construction; and that there are only finitely many equivalence classes to check.

Lemma 6.3.
Let $\mathcal{X}$ be an automaton, with $x \in X$, and let $w \in \Sigma^*$ be a word.

1. $[w]_{\mathcal{X},x}$ is a regular language.
2. $[w]_{\mathcal{X},x}$ contains a word w' such that $|w'| \leq 2^{|X|}$.
3. if $w \sim_{\mathcal{X},x} w'$, then $w \in l_{\mathcal{X}}(x)$ if and only if $w' \in l_{\mathcal{X}}(x)$.
4. if $w \sim_{\mathcal{X},x} w'$, $w^{-1} l_{\mathcal{X}}(x) = (w')^{-1} l_{\mathcal{X}}(x)$

Proof. Let $\mathcal{X}$ be an automaton, $x \in X$, and $w \in \Sigma^*$. Let $\mathcal{X}'$, with state set X', be the result of determinising $\mathcal{X}$ according to the power set construction, covered in theorem 2.11. Note that this means that $X' = \mathcal{P}(X)$.

1. All sets of states in X correspond to a single state in X', so in particular $\delta(x, w)$ corresponds to a state in X'. Alter $\mathcal{X}'$ so that this state is the only accepting state. We claim the state $\{x\}$ in this (finite) automaton accepts exactly $[w]_{\mathcal{X},x}$, and if that is indeed that case, by definition it is a regular language.

   Let $w' \in [w]_{\mathcal{X},x}$. Then by definition, $\delta'(x, w')$ will be the state corresponding to $\delta(x, w') \subseteq X$. Since $\delta(x, w') = \delta(x, w)$, this is exactly the one accepting state of $\mathcal{X}'$, so we are done.

2. Let δ and δ' be the respective transition functions of $\mathcal{X}$, $\mathcal{X}'$. We want to show that there is some word $w^\flat$ of length at most $2^{|X|}$ such that $\delta(x, w^\flat) = \delta(x, w)$. Then since $\mathcal{X}'$ has one state for each set of states in $\mathcal{X}$, $|X'| = 2^{|X|}$.

Now because $\mathcal{X}'$ is a deterministic automaton, we know that $\delta'(x, w)$ is a single state, $x_w \in X'$. Again by determinism, we now that there is *exactly* one $\mathcal{X}'$ trace for w, going from x to x_w. We reasoned that $\mathcal{X}'$ has $2^{|X|}$ states, so this trace cannot visit more than that many *unique* states in this trace.

Of course if $|w| \leq 2^{|X|}$ we are done, so assuming that $|w| > 2^{|X|}$, we can conclude that the trace must visit more than $2^{|X|}$ many states, meaning that it visit some state more than once. In other words, we can dissect the trace:

$$x \xrightarrow{w_0} x' \xrightarrow{w_1} x' \xrightarrow{w_2} x_w$$

where $|w_1| > 0$. Note that this means that $|w_0 w_2| < |w|$. If $|w_0 w_2| \leq 2^{|X|}$, then we are done, otherwise we repeat this argument again until we obtain a word of length less than $2^{|X|}$, which we call $w^\flat$. Then we have:

$$\delta'(x, w^\flat) = x_w = \delta'(x, w) \quad \Rightarrow \quad \delta(x, w^\flat) = \delta(x, w) \quad \Rightarrow \quad w^\flat \sim_{\mathcal{X},x} w.$$

3. We know by definition that $w \in l_{\mathcal{X}}(x)$ if and only if there is some accepting state $z \in X$ such that $z \in \delta(x, w)$. Of course because $\delta(x, w) = \delta(x, w')$, we also know that $z \in \delta(x, w')$, and so $w' \in l_{\mathcal{X}}(x)$ as well.

4. We can prove this claim by simple manipulation:

$$\begin{aligned} w^{-1} l_{\mathcal{X}}(x) &= \{v \in \Sigma^* : wv \in l_{\mathcal{X}}(x)\} \\ &= \{v \in \Sigma^* : \delta(x, wv) \cap \nu \neq \emptyset\} \\ &= \{v \in \Sigma^* : \delta(x, w'v) \cap \nu \neq \emptyset\} \\ &= \{v \in \Sigma^* : w'v \in l_{\mathcal{X}}(x)\} \\ &= (w')^{-1} l_{\mathcal{X}}(x) \end{aligned}$$ 完

Remark 6.4.
This equivalence relation is, in effect, the same one that is used in the Myhill-Nerode theorem. While that relation is defined for words on deterministic automaton, we would obtain the same thing by determinising $\mathcal{X}$ first. We do not explore this relationship here, but mention it because the fact that $\sim_{\mathcal{X},x}$ has finitely many equivalence classes may be familiar to the reader that has studied automaton theory.

The reason that we are interested in this equivalence relation, and its equivalence classes, is that we need a way to bound one step of the construction. The language closure definition uses arbitrary words u as prefixes; naively, we would need to attempt to paste an automaton for every word, which would obviously never terminate.

Instead of doing that, we claim it will be sufficient to just iterate through the equivalence classes of $\sim_{\mathcal{X},x}$. We claim that doing so is sufficient to obtain an equivalent result, we have a constructive way of doing so, and it will always terminate. To show this, we need to prove some facts about the equivalence relation:

Lemma 6.5.
Given a state x in an automaton $\mathcal{X}$, the following holds for every $w \in \Sigma^*$:

$$w^{-1}l_{\mathcal{X}}(x) \subseteq ([w]_{\mathcal{X},x})^{-1}l_{\mathcal{X}}(x)$$

Proof. Let $u \in w^{-1}l_{\mathcal{X}}(x)$ for some $w, u, l_{\mathcal{X}}(x)$. By definition, $wu \in l_{\mathcal{X}}(x)$. We wish to show that $u \in [w]^{-1}_{\mathcal{X},x}l_{\mathcal{X}}(x)$, recalling from definition 4.4 that:

$$[w]^{-1}_{\mathcal{X},x}l_{\mathcal{X}}(x) := \bigcap_{v \in [w]_{\mathcal{X},x}} v^{-1}l_{\mathcal{X}}(x).$$

So to show that u is in this set, we have to show that $u \in v^{-1}l_{\mathcal{X}}(x)$ for every $v \in [w]_{\mathcal{X},x}$. Let $v \in [w]_{\mathcal{X},x}$ arbitrary. By definition,

$$\delta(x, v) = \delta(x, w).$$

Since $wu \in l_{\mathcal{X}}(x)$, take an accepting trace, and let y be the state such that $x \xrightarrow{w} y \xrightarrow{u} \circledcirc$. Note that $y \in \delta(x, w) = \delta(x, v)$, so the following:

$$x \xrightarrow{v} y \xrightarrow{u} \circledcirc.$$

is an accepting $\mathcal{X}$ trace for vu. By definition, we can conclude:

$$\begin{aligned} vu &\in l_{\mathcal{X}}(x) \\ u &\in v^{-1}l_{\mathcal{X}}(x) \\ u &\in [w]^{-1}_{\mathcal{X},x}l_{\mathcal{X}}(x) \end{aligned}$$

完

6.2 Definitions of H, H^*

We now set out to define the construction $H^* : \mathsf{NA} \rightharpoonup \mathsf{NA}$ on automata, which will be a generalisation of the construction $H^*_\bullet$ proposed earlier (definition 4.12). The form of the construction will be the same: for each state we will paste automata to it so it accepts more words. However, in $H_\bullet$ we pasted either zero or one automaton to each state. This time, we will need to paste somewhere from none to $2^{|X|}$ many, for *each state*; this further highlights why we separate the general case from the singleton case: it is orders of magnitude more complex.

As before, we focus at first on the process for just one state. Take the automaton $\mathcal{X}$, and some arbitrary state $x \in X$. We now check for every word u such that $|u| \leq 2^{|X|}$:

$$([u]_{\mathcal{X},x} \cdot [\![f]\!])^{-1}l_{\mathcal{X}}(x) \overset{?}{\subseteq} ([u]_{\mathcal{X},x} \cdot [\![e]\!])^{-1}l_{\mathcal{X}}(x) \tag{6.2}$$

If this is the case, there are no words to be added to x with prefix u, and we move onto the next word. Decidability of this condition follows from lemma 2.6, by checking if:

$$([u]_{\mathcal{X},x} \cdot [\![e]\!])^{-1}l_{\mathcal{X}}(x) \setminus ([u]_{\mathcal{X},x} \cdot [\![f]\!])^{-1}l_{\mathcal{X}}(x) \overset{?}{=} \emptyset$$

If condition 6.2 does not apply, then we need to expand the language of x by some words with prefix u. Namely, we need to take words of the form $u' \cdot w_f \cdot v$ and add words of form $u' \cdot w_e \cdot v$, where u' is some word in $[u]_{\mathcal{X},x}$, $w_f \in [\![f]\!]$, and $w_e \in [\![e]\!]$. To do this we take an automaton representing the language:

$$[u]_{\mathcal{X},x} \cdot [\![e]\!] \cdot ((u \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(x))$$

and paste it onto $\mathcal{X}$ at x.

Remark 6.6.
Our decision to iterate through *every* word of length less than $2^{|X|}$ is not strictly necessary: many of those words will be in the same equivalence class with respect to $\sim_{\mathcal{X},x}$. In reality, since there can be no more than $2^{|X|}$ equivalence classes, no more than that many words need to be iterated through, in theory.

It is not the focus of this thesis to focus on such optimisations. For the time being, the main concern is that there *is* a bound at all, making the definition of H possible and guaranteed to terminate.

Now we move to the formal definition, for operating on all states at the same time. All of the same counterexamples that motivate our "simultaneous" treatment of states from the singleton case (examples 4.10, 4.13, 4.14) still apply here.

Definition 6.7.
In what follows, the set $\Sigma^*_{2^{|X|}}$ refers to the set of all words that are of length at most $2^{|X|}$. Let H be a hypothesis, and let $\mathcal{X}$ be an automaton. Let $x_1, x_2, \ldots x_n \in X$ be the states of $\mathcal{X}$ so that, for each $i \leq n$, there is a nonempty list of words $u_{i,1}, u_{i,2}, \ldots u_{i,m_i} \in \Sigma^*_{2^{|X|}}$ such that for each $j \leq m_i$:

$$([u_{i,j}]_{\mathcal{X},x_i} \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(x_i) \not\subseteq ([u_{i,j}]_{\mathcal{X},x_i} \cdot [\![e]\!])^{-1} l_{\mathcal{X}}(x_i).$$

Now let $\mathcal{Z}_{i,j}$ be the output of Thompson's construction for the regular language:

$$[u_{i,j}]_{\mathcal{X},x_i} \cdot [\![e]\!] \cdot ((u_{i,j} \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(x_i))$$

Then we define:

$$\begin{aligned} H(\mathcal{X}) = \mathcal{X}[&x_1 \to \mathcal{Z}_{1,1}, \mathcal{Z}_{1,2}, \ldots \mathcal{Z}_{1,m_1}, \\ &x_2 \to \mathcal{Z}_{2,1}, \mathcal{Z}_{2,2}, \ldots \mathcal{Z}_{2,m_2}, \\ &\cdots \\ &x_n \to \mathcal{Z}_{n,1}, \mathcal{Z}_{n,2}, \ldots \mathcal{Z}_{n,m_n}] \end{aligned}$$

This definition is quite symbol-dense, so as before, we also provide a pseudocode presentation in figure 2.

As was the case for $H_\bullet$, this construction has two main constituent parts: the condition, and the automaton to be pasted. These are naturally phrased in exactly the form they will need to be in for the correctness proof, and so it will become more apparent there why these particulars were chosen.

Algorithm 2 H is defined by the function AutClose.

```
1: function AutClose(f ≥ e, X)
2:     X' ← X
3:     for all x ∈ X do
4:         for all u ∈ Σ*_{2|X|} do
5:             if ([u]_{X,x} · [[f]])^{-1} l_X(x) ⋢ ([u]_{X,x} · [[e]])^{-1} l_X(x) then
6:                 Z ← regToAut([u]_{X,x} · [[e]] · (u · [[f]])^{-1} l_X(x))
7:                 X' ← X'[x → Z]
8:             end if
9:         end for
10:    end for
11:    return X'
12: end function
```

The main (structural) difference from the definition of $H_\bullet$ is that there is an extra iterative wrapper around our core automaton-pasting operation. Namely, for each state, we test out every word in $\Sigma^*_{2|X|}$ as a possible prefix.

As we did for $H^*_\bullet$ in definition 4.12), we will once again define the automaton closure operation H^* as the stabilisation point of the function H applied to our input automaton, if such a point exists. First, however, we will prove that H is inflationary.

Lemma 6.8.
H is inflationary on automata, with respect to the order $\sqsubseteq$. That is, for some automaton $\mathcal{X}$,

$$\mathcal{X} \sqsubseteq H(\mathcal{X}).$$

Proof. Recall from lemma 4.8 that the pasting operation is inflationary. Therefore:

$$\begin{aligned}
\mathcal{X} &\sqsubseteq \mathcal{X}[x_1 \to \mathcal{Z}_{1,1}] \\
&\sqsubseteq \mathcal{X}[x_1 \to \mathcal{Z}_{1,1}, \mathcal{Z}_{1,2}] \\
&\sqsubseteq \cdots \\
&\sqsubseteq \mathcal{X}[x_1 \to \mathcal{Z}_{1,1}, \mathcal{Z}_{1,2}, \ldots \mathcal{Z}_{1,m_1}, \\
&\qquad x_2 \to \mathcal{Z}_{2,1}, \mathcal{Z}_{2,2}, \ldots \mathcal{Z}_{2,m_2}, \\
&\qquad \cdots \\
&\qquad x_n \to \mathcal{Z}_{n,1}, \mathcal{Z}_{n,2}, \ldots \mathcal{Z}_{n,m_n}] = H(\mathcal{X})
\end{aligned}$$

完

Now we can at last present the definition of H^* on automata, which will naturally look very similar to the definition of $H^*_\bullet$ (definition 4.12).

Definition 6.9.
Let H be a hypothesis and $\mathcal{X}$ an automaton such that there exists some finite stabilisation point of H on $\mathcal{X}$; that is, some $n \in \mathbb{N}$ such that:

$$H^n(\mathcal{X}) = H^{n+1}(\mathcal{X}).$$

Then the *automaton hypothesis closure* H^* is defined:

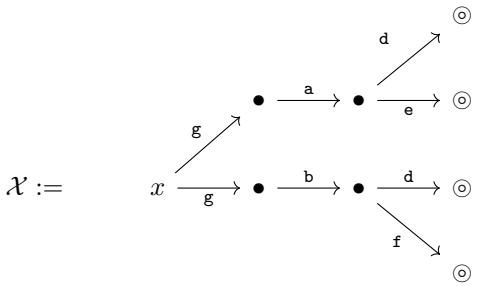
$$H^*(\mathcal{X}) := H^n(\mathcal{X}).$$

In situations where H is a hypothesis and H^* is defined, we will say that H^* *terminated* on $\mathcal{X}$.

Equivalently, the automaton hypothesis closure is the least automaton (with respect to $\sqsubseteq$) $\overline{\mathcal{X}}$ such that $\mathcal{X} \sqsubseteq \overline{\mathcal{X}}$ and $H(\overline{\mathcal{X}}) = \overline{\mathcal{X}}$, if such an automaton exists.

We now return to the problematic automaton seen in example 6.1, to see how the newly defined H fares.

Example 6.10.
Recall the automaton given in 6.1:



We adopt the hypothesis $H := \mathtt{a} + \mathtt{b} \geq \mathtt{c}$, and compute $H(\mathcal{X})$. We iterate through all of the states, seeing if there is any prefix u such that the condition:

$$([u]_{\mathcal{X},x} \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(x) \not\subseteq ([u]_{\mathcal{X},x} \cdot [\![e]\!])^{-1} l_{\mathcal{X}}(x). \tag{6.3}$$

from the definition of H (def. 6.7) is met. It is clear that for any state *other* than x, this will not be the case, because none of them have both $\mathtt{a}$ and $\mathtt{b}$ in words of their languages. While not sufficient, that is absolutely necessary for condition 6.3 to be met. On the other hand, with x, can make the choice of the prefix $\mathtt{g}$; we observe that $[\mathtt{g}]_{\mathcal{X},x} = \{\mathtt{g}\}$ and see that:

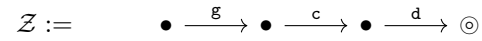
$$\begin{aligned}([u]_{\mathcal{X},x} \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(x) &= ([\mathtt{g}]_{\mathcal{X},x} \cdot \{\mathtt{a}, \mathtt{b}\})^{-1}(\{\mathtt{gad}, \mathtt{gae}, \mathtt{gbd}, \mathtt{gbf}\}) \\ &= (\{\mathtt{ga}, \mathtt{gb}\})^{-1}(\{\mathtt{gad}, \mathtt{gae}, \mathtt{gbd}, \mathtt{gbf}\}) \\ &= \{\mathtt{d}\}\end{aligned}$$

$$\begin{aligned}([u]_{\mathcal{X},x} \cdot [\![e]\!])^{-1} l_{\mathcal{X}}(x) &= ([\mathtt{g}]_{\mathcal{X},x} \cdot \{\mathtt{c}\})^{-1}(\{\mathtt{gad}, \mathtt{gae}, \mathtt{gbd}, \mathtt{gbf}\}) \\ &= \emptyset\end{aligned}$$

We can now observe that condition 6.3 is met, and so we have to paste an automaton onto x for the prefix u. We call this automaton $\mathcal{Z}$, and it is the output of Thompson's construction for the regular language:

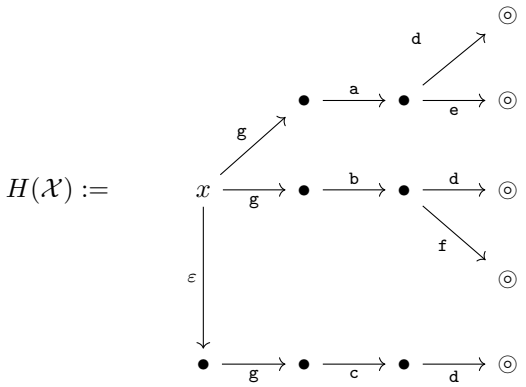
$$\begin{aligned}[u]_{\mathcal{X},x} \cdot [\![e]\!] \cdot ((u \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(x)) &= \{\mathtt{g}\} \cdot \{\mathtt{c}\} \cdot ((\mathtt{g} \cdot (\mathtt{a} + \mathtt{b}))^{-1}(\{\mathtt{gad}, \mathtt{gae}, \mathtt{gbd}, \mathtt{gbf}\}) \\ &= \{\mathtt{gcd}\}\end{aligned} \tag{6.4}$$

This automaton is quite simple:



Notice that in equation 6.4, when calculating the regular expression to create $\mathcal{Z}$, the words with `e` and `f` are ignored, because they are not invariant of every word in the assumption: `d` is in a word after both `a` and `b`, but each of `e` and `f` is only in a word after one of them. This was not something we had to worry about in the singleton hypothesis version, but here we need to ensure we are only including the suffixes that are available from every state reachable by an assumption-word (in this case `a` and `b`).

At last, we paste $\mathcal{Z}$ onto x and obtain the output automaton:



We can then look through each state of this automaton, and find that for each, there is no prefix such that condition 6.3 is met. We can similarly observe this by finding that the language of each state is hypothesis closed with respect to H; so indeed $H^*(\mathcal{X}) = H(\mathcal{X})$.

6.3 Correctness Proof

The correctness proof for this version, much like for the singleton case, will depend on showing that the intermediary function H on languages corresponds to the intermediary function H on automata.

Theorem 6.11 (General Correspondence)**.**
Let H be a hypothesis, and $\mathcal{X}$ an automaton. Then for all $x \in X$:

$$H(l_{\mathcal{X}}(x)) = l_{H(\mathcal{X})}(x).$$

Our approach will be broadly the same as it was for theorem 4.15, and we highly recommend that the reader familiarise themselves with the proof of that result before approaching this proof. Most of the focus in this proof will go to what is new, compared to that proof: dealing with arbitrary prefixes u.

Proof. We begin with the "$\subseteq$" direction. Let $w \in H(l_{\mathcal{X}}(x))$ be newly added in the application of H, because if not, then $w \in l_{\mathcal{X}}(x)$, and H is inflationary (lemma 6.8), so $w \in l_{H(\mathcal{X})}(x)$.

Now since w was newly added to the language by H, by definition 3.5, $w = uw_e v$ where $w_e \in [\![e]\!]$ and u, v are such that $u \cdot [\![f]\!] \cdot v \subseteq l_{\mathcal{X}}(x)$. Therefore:

$$\begin{aligned}
& u \cdot w_f \cdot v \in l_{\mathcal{X}}(x) \quad \text{for all } w_f \in [\![f]\!] \\
\Rightarrow\ & w_f \cdot v \in u^{-1} l_{\mathcal{X}}(x) \quad \text{for all } w_f \in [\![f]\!] \\
\Rightarrow\ & v \in (w_f)^{-1}\big(u^{-1} l_{\mathcal{X}}(x)\big) \quad \text{for all } w_f \in [\![f]\!] \\
\Rightarrow\ & v \in f^{-1}\big(u^{-1} l_{\mathcal{X}}(x)\big) \\
& \quad = (u \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(x)
\end{aligned} \tag{6.5}$$

On the other hand, since w was newly added by H, we know that:

$$\begin{aligned}
& w = u \cdot w_e \cdot v \notin l_{\mathcal{X}}(x) \\
\Rightarrow\ & \quad w_e \cdot v \notin u^{-1} l_{\mathcal{X}}(x) \\
\Rightarrow\ & \quad\quad v \notin (w_e)^{-1}\big(u^{-1} l_{\mathcal{X}}(x)\big) \\
\Rightarrow\ & \quad\quad v \notin e^{-1}\big(u^{-1} l_{\mathcal{X}}(x)\big) \\
& \quad\quad\quad = (u \cdot [\![e]\!])^{-1} l_{\mathcal{X}}(x)
\end{aligned}$$

From these two facts, we can conclude that:

$$([u]_{\mathcal{X},x} \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(x) \not\subseteq ([u]_{\mathcal{X},x} \cdot [\![e]\!])^{-1} l_{\mathcal{X}}(x). \tag{6.6}$$

This is *nearly* the condition for which H will paste an automaton onto x for the prefix u. The missing piece is that u here is not guaranteed to be of length at most $2^{|X|}$. We argue, however, that (6.6) implies the condition we desire. Due to lemma 6.3 item (2), the equivalence class $[u]_{\mathcal{X},x}$ must contain some $u^\flat$ of length at most $2^{|X|}$ such that $u \sim_{\mathcal{X},x} u^\flat$. Therefore, $[u]_{\mathcal{X},x} = [u^\flat]_{\mathcal{X},x}$, so we have:

$$([u^\flat]_{\mathcal{X},x} \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(x) \not\subseteq ([u^\flat]_{\mathcal{X},x} \cdot [\![e]\!])^{-1} l_{\mathcal{X}}(x).$$

Due to the fact that $u^\flat \in \Sigma^*_{2^{|X|}}$, this is now exactly the condition for which H will paste an automaton to onto x with prefix u. Namely, we know that:

$$\mathcal{X} \ \sqsubseteq \mathcal{X}[x \to \mathcal{Z}] \sqsubseteq H(\mathcal{X})$$

where $\mathcal{Z}$ is the pointed automaton produced by Thompson's construction for the regular expression:

$$[u^\flat]_{\mathcal{X},x} \cdot [\![e]\!] \cdot ((u^\flat \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(x)).$$

We now argue that $w \in l_{\mathcal{X}[x \to \mathcal{Z}]}(x)$, from which (by the definition of $\sqsubseteq$) we would know that $w \in l_{H(\mathcal{X})}(x)$, which is the ultimate goal. In a similar fashion to the proof of theorem 4.15, we deduce that by the definition of automaton pasting, the following is an $\mathcal{X}[x \to \mathcal{Z}]$ trace:

$$x \xrightarrow{\varepsilon} \bullet \xrightarrow{u} \bullet \xrightarrow{w_e} \bullet \xrightarrow{v} z$$

where z is an accepting state in $\mathcal{Z}$. We know this because:

- $u \sim_{\mathcal{X},x} u^\flat$
- $w_e \in [\![e]\!]$
- $v \in (u \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(x) = (u^\flat \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(x)$

where the final equality follows by lemma 6.3 item (4). Now that we have finally established the existence of the above trace, we can immediately conclude that $w = uw_e v$ has an accepting $\mathcal{X}[x \to \mathcal{Z}]$ trace, meaning that $w \in l_{\mathcal{X}[x \to \mathcal{Z}]}(x)$ and ultimately that $w \in l_{H(\mathcal{X})}(x)$, as desired.

Now for the "$\supseteq$" direction, we want to show that:

$$H(l_{\mathcal{X}}(x)) \supseteq l_{H(\mathcal{X})}(x).$$

If $w \in l_{\mathcal{X}}(x)$, similarly to the other direction we are done because H is inflationary. So let w be such that it is not accepted by x in X, but is by x in $H(\mathcal{X})$. Then we can observe that all accepting $H(\mathcal{X})$ traces for w cannot be traces in X, and so they all must traverse some transition that was newly added by H. So by lemma 4.8, we know that all accepting $H(\mathcal{X})$ traces must at some point traverse a transition leaving $\mathcal{X}$, and end at a state newly added by H.

We now fix one such trace. Let z be the accepting state that is reached, which we know is an accepting state in a pointed automaton $(\mathcal{Z}, z_0)$ that was pasted onto a state $y \in X$ for some prefix $u^\sharp$. We can now decompose this trace as follows, such that $w = u_0 u^\flat w_e v$:

$$x \xrightarrow{u_0} y \xrightarrow{\varepsilon} z_0 \xrightarrow{u^\flat w_e v} z$$

By lemma 2.13, we can also say the following about $(\mathcal{Z}, z_0)$:

$$l_{\mathcal{X}}(z_0) = [u^\sharp]_{\mathcal{X},y} \cdot [\![e]\!] \cdot ((u^\sharp \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(y)).$$

As a result, we can conclude that $w' \in [u^\sharp]_{\mathcal{X},y} \cdot [\![e]\!] \cdot ((u^\sharp \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(y))$. By definition, we can then decompose $w' = u^\flat w_e v$ such that $u^\flat \in [u^\sharp]_{\mathcal{X},y}$, $w_e \in [\![e]\!]$, and:

$$v \in (u^\sharp \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(y).$$

However, because $u^\flat \sim_{\mathcal{X},y} u^\sharp$, we can apply lemma 6.3 part (4) to find:

$$\begin{aligned}
& v \in (u^\flat \cdot [\![f]\!])^{-1} l_{\mathcal{X}}(y) \\
\Rightarrow\ & u^\flat w_f v \in l_{\mathcal{X}}(y) \quad \text{for all } w_f \in [\![f]\!] \\
\Rightarrow\ & u^\flat \cdot [\![f]\!] \cdot v \subseteq l_{\mathcal{X}}(y) \\
\Rightarrow\ & (u_0 u^\flat) \cdot [\![f]\!] \cdot v \subseteq l_{\mathcal{X}}(x) \\
\Rightarrow\ & (u_0 u^\flat) \cdot [\![e]\!] \cdot v \subseteq H(l_{\mathcal{X}}(x))
\end{aligned}$$

Therefore, since $w_e \in [\![e]\!]$, we have that:

$$w = u_0 u^\flat w_e v \in H(l_{\mathcal{X}}(x)).$$ 完

Now that we have proved correspondence of the individual steps, we can prove the correspondence between H^* on automata and H^* on languages, finally justifying the overloaded naming formally.

Corollary 6.12.
Let H be a hypothesis, $\mathcal{X}$ an automaton with $x \in X$ such that H^* terminates on $\mathcal{X}$. Then:

$$H^*(l_{\mathcal{X}}(x)) = l_{H^*(\mathcal{X})}(x).$$

Proof. H^* terminates on $\mathcal{X}$, so there exists $n \in \mathbb{N}$ such that:

$$H^n(\mathcal{X}) = H^*(\mathcal{X}).$$

By theorem 6.11:

$$H(l_{\mathcal{X}}(x)) = l_{H(\mathcal{X})}(x) \quad \text{for all } \mathcal{X}$$

and so:

$$\begin{aligned} H^n(l_{\mathcal{X}}(x)) &= H^{n-1}(l_{H(\mathcal{X})}(x)) \\ &= H^{n-2}(l_{H^2(\mathcal{X})}(x)) \\ &= \cdots \\ &= l_{H^n(\mathcal{X})}(x) \end{aligned}$$

完

6.4 Corollaries

We now can revisit and generalise many of the corollaries that were proven for the singleton case.

Corollary 6.13.
Let H be a hypothesis; then H is monotone (on automata).

Proof. Identical strategy to proof for corollary 4.17, but using theorem 6.11.
完

Corollary 6.14.
Fix an automaton $\mathcal{X}$, and a hypothesis $H := f \geq e$ such that H^* terminates on $\mathcal{X}$. Then:

$$\forall x \in H^*(\mathcal{X}),\ H(l_{H^*(\mathcal{X})}(x)) = l_{H^*(\mathcal{X})}(x).$$

that is, *all languages in a hypothesis closed automaton are hypothesis closed.*

Proof. Once again, the strategy is the same as it as in the singleton version, corollary 4.18, but using theorem 6.11 instead of theorem 4.15. 完

We also can re-prove the application of the construction we discussed in section 5.1: decidability for $\mathsf{KA} + H$, if we can show that H^* will terminate on all input automata, and completeness of $\mathsf{KA} + H$ reduced to showing it can prove a simpler form of statement.

Proposition 6.15.
Let e, f be regular expressions, and let H be a hypothesis such that H^* terminates for all input automata. Then it is decidable whether

$$[\![e]\!]_H = [\![f]\!]_H.$$

Proof. Similarly to the proof for proposition 5.1, we convert e, f to automata, apply H^* to the automata, then apply theorem 6.11 and Kleene's theorem to obtain regular expressions $\bar{e}, \bar{f}$ such that

$$[\![e]\!]_H = H^*([\![e]\!]) = [\![\bar{e}]\!]$$

and similiar for f. Then using that Kleene algebra is decidable (thm. 2.6), it is decidable whether

$$[\![\bar{e}]\!] \stackrel{?}{=} [\![\bar{f}]\!]$$

so we are done. 完

Proposition 6.16.
Let e, f be regular expressions, and let H be a hypothesis such that H^* terminates for all input automata. Additionally, given any regular expression r, let $\bar{r}$ be its "hypothesis closed" regular expression as defined in the proof of proposition 6.15, and assume that $\mathsf{KA} + H \vdash r = \bar{r}$. Then $\mathsf{KA} + H$ is complete.

Proof. Similarly to the proof of proposition 5.2, we suppose that $[\![e]\!]_H = [\![f]\!]_H$, and use completeness of KA (thm 2.22) with our reasoning in the proof of proposition 6.15 to obtain that:

- $\mathsf{KA} + H \vdash \bar{e} = \bar{f}$
- $\mathsf{KA} + H \vdash \bar{e} = e$
- $\mathsf{KA} + H \vdash \bar{f} = f$

So by transitivity, we have that:

$$\mathsf{KA} + H \vdash e = f$$

as desired. 完

Theorem 6.11 establishes the same correspondence between the language and automaton operations as theorem 4.15, and so the termination corollary we obtain (and the method used to prove it) is effectively identical.

Corollary 6.17.
Let $\mathcal{X}$ be an automaton, and H a hypothesis such that H^* terminates on $\mathcal{X}$. Then the following are equivalent:

1. for every $x \in X$, there exists an $n_x \in \mathbb{N}$ such that:

$$H^*(l_{\mathcal{X}}(x)) = H^{n_x}(l_{\mathcal{X}}(x))$$

2. there exists an $n \in \mathbb{N}$ such that:

$$H^*(\mathcal{X}) = H^n(\mathcal{X})$$

This proof is extremely similar to the proof of corollary 5.4, and so identical formal details will be given in less detail here.

Proof. For the $1 \Rightarrow 2$ direction, we assume that for every $x \in X$ there is an $n_x \in \mathbb{N}$ such that $H^*(l_{\mathcal{X}}(x)) = H^{n_x}(l_{\mathcal{X}}(x))$. $\mathcal{X}$ is a finite automaton, so X is a finite set, meaning there are finitely many n_x; we let n be the largest one. We claim:

$$H(H^n(\mathcal{X})) = H^n(\mathcal{X}).$$

We claim this is the case because, according to theorem 6.11, each application of H to $\mathcal{X}$ is one application of H to $l_{\mathcal{X}}(x)$ for each $x \in X$. Therefore, since by definition $n_x \leq n$, the language $l_{H^n(\mathcal{X})}(x)$ must be hypothesis-closed.

When H is applied to $H^n(\mathcal{X})$, it checks for each state x and prefix $u \in \Sigma_{2^{|X|}}$, if the following condition holds:

$$([u]_{H^n(\mathcal{X}),x} \cdot [\![f]\!])^{-1} l_{H^n(\mathcal{X})}(x) \not\subseteq ([u]_{H^n(\mathcal{X}),x} \cdot [\![e]\!])^{-1} l_{H^n(\mathcal{X})}(x).$$

Since $l_{H^n(\mathcal{X})}(x)$ is hypothesis-closed, it is straightforward to observe that if $v \in ([u]_{H^n(\mathcal{X}),x} \cdot [\![f]\!])^{-1} l_{H^n(\mathcal{X})}(x)$, then:

$$\begin{aligned}
u' w_f v &\in l_{H^n(\mathcal{X})}(x), \quad \forall u' \in [u]_{H^n(\mathcal{X})}, \forall w_f \in [\![f]\!] \\
u' w_e v &\in H(l_{H^n(\mathcal{X})}(x)), \quad \forall u' \in [u]_{H^n(\mathcal{X})}, \forall w_e \in [\![e]\!] \\
u' w_e v &\in l_{H^n(\mathcal{X})}(x), \quad \forall u' \in [u]_{H^n(\mathcal{X})}, \forall w_e \in [\![e]\!] \\
v &\in ([u]_{H^n(\mathcal{X}),x} \cdot [\![e]\!])^{-1} l_{H^n(\mathcal{X})}(x)
\end{aligned}$$

so the H will *not* paste an automaton onto the state x. Since x was arbitrary in $H^n(\mathcal{X})$, we know that H will do nothing to the automaton as a whole, that is:

$$H(H^n(\mathcal{X})) = H^n(\mathcal{X})$$

from which we conclude:

$$H^*(\mathcal{X}) = H^n(\mathcal{X}).$$

Now for the $2 \Rightarrow 1$ direction. If H^* terminates in n steps, then for every $x \in H^*(\mathcal{X}) = H^n(\mathcal{X})$:

$$H^*(l_{H^n(\mathcal{X})}(x)) = l_{H^n(\mathcal{X})}(x)$$

by corollary 6.14. We also know by theorem 6.11 that:

$$l_{H^n(\mathcal{X})}(x)) = H^n(l_{\mathcal{X}}(x))$$

so for each $x \in H^*(\mathcal{X})$, we let $n_x := n$, and we are done. 完

This concludes the corollaries that we extend to the general construction. Many of the remarks made regarding termination in chapter 5 remain true for the general hypothesis case, and in a similar fashion, we can study termination of H^* on automata by studying termination of H^* on languages, using theorem 6.11.

Chapter 7

Conclusion

Throughout this thesis, we have established automaton constructions that correspond to the hypothesis closure put foward by Doumane et al. in [2], and further studied in recent years [12, 5, 14]. The constructions, $H^*_\bullet$ and H^*, give a general procedure for calculating the hypothesis closure semantics of Kleene algebra extended with hypotheses. This can be used to show decidability of the relevant system (prop. 6.15), and takes steps toward showing completeness as well (prop. 6.16).

We proved that $H^*_\bullet$ and H^* produce correct automata, in case they terminate (cors. 4.16, 6.12). Termination conditions of these constructions were discussed at length in chapter 5, with the strategy of saturation (thm. 5.11) given to improve the class of hypotheses for which these constructions will be useful.

A more precise understanding of the exact hypotheses for which the constructions given will terminate remains an open problem, though it was established (in example 5.13, remark 5.14) that they do not terminate in all situations where we would like them to.

Many avenues for further work exist, particularly with regards to improving the constructions and solidifying their termination conditions. For example, the problem solved by saturation (example 5.9) can also be very effectively be solved by minimising the output automaton at every step in a particular way. This naturally breaks the correspondence to the language hypothesis function, requiring a new (equivalent) version of it be put forward, as well. Of course, this would also require that the correspondence results be re-proved.

More generally, the correspondences established in theorems 4.15 and 6.11 can be used to study properties of language and automaton closures in terms of one another. This thesis has provided techniques that, with further development, can be used as a uniform strategy for proving decidability and completeness in the context of Kleene algebra with hypotheses.

Bibliography

[1] Carolyn Jane Anderson et al. "NetKAT: semantic foundations for networks". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 113–126. URL: https://doi.org/10.1145/2535838.2535862.

[2] Amina Doumane et al. "Kleene Algebra with Hypotheses". In: *Foundations of Software Science and Computation Structures - 22nd International Conference, FOSSACS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. Ed. by Mikolaj Bojanczyk and Alex Simpson. Vol. 11425. Lecture Notes in Computer Science. Springer, 2019, pp. 207–223. URL: https://doi.org/10.1007/978-3-030-17127-8%5C_12.

[3] CAR Tony Hoare et al. "Concurrent Kleene Algebra". In: *CONCUR 2009-Concurrency Theory: 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings 20*. Springer. 2009, pp. 399–414.

[4] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. 2nd ed. Addison-Wesley, 2001. ISBN: 0-201-44124-1.

[5] Tobias Kappé et al. "Concurrent Kleene Algebra with Observations: from Hypotheses to Completeness". In: *CoRR* abs/2002.09682 (2020). arXiv: 2002.09682. URL: https://arxiv.org/abs/2002.09682.

[6] Tobias Kappé et al. "Kleene Algebra with Observations". In: *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*. Ed. by Wan J. Fokkink and Rob van Glabbeek. Vol. 140. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fur Informatik, 2019, 41:1–41:16. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2019.41.

[7] SC Kleene. "Representationof Events in Nerve Nets and Finite Automata". In: *CE Shannon and J. McCarthy* (1951).

[8] Dexter Kozen. "A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events". In: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*. IEEE Computer Society, 1991, pp. 214–225. URL: https://doi.org/10.1109/LICS.1991.151646.

[9] Dexter Kozen. "Kleene Algebra with Tests and Commutativity Conditions". In: *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 1055. Lecture Notes in Computer Science. Springer, 1996, pp. 14–33. URL: https://doi.org/10.1007/3-540-61042-1%5C_35.

[10] Dexter Kozen and Konstantinos Mamouras. "Kleene Algebra with Equations". In: *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*. Ed. by Javier Esparza et al. Vol. 8573. Lecture Notes in Computer Science. Springer, 2014, pp. 280–292. URL: https://doi.org/10.1007/978-3-662-43951-7%5C_24.

[11] Dexter Kozen and Frederick Smith. "Kleene Algebra with Tests: Completeness and Decidability". In: *Computer Science Logic, 10th International Workshop, CSL '96, Annual Conference of the EACSL, Utrecht, The Netherlands, September 21-27, 1996, Selected Papers*. Ed. by Dirk van Dalen and Marc Bezem. Vol. 1258. Lecture Notes in Computer Science. Springer, 1996, pp. 244–259. URL: https://doi.org/10.1007/3-540-63172-0%5C_43.

[12] Damien Pous, Jurriaan Rot, and Jana Wagemaker. "On Tools for Completeness of Kleene Algebra with Hypotheses". In: *CoRR* abs/2210.13020 (2022). arXiv: 2210.13020. URL: https://doi.org/10.48550/arXiv.2210.13020.

[13] Hanamantagouda P Sankappanavar and Stanley Burris. "A course in universal algebra". In: *Graduate Texts Math* 78 (1981), p. 56.

[14] Steffen Smolka et al. "Guarded Kleene algebra with tests: verification of uninterpreted programs in nearly linear time". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 61:1–61:28. URL: https://doi.org/10.1145/3371129.

[15] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications." In: *Pacific J. Math* (1955).