Bachelor Data Science and Artificial Intelligence

Completeness and Decidability of (Guarded) Kleene Algebra (with Tests) under the Hypothesis $e = 0$

Jip Raven

Supervisors:
Tobias Kappé & Liam Chung

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

01/07/2025

Abstract

Kleene Algebra (KA) is a framework for reasoning algebraically about the control flow of programs. It consists of regular expressions and axioms defining the equivalence of the expressions. Kleene Algebra with Tests (KAT) is an extension of KA that adds Boolean tests. Guarded Kleene Algebra with Tests (GKAT) is a restriction of KAT that models strictly deterministic control flow. Decidability says there exists an algorithm that determines whether two expressions are provably equivalent. The decidability of KA under the hypothesis $e = 0$ was proved by Cohen [Coh94]. This proof has been adapted to KAT but cannot easily be adapted to GKAT. We present an alternative proof for the decidability of KA under the hypothesis $e = 0$ that can more easily be adapted to GKAT. We then show that this proof strategy is not valid for GKAT.

Contents

1 Introduction

Kleene Algebra (KA) is a framework for reasoning algebraically about the control flow of programs. It consists of *regular expressions* and axioms that define the equivalence of those expressions. The expressions are made up of abstract primitive actions and the operations $+, \cdot$ and $(-)^*$. The expression $f+g$ corresponds to a non-deterministic choice between executing f or g, $f \cdot g$ corresponds to executing f followed by g and f^* corresponds to executing f zero or more times.

KA admits several important models such as formal languages [Kle56] and binary relations [Pra80]. For KA, the *completeness* of a model says that if two expressions are equal in that model then they are also provably equivalent. *Decidability* says that there exists an algorithm that determines whether two expressions are provably equivalent. The equational theory of KA is complete with respect to the language model, decidable and PSPACE-complete [Koz94]. However, reasoning under assumptions, i.e. whether a set of hypotheses entails an equation, is undecidable for arbitrary hypotheses [Kuz23]. Under some restricted forms of hypotheses, such as $e = 0$, KA remains decidable [Coh94]. In KA, the hypothesis $e = 0$ can be intuitively understood as knowing that the program e never terminates successfully. KA is also decidable under other kinds of hypotheses such as $S = 1$, where S is a sum of letters [DKPP19].

One extension of KA is Kleene Algebra with Tests (KAT), which combines KA with *Boolean Algebra*. Boolean tests can be used to model **if then else** statements, and with the combination of Boolean tests and the operation $(-)^*$, the control flow of while loops can be modelled [Koz96]. The equational theory of KAT is decidable [KS97] and PSPACE-complete [CKS96]. Furthermore, the equational theory of KAT is also decidable under the hypothesis $e = 0$ [KS97].

One use case of KAT is the verification of compiler optimisations [KP00]. We can also use KAT to reason about *Hoare triples*. A Hoare triple is a triple $\{P\}S\{Q\}$ where P and Q are assertions and S is a program. It states that if P holds, then after executing S, Q holds. In KAT Hoare triples can be encoded as hypotheses of the form $e = 0$. This allows for reasoning under the assumption of the partial correctness of a program in KAT [Koz00]. KAT has also been used to reason about packets in networks [AFG⁺14].

Smolka et al. [SFH⁺19] identify a fragment of KAT that allows for deciding program equivalence in near linear time, while still being sufficiently expressive to model typical imperative programs. This fragment is called Guarded Kleene Algebra with Tests (GKAT). GKAT replaces union $(f + g)$ and iteration (f^*) with guarded versions $(f +_b g)$ and $(f^{(b)})$ where b is a Boolean test. Here guarded means that for $f +_b g$, whether f or g is executed and for $f^{(b)}$, the amount of times that e is executed, both depend on the guard (Boolean test) b.

We can model the program **while** b **do** f; **if** b **then** g using GKAT as $f^{(b)} \cdot (g +_b 1)$. Intuitively, we know that g will never be executed and that the program should therefore be equivalent to the program **while** b **do** f that does not contain the **if** statement. This program can be modelled in GKAT as $f^{(b)}$ and the GKAT expressions $f^{(b)} \cdot (g +_b 1)$ and $f^{(b)}$ are provably equivalent using the axioms of GKAT.

The axioms model fundamental equivalences of programs. For example, the axiom $f +_b g = g +_{\bar{b}} f$ models flipping the branches of an **if**-statement by negating the condition. Concretely it models that the programs **if** b **then** f **else** g and **if not** b **then** g **else** f are equivalent.

While GKAT is complete and decidable, its axioms are not fully algebraic because one of the axioms has a side condition prohibiting arbitrary substitutions [SFH⁺19]. Kappé et al. [KSS23] provide an algebraic axiomatisation for what they call the *skip-free* fragment of GKAT and prove completeness for this fragment. Whether GKAT is also decidable under the hypothesis $e = 0$ has been left unanswered.

Cohen's proof [Coh94] has been adapted to prove the decidability of KAT under the hypothesis $e = 0$ [KS97]. A similar adaptation to GKAT is not possible. Cohen's proof works via eliminating the hypothesis by "adding" e to the expressions for which equivalence is being decided. Under the hypothesis $e = 0$, the resulting expressions are equivalent to the original expressions. Then because both expressions contain e, the hypothesis is irrelevant for their equivalence. This "addition" is done using the $+$ and cannot be adapted to the $+_b$ in GKAT. In this thesis we first provide an alternate proof for the decidability of KA under the hypothesis $e = 0$ inspired by Cohen's proof. This alternate proof works by "removing" e instead of "adding" it. As a consequence the proof does not rely on the $+$ and can therefore be adapted to GKAT. Subsequently, we attempt to adapt this proof to GKAT and show that GKAT expressions with e "removed" are not always equivalent to the original expressions. Therefore the proof cannot be adapted to GKAT.

1.1 Thesis overview

In Section 2 we give an overview of KA, GKAT and the lemmas used in our proofs. Then in Section 3 we construct a novel proof of the decidability of KA under the hypothesis $e = 0$ inspired by Cohen's proof [Coh94]. Subsequently in Section 4 we attempt to adapt this proof to GKAT and show that a crucial step does not hold there. Finally in Section 5 we discuss our results and offer suggestions for further research.

2 Definitions

This section starts by introducing KA, its axioms, the *language model* and the *relational model*. Then we introduce both deterministic and non-deterministic finite *automata*. We discuss *solutions* to automata and the connection between expressions and automata. We end the section about KA by discussing its completeness and decidability both in general and under the hypothesis $e = 0$.

Subsequently, we introduce GKAT, its axioms, its language model and its relational model. We present the automaton model for GKAT, solutions to automata and the connection with GKAT expressions. Finally, we discuss existing completeness results about GKAT and how they relate to the objective of this thesis.

2.1 Kleene Algebra

KA is a framework that allows us to represent programs in an algebraic way. More concretely it allows us to model the execution traces of programs, i.e. model all possible valid ways a program can be executed. As we will see later, expressions in KA are precisely the regular expressions.

The syntax of KA consists of expressions made up of abstract primitive actions from some set Σ, called an alphabet, and the operations $+$, $\cdot$ and $(-)^*$. Here $f + g$ means non-deterministically executing either f or g, $f \cdot g$ means first executing f then executing g and f^* means executing f zero or more times. We also have a 0 and a 1, where 0 denotes some program that immediately terminates unsuccessfully and 1 denotes a program that immediately terminates successfully without doing anything. Expressions in KA are generated by the following grammar:

$$f, g ::= 0 \mid 1 \mid a \in \Sigma \mid f + g \mid f \cdot g \mid f^*$$

As a notational convention we use a, b, c to denote actions and e, f, g, h to denote expressions. Below we formally define KA.

Definition 2.1 (Kleene Algebra [Koz94])**.** A KA is a tuple $(K, +, \cdot, (-)^*, 0, 1)$ where for all $x, y, z \in K$ the following axioms hold:

$$x + 0 = x \qquad x + x = x \qquad x + y = y + x \qquad x + (y + z) = (x + y) + z$$
$$x \cdot (y \cdot z) = (x \cdot y) \cdot z \qquad x \cdot (y + z) = x \cdot y + x \cdot z \qquad (x + y) \cdot z = x \cdot z + y \cdot z$$
$$x \cdot 1 = x = 1 \cdot x \qquad x \cdot 0 = 0 = 0 \cdot x \qquad 1 + x \cdot x^* = x^* = 1 + x^* \cdot x$$
$$x + y \cdot z \leq z \Longrightarrow y^* \cdot x \leq z \qquad x + y \cdot z \leq y \Longrightarrow x \cdot z^* \leq y$$

Here $x \leq y$ is syntactic sugar for $x + y = y$.

We use $f \equiv g$ to denote provability in KA. Therefore $f \equiv g$ says we can construct a proof using just the axioms of KA that $f = g$. Furthermore we use $f \leqq g$ as syntactic sugar for $f + g \equiv g$ and it can be proved that $\leqq$ forms a partial order on expressions. We can use these axioms to prove some basic equivalences such as the following lemma.

Lemma 2.2. *For any $f, g, h \in \mathsf{Exp}$, $f + g \leqq h$ if and only if $f \leqq h$ and $g \leqq h$.*

Proof. We first prove $f \leqq h \wedge g \leqq h \implies f + g \leqq h$.

$$\begin{aligned} f \leqq h \wedge g \leqq h &\implies f + h \equiv h \wedge g + h \equiv h && \text{definition } \leqq \\ &\implies f + (g + h) \equiv h && g + h \equiv h \\ &\implies f + g \leqq h && \text{definition } \leqq \end{aligned}$$

Now we prove $f + g \leqq h \implies g \leqq h$.

$$\begin{aligned} f + g \equiv f + g &\implies f + g + g \equiv f + g && x + x \equiv x \\ &\implies g + (f + g) \equiv f + g && x + y \equiv y + x \\ &\implies g \leqq f + g && \text{definition } \leqq \\ &\implies g \leqq h && f + g \leqq h\text{, transitivity of } \leqq \end{aligned}$$

Finally we prove $f + g \leqq h \implies f \leqq h$.

$$\begin{aligned} f + g \equiv f + g &\implies f + (f + g) \equiv f + g && x + x \equiv x \\ &\implies f \leqq f + g && \text{definition } \leqq \\ &\implies f \leqq h && f + g \leqq h\text{, transitivity of } \leqq \end{aligned}$$

■

Throughout this thesis, unless otherwise specified, when talking about the semantics of KA we will use the language model, which we will introduce shortly. In the language model expressions are interpreted as languages, which are sets of strings. A string, also called a word, is an ordered sequence of symbols from an alphabet. We use Σ^* to denote all words over the alphabet Σ and use ϵ to denote the empty word. Furthermore we use $\cdot$ to denote concatenation, where $u \cdot v$ is a string consisting of the symbols of u followed by the symbols of v. For languages we also use $\cdot$ for language concatenation, where $L_1 \cdot L_2 := \{u \cdot v \mid u \in L_1, v \in L_2\}$. Often we will drop the $\cdot$ and denote concatenation of both strings and languages simply as uv or L_1L_2. We define exponentiation for both strings and languages as repeated concatenation where: $x^{n+1} := x^n \cdot x$ and $x^0 := \epsilon$ for strings and $x^0 := \{\epsilon\}$ for languages. Finally we have an operation $(-)^*$ for languages, where $L^* := \bigcup_{n \geq 0} L^n$. Now we can formally define the language model.

Definition 2.3 (Language Model [Koz94])**.** The tuple $(2^{\Sigma^*}, \cup, \cdot, (-)^*, \emptyset, \{\epsilon\})$ is a KA and is called the language model. An interpretation of an expression in the language model is defined inductively by the function $[\![-]\!] : \mathsf{Exp} \to 2^{\Sigma^*}$ as follows:

$$[\![a]\!] := \{a\} \qquad [\![0]\!] := \emptyset \qquad [\![1]\!] := \{\epsilon\}$$

$$[\![f + g]\!] := [\![f]\!] \cup [\![g]\!] \qquad [\![f \cdot g]\!] := [\![f]\!] \cdot [\![g]\!] \qquad [\![f^*]\!] := [\![f]\!]^*$$

The above definition defines the interpretation of an expression in the language model as a language. Therefore, two expressions are equal in the language model precisely when their corresponding languages are equal. Not every language can be represented by a regular expression, for example the language $\{a^n b^n \mid n \in \mathbb{N}\}$ cannot be represented by a regular expression [HMU06]. The languages that can be represented by a regular expression are called *regular languages*.

Usually a model requires a separate interpretation function that defines the interpretation of the primitive actions. In this definition we use the standard interpretation for the language model, i.e. we interpret an action a as $\{a\}$. In the following example we demonstrate how we can informally reason about the language of an expression.

Example 2.4. The expression $(b + ab^*a)^*ab^*$ is interpreted in the language model as the set of strings over $\{a, b\}$ that have an odd number of a's. We can read this expression as follows. Every string starts with zero or more times $(b + ab^*a)$ which represents either the string b or the strings that start and end with an a and have any number of b's in the middle. If we repeat those strings a number of times we always add zero or two a's and therefore the number of a's is always even. By choosing the b any number of times at the start we can control the number of b's before the first a. The b^* then controls the number of b's in the middle and by choosing the b a number of times after the second a we can control the number of b's at the end. Therefore, $(b + ab^*a)^*$ contains all words over $\{a, b\}$ with an even number of a's. Finally, at the end of the full expression we have ab^* which adds one a making the total number of a's odd. The b^* at the end then ensures that the expression contains all possible strings over $\{a, b\}$ that have an odd number of a's.

Another model of KA that is of interest is the relational model. The relational model deals with sets of binary relations over a set S. The relational model can be used to reason about programs in terms of the effect a program has on a state from S. In this view, an interpretation of an expression is the relation between states before and after executing the program. Before we can formally introduce the relational model we must first introduce several operations for relations. For two binary relations R_1 and R_2 we have the relational composition $R_1 \circ R_2$ where:

$$R_1 \circ R_2 := \{(s_1, s_3) \mid \exists s_2 \ (s_1, s_2) \in R_1, (s_2, s_3) \in R_2\}$$

We use id_S to denote the identity relation over a set S, i.e. for all $s \in S$ we have $(s, s) \in \mathsf{id}_S$. We can then inductively define exponentiation for relations as follows:

$$R^0 := \mathsf{id}_S \qquad\qquad R^{n+1} := R \circ R^n$$

Furthermore, we use R^* to denote the reflexive-transitive closure of R, which can also be characterised by the following union:

$$R^* = \bigcup_{n \geq 0} R^n$$

Now we can formally define the relational model.

Definition 2.5 (Relational Model [Koz94]). The tuple $(2^{S \times S}, \cup, \circ, (-)^*, \emptyset, \mathsf{id}_S)$ is a KA that is called the relational model. An interpretation for a symbol in the relational model is a pair (S, σ) where $\sigma : \Sigma \to 2^{S \times S}$ interprets a symbol from Σ as a relation on S. The interpretation of an expression is then defined inductively by the function $[\![-]\!]_\sigma : \mathsf{Exp} \to 2^{S \times S}$ as follows:

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

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

The lemma below states the connection between the relation model and the language model.

Lemma 2.6 ([Pra80]). *For all* KA *expressions* f *and* g, $[\![f]\!] = [\![g]\!]$ *if and only if for all relational interpretations* σ, $[\![f]\!]_\sigma = [\![g]\!]_\sigma$.

This lemma states that an equality holds in the language model if and only if it holds for all relational interpretations.

2.2 Finite State Automata

Finite state automata are abstract devices that accept or reject inputs by reading the input, symbol by symbol, and transitioning between states. After reading an entire input we either end up in an accepting state and accept, or we do not and reject the input.

Definition 2.7 (Finite State Automata). A finite state automaton is a tuple (Q, δ, F, q_0), where Q is a set of states, $\delta : Q \times \Sigma \to Q$ is the transition function, $F \subseteq Q$ is the set of accepting states and $q_0 \in Q$ is the initial state.

We use the notation $q \xrightarrow{a}_\delta q'$ to denote $\delta(q, a) = q'$. If δ is clear from the context we will simply write $q \xrightarrow{a} q'$. We now define what words are accepted by a state inductively.

Definition 2.8. The words accepted by a state q are given by the following conditions:

- The empty word is accepted if and only if q is an accepting state.

- The word $a \cdot w$ is accepted if and only if there is a transition $q \xrightarrow{a}_{\delta} q'$ where q' accepts w.

This definition corresponds precisely to starting in state q, reading a word, symbol by symbol, taking the corresponding transitions and ending up in an accepting state after reading the entire word. Finally, a word is accepted by an automaton if it is accepted by the initial state of that automaton. We can then define the language of a state or automaton as the set of words that are accepted by that state or automaton. We will use $L(q)$ and $L(A)$ to denote the language of a state and an automaton respectively.

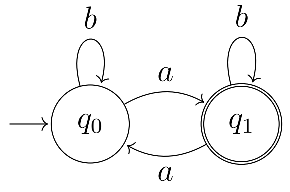
Another way to characterise strings accepted by an automaton is in terms of the extended transition function.

Definition 2.9. The extended transition function $\delta^* : Q \times \Sigma^* \to Q$ defines transition for strings instead of just symbols. It is defined inductively as follows: $\delta^*(q, \epsilon) = q$ and $\delta^*(q, w \cdot a) = \delta(\delta^*(q, w), a)$.

Using the extended transition function we can characterise $L(q)$ as the set of words w where $\delta^*(q, w)$ is an accepting state.

An automaton is drawn with circles representing states and arrows between the states representing the transitions. The initial state is marked by an arrow without source and accepting states are drawn as a double circle.

Example 2.10. The automaton below has two states: q_0 and q_1. The state q_0 is the initial state and not an accepting state. The state q_1 is an accepting state. We can read the word *bbab* and check if its accepted as follows. We start in q_0 and take the transition corresponding to the first symbol in the string. This is the b transition. After taking this transition we are still in q_0. Then we check which transition corresponds to the next symbol. Once again this is the b transition and we stay in q_0. Next we read an a and and after taking the a transition we end up in q_1. Finally we read b, take the b transition in q_1 and stay in q_1. Now that the entire word has been read we check if we are in an accepting state, which we are and therefore we accept.



The automaton above accepts exactly all strings over the alphabet $\{a, b\}$ with an odd number of a's. It is a deterministic finite state automaton (DFA). We now define a non-deterministic finite state automaton (NFA).

Definition 2.11 (NFA)**.** A non-deterministic finite state automaton is a tuple (Q, δ, F, q_0), where Q is a set of states, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, $F \subseteq Q$ is the set of accepting states and $q_0 \in Q$ is the initial state.

The difference between a DFA and an NFA is that a DFA transitions to a state while an NFA transitions to a set of states. Therefore, when you read the symbol a in a state there can be multiple transitions to different states for the same input symbol or no transitions at all. We now inductively define the words accepted by a state in an NFA as follows.

Definition 2.12. The words accepted by a state q in an NFA are defined by the following conditions:

- The empty word is accepted if and only if q is an accepting state.
- The word $a \cdot w$ is accepted if and only if w is accepted by a state in $\delta(q, a)$.

Note that there only needs to be at least one path of transitions that reaches an accepting state, not every path. NFAs are related to DFAs by the following theorem.

Theorem 2.13 (Subset construction [RS59])**.** *For every NFA $A = (Q, \delta, F, q_0)$ if we define the A_d as $A_d = (2^Q, \delta_d, \{S \mid F \cap S \neq \emptyset\}, \{q_0\})$, where $\delta_d(S, a) = \bigcup_{q \in S} \delta(q, a)$ then it holds that $L(A) = L(A_d)$.*

This theorem states that for every NFA there is a DFA that accepts the same language. We can achieve the inverse, i.e. for every DFA construct an NFA that accepts the same language, by converting every state into the set containing that state. Therefore NFA and DFA are equivalent.

An important result regarding the connection between automata and expressions is the following theorem.

Theorem 2.14 (Kleene's Theorem [Kle56])**.** *For every expression e in KA there is a finite state automaton A_e that accepts the same language and for every finite state automaton A there is an expression e_A in KA that accepts the same language.*

A well known algorithm for constructing finite state automata for regular expressions is the Thompson construction [Tho68]. The Thompson construction creates an NFA, from which we can then obtain an equivalent DFA using Theorem 2.13.

We can describe properties of the expression corresponding to an automaton by defining the notion of a solution.

Definition 2.15. A solution to an automaton, (Q, δ, F, q_0), is a function, $s : Q \to \mathsf{Exp}$, that assigns an expression to every state. This function must satisfy the following equivalence for all $q \in Q$:

$$[q \in F] + \sum_{q \xrightarrow{a}_{\delta} q'} a \cdot s(q') \leqq s(q)$$

We use the notation $[q \in F]$ to represent 1 if $q \in F$ and 0 otherwise. We use the Σ in this equivalence to denote a summation over the transitions from state q. By summation we mean adding up all the elements using the $+$. The summation does not describe the order in which the elements are added but, because KA is commutative and associative, this is not relevant.

The *least solution* s to an automaton is a solution such that for every solution t to the same automaton and state q, it holds that $s(q) \leqq t(q)$. The following lemma connects solutions to Theorem 2.14. An example of a proof of this lemma can be found in [Kap23].

Lemma 2.16. *If $s : Q \to \mathsf{Exp}$ is the least solution to the automaton (Q, δ, F, q_0) then for every $q \in Q$ it holds that $[\![s(q)]\!] = L(q)$.*

In particular this lemma states that the least solution to the initial state has the same language as the automaton, i.e. for an automaton $A = (Q, \delta, F, q_0)$ and a least solution s of A, it holds that $[\![s(q_0)]\!] = L(A)$. We also know the following about least solutions.

Lemma 2.17 ([Ard61])**.** *For every automaton there exists a least solution and this least solution can be computed.*

Another important theorem about least solutions is the following:

Theorem 2.18 (Round Trip [Koz94])**.** *Let $A = (Q, \delta, F, q_0)$ be an automaton constructed for the expression f and let s be the least solution to A. Then $s(q_0) \equiv f$.*

This theorem states that after constructing an automaton for an expression we can recover the expression by finding the least solution to the automaton.

2.3 Completeness and Decidability of KA

For KA, completeness of a model says that if two expressions have the same semantics in that model, then they are provably equivalent. The completeness of KA refers specifically to the completeness of KA with respect to the language model. Therefore the completeness of KA says that if two expressions have the same language we can construct a proof from the axioms of KA to prove their equivalence.

Theorem 2.19 (Completeness of KA [Koz94])**.** *For any two KA expressions f and g, if $[\![f]\!] = [\![g]\!]$ then $f \equiv g$.*

This is not true for all systems. For example there are true statements in any consistent mathematical system capable of describing basic arithmetic that cannot be proven [Gö31]. Decidability says there exists a terminating algorithm that decides if two expressions are provably equivalent. The decidability of KA follows from its completeness and the decidability of equality for regular languages. The equality of two regular languages can be decided by checking the equality of the automata corresponding to the languages.

2.4 Hypothesis $e = 0$

The hypothesis $e = 0$ allows us to prove new equivalences. As an example take the equivalence $f + e \equiv f$. Without the hypothesis this equivalence is not true for arbitrary expressions. However, when we use the hypothesis $e = 0$, the equivalence follows from $e = 0$ and $f = f$. Therefore the completeness of KA is not sufficient to guarantee decidability under the hypothesis $e = 0$. However KA is actually still decidable under the hypothesis $e = 0$ [Coh94]. Cohen proves the decidability of KA under the hypothesis $e = 0$ by proving the completeness of a specific model and interpretation where $e = 0$ and showing that equality is decidable in that model. When we write that $e = 0$ in a model K and interpretation σ_K we mean that $[\![e]\!]_{\sigma_K} = [\![0]\!]_{\sigma_K}$.

As an equation the completeness of a model K and interpretation σ_K under the hypothesis $e = 0$ is:

$$K, \sigma_K \models f = g \implies \mathsf{KA}, e = 0 \vdash f = g$$

Here we use $K, \sigma_K \models f = g$ to denote that $[\![f]\!]_{\sigma_K} = [\![g]\!]_{\sigma_K}$. Furthermore we use $\mathsf{KA}, e = 0 \vdash f = g$ to denote that $f = g$ is provable from the axioms of KA and $e = 0$. Without a hypothesis the notation $\mathsf{KA} \vdash f = g$ is equivalent to $f \equiv g$.

An example of a practical use of a hypothesis $e = 0$ is the encoding of a Hoare triple in KAT. In KAT we can encode a Hoare triple $\{b\}S\{c\}$ as $bS\overline{c} = 0$ [Koz00]. The partial correctness of a Hoare triple $\{b\}S\{c\}$ denotes that if the test b holds at the start and we run the program S (and S terminates), then the test c holds at the end. Therefore the completeness of KA under the hypothesis $e = 0$ means we can prove statements under the assumption of the partial correctness of certain programs using KA.

2.5 Guarded Kleene Algebra with Tests

GKAT [SFH+19] differs from KA in two aspects. Firstly, it adds Boolean tests which are used in expressions as assertions. An assertion immediately terminates either with failure or success depending on whether the test is true or not. Secondly, it changes the operations $+$ and $(-)^*$ into deterministic variants. The $+$ becomes $f +_b g$ which models **if** b **then** f **else** g. The $(-)^*$ becomes $f^{(b)}$ which models **while** b **do** f. This allows GKAT to very directly model imperative programs. The Boolean expressions (BExp) used in GKAT expressions are generated by the following grammar:

$$b, c ::= 0 \mid 1 \mid t \in T \mid b \cdot c \mid b + c \mid \overline{b}$$

Here T is a set of abstract primitive tests, 0 is used to mean **false** and 1 is used to mean **true**. As a convention we use b and c to denote Boolean expressions. We now formally define Boolean algebra.

Definition 2.20 (Boolean Algebra [Hun04])**.** A Boolean algebra is a tuple $(B, +, \cdot, \neg, 0, 1)$ where for all $b, c, d \in B$ the following axioms hold:

$$b + 0 = b \qquad b \cdot 1 = b \qquad b + c = c + b \qquad b \cdot c = c \cdot b$$

$$b + (c \cdot d) = (b + c) \cdot (b + d) \qquad b \cdot (c + d) = (b \cdot c) + (b \cdot d) \qquad b + \neg b = 1 \qquad b \cdot \neg b = 0$$

We will use $\equiv_{BA}$ to denote provability in Boolean algebra. Notably both the $\cdot$ and the $+$ are associative as well, but these properties are not axioms themselves because they are provable from the stated axioms [Hun04].

Expressions in GKAT are then generated by the following grammar:

$$f, g ::= p \in \Sigma \mid b \in \mathsf{BExp} \mid f \cdot g \mid f +_b g \mid f^{(b)}$$

Here Σ is a set of abstract primitive actions. In a GKAT expression an action p means do p and a Boolean test b means assert b. As a convention we will use e, f, g and h to denote expressions in GKAT. Lastly we introduce an auxiliary function E that is used in the axiomatisation.

Definition 2.21 ([SFH+19])**.** The function $E : \mathsf{Exp} \to \mathsf{BExp}$ is defined inductively as follows:

$$E(b) := b \qquad E(p) := 0 \qquad E(f +_b g) := b \cdot E(f) + \overline{b} E(g) \qquad E(f \cdot g) := E(f) \cdot E(g) \qquad E(f^{(b)}) := \overline{b}$$

The function $E(f)$ assigns to the expression f a test that determines whether f terminates without performing an action. If $E(f) = 0$ then some action p is executed during the execution of f. If $E(f) = b$ then if b is true no action is executed during the execution of f. We can now formally define GKAT.

Definition 2.22 (Guarded Kleene Algebra with Tests [SFH+19])**.** A GKAT is a tuple

$$(K, B, +_b, \cdot, (-)^{(b)}, 0, 1, +, \neg, F)$$

where $(B, +, \cdot, \neg, 0, 1)$ is a Boolean algebra, $F : K \to \mathsf{BExp}$ is a function such that for every interpretation σ_K, $F(\llbracket f \rrbracket_{\sigma_K}) = E(f)$, and for all $x, y, z \in K$ and $b, c \in B$ the following axioms hold:

$$x +_b x = x \qquad x +_b y = y +_{\overline{b}} x \qquad (x +_b y) +_c z = x +_{bc} (y +_c z)$$

$$x +_b y = b \cdot x +_b y \qquad x \cdot z +_b y \cdot z = (x +_b y) \cdot z \qquad (x \cdot y) \cdot z = x \cdot (y \cdot z)$$

$$0 \cdot x = 0 \qquad x \cdot 0 = 0 \qquad 1 \cdot x = x \qquad x \cdot 1 = x$$

$$x^{(b)} = xx^{(b)} +_b 1 \qquad (x +_c 1)^{(b)} = (cx)^{(b)} \qquad \frac{z = xz +_b y}{z = x^{(b)} y} \text{ if } F(x) = 0$$

Furthermore, because B is a Boolean algebra, Boolean equivalence still holds in GKAT, i.e. if $b \equiv_{BA} c$ then $b \equiv c$. Here we use $\equiv$ to denote provability in GKAT and $\equiv_{BA}$ to denote provability in Boolean algebra.

These axioms are quite intuitive when thought of as representing programs. For example take the axiom $x +_b y = y +_{\overline{b}} x$. The left hand side models the program **if** b **then** x **else** y and the right hand side models the program **if not** b **then** y **else** x. In terms of programs it corresponds to the expectation that we obtain an equivalent program if we flip both the condition and the branches. Another example is the axiom $x^{(b)} = xx^{(b)} +_b 1$. The left side of the equality models the program **while** b **do** x and the right side models the program **if** b **then** (x; **while** b **do** x) **else skip**. This axiom unrolls the loop. Either the condition does not hold and we terminate immediately or we execute the loop once and go back to the start of the loop.

The last axiom has the side condition $F(x) = 0$. This side condition states that we can only apply the rule when the loop's body x is productive. This is to prevent situations like in the following example.

Example 2.23. From the axioms we can prove that $1 \equiv 1 \cdot 1 +_1 1$. If we then apply the last axiom, ignoring the side condition, we obtain $1 \equiv 1^{(1)} \cdot 1$. The problem is that $1^{(1)} \equiv 0$ because $1^{(1)}$ never terminates or performs any action. Therefore $1 \equiv 1^{(1)} \cdot 1$ becomes $1 \equiv 0$, which is false.

For GKAT unless otherwise specified we will use the language model. The language model for KA and the language model for GKAT are quite different. The language model for GKAT interprets expressions as *guarded languages*, i.e. sets of guarded strings. A guarded string is a string that alternates between *atoms* and actions. An atom describes the state of a program. Formally an atom of a set of tests $T = \{b_1, ...b_k\}$ is a Boolean expression $c_1 \cdot c_2 ... c_k$ where $c_i \in \{b_i, \overline{b_i}\}$. This represents a truth assignment on T as follows: if $c_i = b_i$ then b_i is true and if $c_i = \overline{b_i}$ then b_i is false. We use At to denote the set of all atoms and as a convention we use α and β to denote atoms. Furthermore we use $\alpha \leq b$ to denote that b is true for the truth assignment α. A guarded string is then an element of the regular language $\mathsf{GS} := \mathsf{At} \cdot (\Sigma \cdot \mathsf{At})^*$. This alternation between atoms and actions describes a trace of an abstract program. In other words it describes the state of a program over time and the actions that changed the state.

We can sequentially compose guarded strings through a partial *fusion product* $\diamond : \mathsf{GS} \times \mathsf{GS} \rightharpoonup \mathsf{GS}$. The partial fusion product is defined as follows:

$$x\alpha \diamond \beta y := \begin{cases} x\alpha y & \text{if } \alpha = \beta \\ \text{undefined} & \text{otherwise} \end{cases}$$

Here $x\alpha$ and βy are guarded strings. This definition states that we can only sequentially compose two guarded strings if the last atom of the first guarded string and the first atom of the second guarded string are identical. If we were to define composition of guarded strings identical to composition of normal strings we could have traces where the state changes without any action occurring in between.

Before we can define the language model we need to introduce several more operations. We lift the partial fusion product to languages with the following definition: $L_1 \diamond L_2 := \{x \diamond y \mid x \in L_1, y \in L_2\}$. We can then inductively define exponentiation for guarded languages as $L^{n+1} := L^n \diamond L$ where $L^0 := \mathsf{At}$. For a guarded language $B \subseteq \mathsf{At}$ we use $\overline{B}$ to denote $\mathsf{At} \setminus B$. Subsequently we define two more operations for guarded languages.

$$L_1 +_B L_2 := (B \diamond L_1) \cup (\overline{B} \diamond L_2) \qquad\qquad L^{(B)} := \bigcup_{n \geq 0} (B \diamond L)^n \diamond \overline{B}$$

The result of $L_1 +_B L_2$ is the set of traces from L_1 that start with an atom from B and the traces from L_2 that start with an atom from $\overline{B}$. The result of $L^{(B)}$ is the set of traces in which each trace consists of a sequence of traces from L, where the traces from L start and end with an atom from B, and this sequence ends with an atom from $\overline{B}$. Now we can finally define the language model.

Definition 2.24 (Language Model [SFH+19])**.** The tuple $(2^{\mathsf{GS}}, 2^{\mathsf{At}}, +_B, \diamond, (-)^{(B)}, \emptyset, \mathsf{At}, +, \neg, F)$, where $F(L) = L \cap \mathsf{At}$, is a GKAT that is called the language model. An interpretation of an expression in the language model is defined inductively by the function $[\![-]\!] : \mathsf{Exp} \to 2^{\mathsf{GS}}$ as follows:

$$[\![b + c]\!] := [\![b]\!] \cup [\![c]\!] \qquad [\![\bar{b}]\!] := \overline{([\![b]\!])}$$

$$[\![p]\!] := \{\alpha p \beta \mid \alpha, \beta \in \mathsf{At}\} \qquad [\![b]\!] := \{\alpha \in \mathsf{At} \mid \alpha \leq b\} \qquad [\![0]\!] := \emptyset \qquad [\![1]\!] := \mathsf{At}$$

$$[\![f +_b g]\!] := [\![f]\!] +_{[\![b]\!]} [\![g]\!] \qquad [\![f \cdot g]\!] := [\![f]\!] \diamond [\![g]\!] \qquad [\![f^{(b)}]\!] := [\![f]\!]^{([\![b]\!])}$$

In this definition an action p is interpreted as all traces that start in an arbitrary state, do p and end in an arbitrary state. Actions are interpreted in this way because we do not make any assumptions about the behaviour of primitive actions. A test is interpreted as any atom for which the test holds and the Boolean algebra of the language model consists of sets of atoms. The three operations are interpreted via the three operations we defined for guarded languages. In conclusion, an interpretation in the language model is a set of traces that represent valid executions of an expression.

Furthermore, GKAT has a relational model in which expressions are interpreted as binary relations over a set of states S. A primitive action is interpreted as the relation between states before and after executing the action. This interpretation is given by the function $\mathsf{eval} : \Sigma \to 2^{S \times S}$. A primitive test is interpreted as the identity relation over the states for which the test holds. The set of states for which a test holds is given by the function $\mathsf{sat} : T \to 2^S$. Before we can formally introduce the relational model we must first introduce some new operations. For a relation $B \subseteq \mathsf{id}_S$ we use $\overline{B}$ to denote $\mathsf{id}_S \setminus B$. We then define two more operations for relations:

$$R_1 +_B R_2 := B \circ R_1 \cup \overline{B} \circ R_2 \qquad\qquad R^{(B)} := (B \circ R)^* \circ \overline{B}$$

The result of $R_1 +_B R_2$ consists of pairs (s_1, s_2) from R_1 where $(s_1, s_1) \in B$ and pairs (s_3, s_4) where $(s_3, s_3) \in \overline{B}$. The results of $R^{(B)}$ consists of pairs (s_1, s_2) from the reflexive-transitive closure of $B \circ R$ where $(s_2, s_2) \in \overline{B}$. Now we can formally define the relational model.

Definition 2.25 (Relational Model [SFH+19])**.** The tuple $(2^{S\times S}, 2^{\mathsf{id}_S}, +_B, \circ, (-)^{(B)}, \emptyset, \mathsf{id}_S, +, \neg, F)$, where $F(R) = R \cap \mathsf{id}_S$, is a GKAT that is called the relational model. The interpretation of an expression for the relational interpretation $\sigma = (S, \mathsf{eval}, \mathsf{sat})$ is defined inductively by the function $[\![-]\!]_\sigma : \mathsf{Exp} \to 2^{S\times S}$ as follows:

$$[\![b+c]\!]_\sigma := [\![b]\!]_\sigma \cup [\![c]\!]_\sigma \qquad [\![\bar{b}]\!]_\sigma := \overline{([\![b]\!]_\sigma)}$$

$$[\![p]\!]_\sigma := \mathsf{eval}(p) \qquad [\![b]\!]_\sigma := \{(s,s) \mid s \in \mathsf{sat}(b)\} \qquad [\![0]\!]_\sigma := \emptyset \qquad [\![1]\!]_\sigma := \mathsf{id}_S$$

$$[\![f +_b g]\!]_\sigma := [\![f]\!]_\sigma +_{[\![b]\!]_\sigma} [\![g]\!]_\sigma \qquad [\![f \cdot g]\!]_\sigma := [\![f]\!]_\sigma \circ [\![g]\!]_\sigma \qquad [\![f^{(b)}]\!] := [\![f]\!]_\sigma^{([\![b]\!]_\sigma)}$$

The Boolean algebra for the relational model consists of subsets of the diagonal and a test is interpreted as the pairs (s,s) where b holds in s. Analogous to KA, there is a connection between the relational model and the language model in GKAT.

Lemma 2.26 ([SFH+19])**.** *For all* GKAT *expressions* f *and* g, $[\![f]\!] = [\![g]\!]$ *if and only if for all relational interpretations* σ, $[\![f]\!]_\sigma = [\![g]\!]_\sigma$.

2.6 Automaton Model for GKAT

GKAT has an automaton model resembling the automaton model of KA. The automaton models are quite similar but the GKAT model is slightly more complicated in order to allow it to handle both tests and actions. A GKAT automaton reads a guarded string as input. Unlike KA automata, a GKAT automaton has explicit transitions for accepting and rejecting. Furthermore, transitions between states specify both an atom and an action. We use 0 to denote rejection, 1 to denote acceptance and 2 to denote the set $\{0,1\}$. A guarded string is accepted if after reading the string until the last atom we end up in a state that has an accepting transition for the last atom.

Definition 2.27 (GKAT automata [SFH+19])**.** A GKAT automaton is a tuple (Q, δ, ι), where Q is a set of states, $\delta : Q \times \mathsf{At} \to 2 \cup \Sigma \times Q$ defines transitions between states or acceptance or rejection and $\iota \in Q$ is the initial state.

It is important to note that in the definition above, 2 and $\Sigma \times Q$ are disjoint and therefore it is always clear whether δ specifies a termination or a transition to a state. We define acceptance formally as follows.

Definition 2.28 ([SFH+19])**.** The guarded words accepted by a state q in a GKAT automaton are defined inductively by the following conditions:

- An atom α is accepted by a state q if and only if $\delta(q)(\alpha) = 1$.
- A guarded string $\alpha p x$ is accepted by a state q if and only if $\delta(q)(\alpha) = (p, q')$ where x is accepted by q'.

As in KA, the language of a state is the set of all words accepted by that state and the language of an automaton is the language of the initial state of that automaton.

A GKAT automaton is depicted by drawing arrows between the states and an arrow pointing towards the initial state. An accepting transition is denoted with a double arrow pointing towards an atom and all atoms not covered by a transition are implicit rejections. Transitions between states are labelled with both an atom and an action. When reading an input we can only take a transition if both the atom and the action match. Automata are often drawn with tests on the transitions instead of atoms. In this case we can take the transition labelled with the test b if we read an atom α where $\alpha \leq b$. It is important to note that tests on transitions from the same state are not allowed to overlap. In other words for every state, every atom should only match with a single transition.

Example 2.29. In the following automaton we start in the initial state q_1. If we read the input $\alpha q \beta$ where $\alpha \leq \bar{b}$ and $\beta \leq c$ we first transition to q_2 after reading αq and then accept using the c transition after reading β. If we read the input $\alpha q \beta p \alpha$ instead, we would still take the accepting transition after reading β but because the string has not been fully read we do not accept. In the state q_3 there is a transition for $\bar{b}$ and c but not for $b \cdot \bar{c}$. Therefore this transition would be an immediate rejection. Finally, the string $\alpha p \beta$ would be rejected because α means we take the transition to q_2 but the action does not match.

$$\begin{tikzcd}
& & q_3 \arrow[loop above, "c|r"] \arrow[dd, "\bar{b}|q"'] \\
{} \arrow[r] & q_1 \arrow[ur, "b|p"] \arrow[dr, "\bar{b}|q"'] & \\
& & q_2 \arrow[loop right, "b|p"] \arrow[d, Rightarrow] \\
& & c
\end{tikzcd}$$

As with KA there is a correspondence between automata and expressions in GKAT. However, contrary to KA not every GKAT automaton has a corresponding GKAT expression [KT08]. There is a Kleene Theorem for a subset of GKAT automata called *well-nested* automata.

Theorem 2.30 (Kleene Theorem GKAT [SFH+19])**.** *For every expression e in* GKAT *there is a well-nested* GKAT *automaton A_e that accepts the same guarded language and for every well-nested* GKAT *automaton A there is an expression e_A in* GKAT *that accepts the same guarded language.*

Non-well-nested automata encode control flows that use **goto** or **break** statements which cannot be encoded in GKAT expressions. For a formal definition of well-nested automata we refer to Smolka et al. [SFH+19]. The expression corresponding to a GKAT automaton is given by the solution to that automaton. We define a solution to a GKAT automaton as follows:

Definition 2.31 (Solution [SFH+19])**.** Let $A = (Q, \delta, \iota)$ be a GKAT automaton. We call $s : Q \to \mathsf{Exp}$ a solution to A if for all $\alpha \in \mathsf{At}$ and $q \in Q$ the following holds:

$$\alpha \cdot s(q) \equiv \alpha \cdot \lfloor \delta(q)(\alpha) \rfloor_s \qquad \text{where} \qquad \lfloor 0 \rfloor_s := 0 \qquad \lfloor 1 \rfloor_s := 1 \qquad \lfloor (p, q) \rfloor_s := p \cdot s(q)$$

Not every automaton has a corresponding expression and therefore not every automaton has a solution. In the case that an automaton has a solution, it is unique up to equivalence. The construction used to create automata from expressions always yield well-nested automata and these automata always have a solution. As stated in the following theorem this solution is in fact equivalent to the original expression.

Theorem 2.32 (Round-Trip Theorem [SFH+19]). *For any* GKAT *expression f the corresponding automaton $A_f = (Q, \delta, \iota)$ has a solution such that $f \equiv s(\iota)$.*

2.7 Completeness and Decidability of GKAT

Analogous to KA, the completeness of GKAT is with respect to its language model. Before we discuss the completeness of GKAT we must introduce another axiom that was left out of Definition 2.22. This is the uniqueness axiom which makes a statement about *left-affine* systems, which are systems of the following form:

$$\begin{gathered} \boldsymbol{x_1} = e_{11}\boldsymbol{x_1} +_{b_{11}} \cdots +_{b_{1(n-1)}} e_{1n}\boldsymbol{x_n} +_{b_{1n}} d_1 \\ \vdots \\ \boldsymbol{x_n} = e_{n1}\boldsymbol{x_1} +_{b_{n1}} \cdots +_{b_{n(n-1)}} e_{nn}\boldsymbol{x_n} +_{b_{nn}} d_n \end{gathered}$$

where x_i are variables, e_{ij} are GKAT expressions and b_{ij} and d_i are Boolean guards. In these equations all the guards in a row are disjoint, i.e. for all j and k, $b_{ij} \cdot b_{ik} \equiv_{BA} 0$ and $b_{ij} \cdot d_i \equiv_{BA} 0$. A system is called *Salomaa* if for all e_{ij} it holds that $E(e_{ij}) = 0$ [SFH+19]. The term Salomaa is a reference to Salomaa's axiomatisation of regular languages which contains an equivalent property called the empty-word property [Sal66]. The uniqueness axiom then states that all left-affine systems that are Salomaa have at most one solution. Finally, when we combine the axioms from Definition 2.22 with the uniqueness axiom the following theorem holds.

Theorem 2.33 (Completeness of GKAT [SFH+19]). *For any two* GKAT *expressions f and g, if $[\![f]\!] = [\![g]\!]$ then $f \equiv g$.*

Following from the completeness of GKAT and the decidability of equality of guarded languages [SFH+19] we deduce that equivalence in GKAT is decidable. The uniqueness axiom is less convenient than the algebraic axioms from Definition 2.22. For example, checking if a model satisfies the uniqueness axiom would require verifying that a possibly infinite number of systems of equations has at most one solution, which is often not feasible. There is a fragment of GKAT, called skip-free, that is complete without this uniqueness axiom [KSS23] but we will not be using this fragment.

Finally, contrary to KA, the decidability of GKAT under the hypothesis $e = 0$ has not been proven. Cohen [Coh94] proves this for KA by characterising provability in KA under the hypothesis $e = 0$ as equality in a model M with interpretation m, where for all expressions f, $[\![f]\!]_m = [\![f + \Sigma^* e \Sigma^*]\!]$. The decidability of KA under the hypothesis $e = 0$ then follows from the decidability of equality for regular languages. In GKAT, however, we have $+_b$ instead of $+$. Therefore, a characterisation as in Cohen's proof is not possible in GKAT. In this thesis we create a similar proof in which we characterise provability in KA under the hypothesis $e = 0$ as equality in a model K_e with interpretation σ_e, where for all expressions f, $[\![f]\!]_{\sigma_e} = [\![f]\!] \setminus [\![\Sigma^* e \Sigma^*]\!]$.

3 Decidability of KA under $e = 0$

In this section we prove the decidability of KA under the hypothesis $e = 0$. We start with KA because we know that KA is decidable under the hypothesis $e = 0$ and because KA is simpler to work with than GKAT. This allows to us to determine if this strategy is a suitable way to prove decidability under the hypothesis $e = 0$ before attempting to create the proof for GKAT.

3.1 Proof Sketch

We prove the decidability of KA by characterising provability in KA under the hypothesis $e = 0$ as equality in the model K_e and interpretation σ_e. The proof consists mainly of the creation of two new expressions $\hat{f}$ and $\hat{g}$ that satisfy the following two properties. Firstly that $\mathsf{KA}, e = 0 \vdash f = \hat{f}$ and $\mathsf{KA}, e = 0 \vdash g = \hat{g}$. Secondly that $[\![f]\!]_{\sigma_e} = [\![\hat{f}]\!]$ and $[\![g]\!]_{\sigma_e} = [\![\hat{g}]\!]$.

Intuitively the expressions $\hat{f}$ and $\hat{g}$ are created by "removing" e from f and g, where by "removing" e from f and g we mean removing all words from f and g that contain a word from e. Under the hypothesis $e = 0$ we know that for all words x that contain a word from e we have $x = 0$. Therefore "removing" x from f will not change f. After "removing" all words x from f, the hypothesis $e = 0$ will no longer be relevant because e will not occur anywhere in f. Then because the hypothesis has become irrelevant, proving the equivalence of those expressions is solved by Theorem 2.19.

Concretely our proof consists of the following steps. Firstly, we show that $\mathsf{KA}, e = 0 \vdash f = \hat{f}$ and $\mathsf{KA}, e = 0 \vdash g = \hat{g}$. Secondly, we construct a KA K_e and interpretation σ_e where $[\![e]\!]_{\sigma_e} = [\![0]\!]_{\sigma_e}$ such that $[\![f]\!]_{\sigma_e} = [\![\hat{f}]\!]$ and $[\![g]\!]_{\sigma_e} = [\![\hat{g}]\!]$. This allows us to conclude that:

$$[\![f]\!]_{\sigma_e} = [\![g]\!]_{\sigma_e} \implies [\![\hat{f}]\!] = [\![\hat{g}]\!]$$

Note that the equality $[\![\hat{f}]\!] = [\![\hat{g}]\!]$ does not depend on our hypothesis. Therefore we can use Theorem 2.19 to conclude that $\hat{f} \equiv \hat{g}$. Combined with $\mathsf{KA}, e = 0 \vdash f = \hat{f}$ and $\mathsf{KA}, e = 0 \vdash g = \hat{g}$ this results in $\mathsf{KA}, e = 0 \vdash f = g$, which concludes our proof.

3.2 Removal Operation

The construction for $\hat{f}$ and $\hat{g}$ is the same. Therefore we write proofs about an arbitrary expression f. Before we can prove that $\mathsf{KA}, e = 0 \vdash f = \hat{f}$, we first have to define exactly what we mean by "removing" words containing a word in e from f. The expression $e' = \Sigma^* e \Sigma^*$ corresponds to all words that contain a word in e. We create $\hat{f}$ by removing e' from f. By "removing" e' from f we intend that the language of $\hat{f}$ contains all the words from f without any of the words from e'. In other words for any expression f we want the language of $\hat{f}$ to be $[\![f]\!] \setminus [\![e']\!]$. For our proof however, we need an expression, not merely this description in terms of the difference of languages.

To accomplish this we first use the Thompson construction to create an automaton for f and then use the construction from Theorem 2.13 to obtain a DFA that accepts the same language. Theorem 2.18 states that the least solution to this automaton is equivalent to f. We use $A_f = (Q_f, \delta_f, F_f, q_f)$ to denote this automaton. We then go through the same steps to obtain a deterministic automaton for e' that we denote with $A'_e = (Q'_e, \delta'_e, F'_e, q'_e)$. Finally, we use the *product construction* [RS59] to construct an automaton that accepts the difference of these two automata, which is exactly $[\![f]\!] \setminus [\![e']\!]$. The product construction produces the following automaton:

$$A = (Q_f \times Q'_e, \delta, F, (q_f, q'_e)) \qquad \text{where}$$
$$\delta((p,q),a) = (\delta_f(p,a), \delta'_e(q,a)) \qquad \text{and} \qquad F = \{(p,q) \mid p \in F_f \land q \notin F'_e\}$$

This automaton works by simulating both A_f and A'_e and only accepting words that are accepted by A_f but not by A'_e. It is able to simulate both automata by having a state space that consists of tuples. Each tuple contains the states from A_f and A'_e that we would be in if we read the input in those automata. Before we prove the correctness of the construction we must first prove a property of the extended transition function of the product automaton.

Lemma 3.1. *Let $A_0 = (Q_0, \delta_0, F_0, q_0)$ and $A_1 = (Q_1, \delta_1, F_1, q_1)$ be DFAs and let $A = (Q, \delta, F, (q_0, q_1))$ be the product automaton for A_0 and A_1. Then $\delta^*((p,q),w) = (\delta_0^*(p,w), \delta_1^*(q,w))$.*

Proof. We prove the lemma by induction. The base case is that for strings of length 0, the lemma holds by the definition of δ^*. This follows from $\delta^*((p,q),\epsilon) = (p,q)$, $\delta_0^*(p,\epsilon) = p$ and $\delta_1^*(q,\epsilon) = q$. Assume as induction hypothesis that the lemma holds for strings w. We now show the lemma then also holds for a string wa.

We know from the induction hypothesis that $\delta^*((p,q),w) = (\delta_0^*(p,w), \delta_1^*(q,w))$. Then by the definition of δ:

$$\delta^*((p,q),wa) = (\delta_f(\delta_0^*(p,w),a), \delta'_e(\delta_1^*(q,w),a))$$

Finally by the definition of δ^* we obtain $\delta^*((p,q),wa) = (\delta_0^*(p,wa), \delta_1^*(q,wa))$. ■

Now we can prove the correctness of the product construction.

Lemma 3.2. *Let $A_0 = (Q_0, \delta_0, F_0, q_0)$ and $A_1 = (Q_1, \delta_1, F_1, q_1)$ be DFAs and let $A = (Q, \delta, F, (q_0, q_1))$ be the product automaton for the difference of A_0 and A_1. Then $L(A) = L(A_0) \setminus L(A_1)$.*

Proof. The language of A is precisely the words w such that $\delta^*((q_0,q_1),w) \in F$. Lemma 3.1 states that:

$$\delta^*((q_0,q_1),w) = (\delta_0^*(q_0,w), \delta_1^*(q_1,w))$$

From the definition of F we know that $(\delta_0^*(q_0,w), \delta_1^*(q_1,w)) \in F$ precisely when $\delta_0^*(q_0,w) \in F_0$ and $\delta_1^*(q_1,w) \notin F_1$. Finally, because $\{w \mid \delta_0^*(q_0,w) \in F_0\} = L(A_0)$ and $\{w \mid \delta_1^*(q_1,w) \in F_1\} = L(A_1)$ we conclude that $L(A) = L(A_0) \setminus L(A_1)$. ■

Lemma 2.16 states that we can obtain an expression for A by computing the least solution of A. We call this expression $\hat{f}$, because it accepts exactly the language we desired. Therefore the least solution of A gives us the expression $\hat{f}$.

3.3 Equivalence of Removal Operation

Now that we have constructed the expression $\hat{f}$, we have to prove that $\mathsf{KA}, e = 0 \vdash f = \hat{f}$. We divide the proof into two parts, in the first part we prove that $\hat{f} \leqq f$ and in the second part we prove that $\mathsf{KA}, e = 0 \vdash f \leq \hat{f}$. Combining the results proves that $\mathsf{KA}, e = 0 \vdash f = \hat{f}$.

We first prove that $\hat{f} \leqq f$. Note that we do not use the hypothesis $e = 0$ because $\hat{f}$ is created by "removing" things from f. Therefore, it is to be expected that $\hat{f} \leqq f$ regardless of what is "removed". We prove this by proving that the least solution to A_f can be used to construct a solution for A. Let $s(p)$ be the least solution to A_f and let $s'(p,q) = s(p)$, where (p,q) is a state in A and p is a state in A_f. The following lemma then holds.

Lemma 3.3. *If $s(p)$ is a least solution to A_f and $s'(p,q) = s(p)$ then s' is a solution to A.*

Proof. To prove that $s'(p,q)$ is a solution to A we must prove that it satisfies the equivalence from Definition 2.15:

$$[(p,q) \in F] + \sum_{(p,q) \xrightarrow{a}_{\delta} (p',q')} a \cdot s'(p',q') \leqq s'(p,q)$$

Due to Lemma 2.2 we can split the proof into two separate steps:

$$[(p,q) \in F] \leqq s'(p,q) \qquad \text{and} \qquad \sum_{(p,q) \xrightarrow{a}_{\delta} (p',q')} a \cdot s'(p',q') \leqq s'(p,q)$$

The equivalence on the left can be proven as follows. If $(p,q) \notin F$ then $[(p,q) \in F] = 0 \leqq s'(p,q)$ and if $(p,q) \in F$ then $p \in F_f$ by the definition of F and therefore $[(p,q) \in F] \leqq [p \in F_f]$. Finally, because s is a solution we know that $[p \in F_f] \leqq s(p) = s'(p,q)$ and therefore $[(p,q) \in F] \leqq s'(p,q)$.

The equivalence noted on the right can be proven as follows. Because $s(p)$ is a solution to A_f we know it satisfies the equivalence from Definition 2.15:

$$[p \in F_f] + \sum_{p \xrightarrow{a}_{\delta_f} p'} a \cdot s(p') \leqq s(p)$$

Due to Lemma 2.2 we know that for all transitions $p \xrightarrow{a}_{\delta_f} p'$:

$$a \cdot s(p') \leqq s(p)$$

The next step is to show that the above also holds for all transitions $(p,q) \xrightarrow{a}_{\delta} (p',q')$. From the definition of δ we know the following:

$$\{(a,p') \mid \delta((p,q),a) = (p',q')\} \subseteq \{(a,p') \mid \delta_f(p,a) = p'\}$$

The equivalence $a \cdot s(p') \leqq s(p)$ holds for all transitions $p \xrightarrow{a}_{\delta_f} p'$ and $(p,q) \xrightarrow{a}_{\delta} (p',q')$ is a subset of those transitions when we only consider p and p'. Therefore, we know that for all $(p,q) \xrightarrow{a}_{\delta} (p',q')$ we have $a \cdot s(p') \leqq s(p)$. Using the definition of $s'(p,q)$ we obtain that for all q and q':

$$a \cdot s'(p',q') \leqq s'(p,q)$$

Lemma 2.2 then gives us:

$$\sum_{(p,q) \xrightarrow{a}_{\delta} (p',q')} a \cdot s'(p',q') \leqq s'(p,q)$$

Finally, because both equivalences hold we have proved that $s'(p,q)$ is a solution to A. ■

Now we can prove our desired result using Lemma 3.3.

Lemma 3.4. *For any expression f and the corresponding expression $\hat{f}$ it holds that $\hat{f} \leqq f$.*

Proof. Let s be the least solution to A_f. Lemma 3.3 then states that s' is a solution to A. Furthermore, we know that $t \leqq s'$ where t is the least solution to A. More specifically, it follows that $t(q_f, q_e') \leqq s'(q_f, q_e')$ and consequently that $t(q_f, q_e') \leqq s(q_f)$. Finally, because both t and s are least solutions to the automata for $\hat{f}$ and f respectively, it follows from Theorem 2.18 that $\hat{f} \leqq f$. ■

Now we prove that $\mathsf{KA}, e = 0 \vdash f \leq \hat{f}$. Note that this time we do need the hypothesis $e = 0$. We accomplish this by proving an equivalence about the least solutions of A_f, A'_e and A, which we denote with s_f, s'_e and $\hat{s}$ respectively. Concretely we prove that $\hat{s}(p,q) + s'_e(q) \geqq s_f(p)$. Intuitively this should be true because we created A by "removing" A'_e from A_f, therefore if we "add" back A'_e, the result should contain everything from A_f. Then, because of Theorem 2.18 and our hypothesis $e = 0$ we deduce that $\mathsf{KA}, e = 0 \vdash f \leq \hat{f}$. We start by proving the following lemma.

Lemma 3.5. *Let $\hat{s}(p,q)$ be the least solution to A, let $s_f(p)$ be the least solution to A_f and let $s'_e(q)$ be the least solution to A'_e. Then $\hat{s}(p,q) + s'_e(q) \geqq s_f(p)$.*

Proof. Due to Theorem 2.19 we know that if the lemma is true for the languages corresponding to the solutions it is provable in KA. Therefore it suffices to prove the following:

$$L(\hat{s}(p,q)) \cup L(s'_e(q)) \supseteq L(s_f(p))$$

Lemma 2.16 states that the language of the least solution of a state is the language of that state. Therefore we can prove the above by reasoning about the languages of states. The language $L(\hat{s}(p,q))$ is precisely the words w where $\delta^*((p,q),w) \in F$. Due to Lemma 3.1 and the definition of F we know this is when $\delta^*_f(p,w) \in F_f$ and $\delta^*_{e'}(q,w) \notin F'_e$. For every word $w \in L(s_f(p))$ we know that $\delta^*_f(p,w) \in F_f$. Then either $\delta^*_{e'}(q,w) \in F'_e$ and therefore $w \in L(s'_e(q))$ or $\delta^*_{e'}(q,w) \notin F'_e$ so $\delta^*((p,q),w) \in F$ and $w \in L(\hat{s}(p,q))$. In both cases the lemma holds. ■

Now we are ready to prove our desired lemma.

Lemma 3.6. *For any $f \in \mathsf{Exp}$ and the corresponding $\hat{f}$ it holds that $\mathsf{KA}, e = 0 \vdash f \leq \hat{f}$.*

Proof. Theorem 2.18 states that the least solution of the first state of an automaton is equivalent to the expression for that automaton. Therefore we know that $s'_e(q'_e) \equiv e'$. Combined with the hypothesis $e = 0$ we can deduce that $\mathsf{KA}, e = 0 \vdash s'_e(q'_e) = 0$. Then, Lemma 3.5 states that $\hat{s}(p,q) + s'_e(q) \geqq s_f(p)$ and specifically for $p = q_f$ and $q = q'_e$ we have $\hat{s}(q_f, q'_e) + s'_e(q'_e) \geqq s_f(q_f)$. If we then use $\mathsf{KA}, e = 0 \vdash s'_e(q'_e) = 0$ we obtain $\mathsf{KA}, e = 0 \vdash \hat{s}(q_f, q'_e) + 0 \geq s_f(q_f)$ and therefore $\mathsf{KA}, e = 0 \vdash \hat{s}(q_f, q'_e) \geq s_f(q_f)$. Finally, because q_f and q'_e are the initial states states and $\hat{s}$ and s_f are least solutions, we conclude from Theorem 2.18 that $\mathsf{KA}, e = 0 \vdash f \leq \hat{f}$. ■

Combining Lemma 3.4 and Lemma 3.6 yields the following lemma.

Lemma 3.7. *For any expression f and the corresponding $\hat{f}$ it holds that $\mathsf{KA}, e = 0 \vdash f = \hat{f}$.*

This concludes the first part of our proof.

3.4 Construction of Corresponding KA

Now we proceed with the second part of the proof. Our aim is to create a KA K_e and interpretation σ_e such that $[\![e]\!]_{\sigma_e} = [\![0]\!]_{\sigma_e}$ and $[\![f]\!]_{\sigma_e} = [\![\hat{f}]\!]$.

Let $K_e = (2^{\Sigma^*}, \cup', \cdot', \emptyset, \{\epsilon \mid \text{if } \epsilon \notin [\![e]\!]\}, (-)^{*'})$. The interpretation of expressions is defined inductively by the function $[\![-]\!]_{\sigma_e} : \mathsf{Exp} \to 2^{\Sigma^*}$ as follows:

$$[\![a]\!]_{\sigma_e} := \{a\} \setminus [\![e']\!] \qquad [\![0]\!]_{\sigma_e} := \emptyset \qquad [\![1]\!]_{\sigma_e} := \{\epsilon \mid \text{if } \epsilon \notin [\![e]\!]\}$$
$$[\![f+g]\!]_{\sigma_e} := [\![f]\!]_{\sigma_e} \cup' [\![g]\!]_{\sigma_e} \qquad [\![f \cdot g]\!]_{\sigma_e} := [\![f]\!]_{\sigma_e} \cdot' [\![g]\!]_{\sigma_e} \qquad [\![f^*]\!]_{\sigma_e} := [\![f]\!]^{*'}_{\sigma_e}$$

where the operations $\cup'$, $\cdot'$ and $(-)^{*'}$ are defined as follows:

$$L_1 \cup' L_2 := (L_1 \cup L_2) \setminus \llbracket e' \rrbracket \qquad L_1 \cdot' L_2 := (L_1 \cdot L_2) \setminus \llbracket e' \rrbracket \qquad L_1^{*'} := L_1^* \setminus \llbracket e' \rrbracket$$

Before we can prove a connection between interpretations by σ_e and interpretations in the language model we have to prove the following lemma.

Lemma 3.8. *For any two languages L_1 and L_2 the following holds:*

$$((L_1 \setminus \llbracket e' \rrbracket) \cdot (L_2 \setminus \llbracket e' \rrbracket)) \setminus \llbracket e' \rrbracket = (L_1 \cdot L_2) \setminus \llbracket e' \rrbracket$$

Proof. We prove the lemma with a double inclusion starting with:

$$((L_1 \setminus \llbracket e' \rrbracket) \cdot (L_2 \setminus \llbracket e' \rrbracket)) \setminus \llbracket e' \rrbracket \supseteq (L_1 \cdot L_2) \setminus \llbracket e' \rrbracket$$

For every $x \in (L_1 \cdot L_2) \setminus \llbracket e' \rrbracket$ let $x = x_1 \cdot x_2$ such that $x_1 \in L_1$ and $x_2 \in L_2$. We know the following statements are true:

$$x \notin \llbracket e' \rrbracket \qquad \llbracket e' \rrbracket = \llbracket e' \rrbracket \cdot \llbracket \Sigma^* \rrbracket \qquad \llbracket e' \rrbracket = \llbracket \Sigma^* \rrbracket \cdot \llbracket e' \rrbracket$$

If $x_1 \in \llbracket e' \rrbracket$ then $x \in \llbracket e' \rrbracket$ because $x_2 \in \llbracket \Sigma^* \rrbracket$ and $\llbracket e' \rrbracket = \llbracket e' \rrbracket \cdot \llbracket \Sigma^* \rrbracket$. This is a contradiction and therefore $x_1 \notin \llbracket e' \rrbracket$ and $x_1 \in L_1 \setminus \llbracket e' \rrbracket$. In an analogous manner we have that $x_2 \in L_2 \setminus \llbracket e' \rrbracket$. Then we deduce that $x \in (L_1 \setminus \llbracket e' \rrbracket) \cdot (L_2 \setminus \llbracket e' \rrbracket)$. Finally because $x \notin \llbracket e' \rrbracket$ we know that:

$$x \in ((L_1 \setminus \llbracket e' \rrbracket) \cdot (L_2 \setminus \llbracket e' \rrbracket)) \setminus \llbracket e' \rrbracket$$

Therefore we have proved that:

$$((L_1 \setminus \llbracket e' \rrbracket) \cdot (L_2 \setminus \llbracket e' \rrbracket)) \setminus \llbracket e' \rrbracket \supseteq (L_1 \cdot L_2) \setminus \llbracket e' \rrbracket$$

The other inclusion is proved as follows. Because $L_1 \setminus \llbracket e' \rrbracket \subseteq L_1$ and $L_2 \setminus \llbracket e' \rrbracket \subseteq L_2$ we know that:

$$(L_1 \setminus \llbracket e' \rrbracket) \cdot (L_2 \setminus \llbracket e' \rrbracket) \subseteq L_1 \cdot L_2$$

and therefore:

$$((L_1 \setminus \llbracket e' \rrbracket) \cdot (L_2 \setminus \llbracket e' \rrbracket)) \setminus \llbracket e' \rrbracket \subseteq (L_1 \cdot L_2) \setminus \llbracket e' \rrbracket$$

Consequently, the lemma holds. ■

Now we can prove the following lemma about the relation between an interpretation by σ_e and an interpretation in the language model.

Lemma 3.9. *For all expressions f it holds that $\llbracket f \rrbracket_{\sigma_e} = \llbracket f \rrbracket \setminus \llbracket e' \rrbracket$.*

Proof. We prove the lemma by induction on the structure of expressions. As a base case this holds for a letter a and for 1 by the definitions of their interpretations. For 0 it holds because $\emptyset \setminus \llbracket e' \rrbracket = \emptyset$.

Assume as induction hypothesis that the lemma holds for two expressions f and g. Then $f+g$ results in:

$$\begin{aligned}
[\![f+g]\!]_{\sigma_e} &= [\![f]\!]_{\sigma_e} \cup' [\![g]\!]_{\sigma_e} && \text{Definition } [\![-]\!]_{\sigma_e} \\
&= ([\![f]\!]_{\sigma_e} \cup [\![g]\!]_{\sigma_e}) \setminus [\![e']\!] && \text{Definition } \cup' \\
&= (([\![f]\!] \setminus [\![e']\!]) \cup ([\![g]\!] \setminus [\![e']\!])) \setminus [\![e']\!] && \text{Induction Hypothesis} \\
&= (([\![f]\!] \cup [\![g]\!]) \setminus [\![e']\!]) \setminus [\![e']\!] && (A \setminus C) \cup (B \setminus C) = (A \cup B) \setminus C \\
&= ([\![f]\!] \cup [\![g]\!]) \setminus [\![e']\!] && (A \setminus C) \setminus C = A \setminus C \\
&= [\![f+g]\!] \setminus [\![e']\!] && \text{Definition 2.3}
\end{aligned}$$

For $f \cdot g$ it follows that:

$$\begin{aligned}
[\![f \cdot g]\!]_{\sigma_e} &= [\![f]\!]_{\sigma_e} \cdot' [\![g]\!]_{\sigma_e} && \text{Definition } [\![-]\!]_{\sigma_e} \\
&= ([\![f]\!]_{\sigma_e} \cdot [\![g]\!]_{\sigma_e}) \setminus [\![e']\!] && \text{Definition } \cdot' \\
&= (([\![f]\!] \setminus [\![e']\!]) \cdot ([\![g]\!] \setminus [\![e']\!])) \setminus [\![e']\!] && \text{Induction Hypothesis} \\
&= ([\![f]\!] \cdot [\![g]\!]) \setminus [\![e']\!] && \text{Lemma 3.8} \\
&= [\![f \cdot g]\!] \setminus [\![e']\!] && \text{Definition 2.3}
\end{aligned}$$

For f^* it holds that:

$$\begin{aligned}
[\![f^*]\!]_{\sigma_e} &= [\![f]\!]_{\sigma_e}^{*'} && \text{Definition } [\![-]\!]_{\sigma_e} \\
&= [\![f]\!]_{\sigma_e}^{*} \setminus [\![e']\!] && \text{Definition } (-)^{*'} \\
&= ([\![f]\!] \setminus [\![e']\!])^* \setminus [\![e']\!] && \text{Induction Hypothesis} \\
&= \left(\bigcup_{n \geq 0} ([\![f]\!] \setminus [\![e']\!])^n \right) \setminus [\![e']\!] && \text{Definition } L^* \\
&= \bigcup_{n \geq 0} (([\![f]\!] \setminus [\![e']\!])^n \setminus [\![e']\!]) && (\textstyle\bigcup_{X \in S} X) \setminus A = \textstyle\bigcup_{X \in S} (X \setminus A) \\
&= \bigcup_{n \geq 0} ([\![f]\!]^n \setminus [\![e']\!]) && \dagger \\
&= \left(\bigcup_{n \geq 0} [\![f]\!]^n \right) \setminus [\![e']\!] && (\textstyle\bigcup_{X \in S} X) \setminus A = \textstyle\bigcup_{X \in S} (X \setminus A) \\
&= [\![f^*]\!] \setminus [\![e']\!] && \text{Definition } L^*
\end{aligned}$$

The line marked with † follows from applying Lemma 3.8, n times in each term of the union. ■

From Lemma 3.9 we can conclude that $[\![e]\!]_{\sigma_e} = [\![e]\!] \setminus [\![e']\!] = [\![0]\!]_{\sigma_e}$. Now all that is left to prove is that K_e is a KA.

Lemma 3.10. *The definition of K_e is sound for the axioms of* KA.

Proof. For K_e to be sound it must be that for all axioms $f = g$ of KA we have $[\![f]\!]_{\sigma_e} = [\![g]\!]_{\sigma_e}$, by Lemma 3.9 this is the same as $[\![f]\!] \setminus [\![e']\!] = [\![g]\!] \setminus [\![e']\!]$. Then because $f = g$ is an axiom of KA and because the language model is sound we know that $[\![f]\!] = [\![g]\!]$. Finally, because $A = B \implies A \setminus C = B \setminus C$ holds for all sets A, B and C, the equation holds. Therefore K_e is a KA. ■

3.5 Completeness and Decidability

Now we are finally equipped to prove the completeness of KA under the hypothesis $e = 0$ with respect to the model K_e and interpretation σ_e.

Theorem 3.11. *For all expressions f and g, if $K_e, \sigma_e \models f = g$ then $\mathsf{KA}, e = 0 \vdash f = g$.*

Proof. Let f and g be expressions such that $K_e, \sigma_e \models f = g$, then we have:

$$\begin{aligned} [\![f]\!]_{\sigma_e} &= [\![g]\!]_{\sigma_e} \\ [\![f]\!] \setminus [\![e']\!] &= [\![g]\!] \setminus [\![e']\!] && \text{Lemma 3.9} \\ [\![\hat{f}]\!] &= [\![\hat{g}]\!] && \text{Lemma 3.2} \end{aligned}$$

Using Theorem 2.19 we know that $\hat{f} \equiv \hat{g}$. Lemma 3.7 states that $\mathsf{KA}, e = 0 \vdash f = \hat{f}$ and $\mathsf{KA}, e = 0 \vdash g = \hat{g}$. Using this we obtain: $\mathsf{KA}, e = 0 \vdash f = g$. Therefore we have proved the completeness of KA under the hypothesis $e = 0$ with respect to K_e and σ_e. ■

The inverse of Theorem 3.11 is also true.

Lemma 3.12. *For all expressions f and g, if $\mathsf{KA}, e = 0 \vdash f = g$ then $K_e, \sigma_e \models f = g$.*

Proof. Let f and g be expressions such that $\mathsf{KA}, e = 0 \vdash f = g$. Then, by definition of being sound it must be that in every model X with interpretation σ_X where $[\![e]\!]_{\sigma_X} = [\![0]\!]_{\sigma_X}$ we also have $[\![f]\!]_{\sigma_X} = [\![g]\!]_{\sigma_X}$. Lemma 3.10 and Lemma 3.9 show that K_e is a KA where $[\![e]\!]_{\sigma_e} = [\![0]\!]_{\sigma_e}$. Therefore we know that $[\![f]\!]_{\sigma_e} = [\![g]\!]_{\sigma_e}$. ■

Theorem 3.11 and Lemma 3.12 together prove that provability in KA under the hypothesis $e = 0$ is characterised by equality in the model K_e and interpretation σ_e. In other words $\mathsf{KA}, e = 0 \vdash f = g$ if and only if $[\![f]\!]_{\sigma_e} = [\![g]\!]_{\sigma_e}$. Furthermore we showed that $[\![f]\!]_{\sigma_e} = [\![g]\!]_{\sigma_e}$ if and only if $[\![\hat{f}]\!] = [\![\hat{g}]\!]$. Therefore we have reduced the decidability of KA under the hypothesis $e = 0$ to the decidability of equality of the languages of two expressions, which is decidable [RS59]. Consequently, we have proved the following theorem.

Theorem 3.13. KA *is decidable under the hypothesis $e = 0$.*

Concretely we can check whether two expressions f and g are probably equivalent in KA under the hypothesis $e = 0$ by constructing the automata for $\hat{f}$ and $\hat{g}$ and checking whether the automata accept the same language.

4 Decidability of GKAT under $e = 0$

Now we know the proof strategy is valid for KA we can adapt it to GKAT. We start by constructing an expression $\hat{f}$ that "removes" words in e from f. This requires some extra work because, in contrast to KA, no product construction is known for GKAT and there does not exist a GKAT automaton that accepts e'. We then show that $\mathsf{KA}, e = 0 \nvdash f = \hat{f}$ with a counter model.

4.1 Removal Operation for GKAT

We first create a construction for a GKAT automaton that accepts the difference of two GKAT automata. Subsequently we adapt this construction to make a construction for the automaton of $\hat{f}$.

Let $A_f = (Q_f, \delta_f, \iota_f)$ and $A_e = (Q_e, \delta_e, \iota_e)$ be GKAT automata. Then $A = (Q, \delta, (\iota_f, \iota_e))$ is a GKAT automaton such that $L(A) = L(A_f) \setminus L(A_e)$, where we define Q and δ as follows:

$$Q = Q_f \times (Q_e + q_0)$$

$$\delta(q_f, q_e)(\alpha) = \begin{cases} 1 & \delta_f(q_f)(\alpha) = 1 \ \wedge \ \delta_e(q_e)(\alpha) \neq 1 \\ 0 & \delta_f(q_f)(\alpha) = 0 \ \vee \ (\delta_f(q_f)(\alpha) = 1 \ \wedge \ \delta_e(q_e)(\alpha) = 1) \\ (p_1, (q'_f, q_0)) & (\delta_f(q_f)(\alpha) = (p_1, q'_f) \ \wedge \ \delta_e(q_e)(\alpha) \in 2) \ \vee \\ & (\delta_f(q_f)(\alpha) = (p_1, q'_f) \ \wedge \ \delta_e(q_e)(\alpha) = (p_2, q'_e) \ \wedge \ p_1 \neq p_2) \\ (p, (q'_f, q'_e)) & \delta_f(q_f)(\alpha) = (p, q'_f) \ \wedge \ \delta_e(q_e)(\alpha) = (p, q'_e) \end{cases}$$

The idea behind this construction is the same as rationale behind the product construction for KA. The automaton simulates both A_f and A_e by having a state space consisting of tuples of states from A_f and A_e. If both automata terminate at the same time we accept if A_f accepts and A_e rejects, otherwise we reject. One crucial difference between KA automata and GKAT automata is how they terminate. For GKAT automata termination occurs via an explicit transition while for KA automata termination only occurs when the entire string has been read. In the case that A_e has rejected the input but A_f has not terminated yet, we do not know whether the string is accepted by A. The states $Q_f \times \{q_0\}$ are added specifically to resolve this. We assume that $q_0 \notin Q_e$ and we define $\delta_e(q_0)(\alpha) = 0$ for the transition function of A. Then if A_e rejects and A_f has not yet terminated we transition to some state (q, q_0). In such a state our transition function is fully determined by the transition function for A_f.

If both A_f and A_e make a transition the result depends on the corresponding actions. If the action on the transition of A_f does not correspond to the action in the input, we reject. The reason for this is that in both cases that A transitions, it does so with the action from A_f. Therefore if the action does not match with A_f, it does not match with the transition in A and we reject. If the actions do correspond, the result is determined by whether the actions also correspond to the action on the transition of A_e. If the action on the transition of A_e differs, the input is not accepted by A_e and therefore we transition to a state in $Q_f \times \{q_0\}$. If all actions match we transition to a state given by both their transition functions. We illustrate this construction in the following example.

Example 4.1 (Product Construction GKAT)**.** We execute the GKAT product construction on the following two automata:

$$\begin{tikzcd}
{} & {} & q_3 \arrow[dd, "\beta|p_2"] \\
{} \arrow[r] & q_1 \arrow[ru, "\beta|p_2"] \arrow[rd, "\alpha|p_1"'] & \\
{} & {} & q_2 \arrow[loop right, "\beta|p_2"] \arrow[d, Rightarrow] \\
{} & {} & \alpha
\end{tikzcd}
\qquad
\begin{tikzcd}
{} \arrow[r] & q_a \arrow[r, "\alpha|p_1"] & q_b \arrow[loop right, "\beta|p_1"] \arrow[d, Rightarrow] \\
{} & {} & \alpha
\end{tikzcd}$$

We call the left automaton A_f and the right automaton A_e. The automaton below then accepts $L(A_f) \setminus L(A_e)$.

$$\begin{tikzcd}
 & & (q_3, q_0) \arrow[r, "\beta|p_2"] & (q_2, q_0) \arrow[loop above, "\beta|p_2"] \arrow[r, Rightarrow] & \alpha \\
{} \arrow[r] & (q_1, q_a) \arrow[ur, "\beta|p_2"] \arrow[dr, "\alpha|p_1"'] & & & \\
 & & (q_2, q_b) \arrow[uur, "\beta|p_2"'] & &
\end{tikzcd}$$

This automaton is created as follows. We start with the initial state (q_1, q_a). Then for every atom we compare the transitions of A_f and A_e. For (q_1, q_a) and β we get $\delta_f(q_1)(\beta) = (p_2, q_3)$ and $\delta_e(q_a)(\beta) = 0$. Because A_e has terminated unsuccessfully we only have to keep simulating A_f and therefore we transition to (q_3, q_0). Given α we have $\delta_f(q_1)(\alpha) = (p_1, q_2)$ and $\delta_e(q_1)(\alpha) = (p_1, q_b)$. Both automata transition with the same action, thus we take both their transitions and go to (q_2, q_b). In the case of (q_2, q_b) and α it follows that $\delta_f(q_2)(\alpha) = 1$ and $\delta_e(q_b)(\alpha) = 1$. As both automata accept, we reject the input. The atom β results in $\delta_f(q_2)(\beta) = (p_2, q_2)$ and $\delta_e(q_b)(\beta) = (p_1, q_b)$. The actions on the transitions do not match, consequently we only have to keep simulating A_f and we transition to (q_2, q_0) with the action from A_f. For (q_3, q_0) and β we get $\delta_f(q_3)(\beta) = (p_2, q_2)$ and by definition $\delta_e(q_0)(\beta) = 0$, therefore we go to (q_2, q_0) with p_2. Then for α we get $\delta_f(q_3)(\alpha) = 0$ and therefore we reject. Finally, for (q_2, q_0) and α we obtain $\delta_f(q_2)(\alpha) = 1$ and by definition $\delta_e(q_0)(\alpha) = 0$, therefore we accept. Then, β gives us $\delta_f(q_2)(\beta) = (p_2, q_2)$ and by definition $\delta_e(q_0)(\beta) = 0$, therefore we transition to (q_2, q_0) with p_2.

For KA we first created automata for f and $\Sigma^* e \Sigma^*$ and then used the product construction to construct an automaton that accepts $[\![f]\!] \setminus [\![\Sigma^* e \Sigma^*]\!]$. This approach cannot be adapted to GKAT because the KA expression $\Sigma^* e \Sigma^*$ does not have a corresponding GKAT expression. To resolve this problem we directly define a construction for an automaton that accepts all words from f that do not contain a guarded subword that is in e, where a guarded subword is any subword that is itself a guarded word. This approach yields an automaton for precisely the language we want but only requires an automaton for f and e, not for $\Sigma^* e \Sigma^*$. The construction is based on the product construction we introduced previously and the subset construction for KA automata.

This construction is defined as follows. Let $A_f = (Q_f, \delta_f, \iota_f)$ be a GKAT automaton for f and let $A_e = (Q_e, \delta_e, \iota_e)$ be a GKAT automaton for e. Then $A = (Q, \delta, (\iota_f, \{\iota_e\}))$ is a GKAT automaton that accepts the words accepted by f that do not contain a guarded subword that is accepted by e. We define Q and δ as follows:

$$Q = Q_f \times \{S \mid \iota_e \in S \wedge S \subseteq Q_e\}$$

$$\delta(q_f, S)(\alpha) = \begin{cases} 1 & \delta_f(q_f)(\alpha) = 1 \;\wedge\; 1 \notin \{\delta_e(q_e)(\alpha) \mid q_e \in S\} \\ 0 & \delta_f(q_f)(\alpha) = 0 \;\vee\; 1 \in \{\delta_e(q_e)(\alpha) \mid q_e \in S\} \\ (p, (q_f', S')) & \delta_f(q_f)(\alpha) = (p, q_f') \;\wedge \\ & S' = \{\iota_e\} \;\cup\; \{q_e' \mid q_e \in S \;\wedge\; \delta_e(q_e)(\alpha) = (p, q_e')\} \end{cases}$$

In the initial state we have $\iota_e \in S$ and for any transition to (q', S') it holds that $\iota_e \in S'$ by definition. Therefore the automaton always stays within its defined state space.

The states are tuples where the first element is a state in A_f and the second element is a set of states S from A_e. If we have read part of an input in A then S contains all possible states we can end up in after reading a suffix of this partial input in A_e. If a state in S accepts a suffix of the partial input in A_e then the input contains a subword accepted by A_e. Therefore the input will be rejected by A. If this is not the case and the input is accepted by A_f, we accept. Finally, if no state in S accepts and A_f transitions, we take the following transitions: we take the transition in A_f, take all transition from states in S in A_e where the action matches with the action from A_f and we start again in the initial state of A_e. We illustrate this construction in the following example.

Example 4.2. We apply the construction for the automaton for $\hat{f}$ to the automata below where the left automaton is A_f and the right automaton is A_e.

$$\begin{tikzcd}
 & & q_3 \arrow[dd, "\beta|p_2"'] & \\
{} \arrow[r] & q_1 \arrow[ur, "\beta|p_2"] \arrow[dr, "\alpha|p_1"'] & & \alpha \\
 & & q_2 \arrow[r, "\alpha|p_1"] \arrow[loop below, "\beta|p_2"] & q_4 \arrow[u, Rightarrow] \arrow[loop below, "\beta|p_1"]
\end{tikzcd}
\qquad
\begin{tikzcd}
{} \arrow[r] & q_a \arrow[r, "\alpha|p_1"] & q_b \arrow[loop right, "\beta|p_2"] \arrow[d, Rightarrow] \\
 & & \alpha
\end{tikzcd}$$

This yields the following automaton:

$$\begin{tikzcd}
{} \arrow[r] & (q_1, \{a\}) \arrow[r, "\alpha|p_1"] \arrow[d, "\beta|p_2"'] & (q_2, \{a, b\}) \arrow[loop above, "\beta|p_2"] & (q_4, \{a\}) \arrow[loop above, "\beta|p_1"] \arrow[r, Rightarrow] & \alpha \\
 & (q_3, \{a\}) \arrow[r, "\beta|p_2"'] & (q_2, \{a\}) \arrow[r, "\alpha|p_1"'] \arrow[loop below, "\beta|p_2"] & (q_4, \{a, b\}) \arrow[u, "\beta|p_1"'] &
\end{tikzcd}$$

This automaton is constructed as follows. For every state and atom we compare both transition functions. Following a transition we always start over in q_a. We start in the initial state $(q_1, \{q_a\})$.

For β we get $\delta_f(q_1)(\beta) = (p_2, q_3)$ and $\delta_e(q_a)(\beta) = 0$, therefore we only take the transition from A_f and transition to $(q_3, \{q_a\})$. Given α we have $\delta_f(q_1)(\alpha) = (p_1, q_2)$ and $\delta_e(q_a)(\alpha) = (p_1, q_b)$, because the actions match we take both transitions and go to $(q_2, \{q_a, q_b\})$. In the case of $(q_3, \{q_a\})$ and β we obtain $\delta_f(q_3)(\beta) = (p_2, q_2)$ and $\delta_e(q_a)(\beta) = 0$, consequently we only take the transition from A_f and go to $(q_2, \{q_a\})$. Then, for α we get $\delta_f(q_3)(\alpha) = 0$, therefore we reject. With $(q_2, \{q_a, q_b\})$ and β we have $\delta_f(q_2)(\beta) = (p_2, q_2)$, $\delta_e(q_a)(\beta) = 0$ and $\delta_e(q_b)(\beta) = (p_2, q_b)$, the action from q_b matches and as a result we transition to $(q_2, \{q_a, q_b\})$. Given α we get $\delta_e(q_b)(\alpha) = 1$ and therefore we reject. For $(q_2, \{q_a\})$ and β we obtain $\delta_f(q_2)(\beta) = (p_2, q_2)$ and $\delta_e(q_a)(\beta) = 0$, consequently we take the transition from q_2 and go to $(q_2, \{q_a\})$. Then, α gives us $\delta_f(q_2)(\alpha) = (p_1, q_4)$ and $\delta_e(q_a)(\alpha) = (p_1, q_b)$.

Because the actions match we transition to $(q_4, \{q_a, q_b\})$ with p_1. From $(q_4, \{q_a, q_b\})$ and β it follows that $\delta_f(q_4)(\beta) = (p_1, q_4)$, $\delta_e(q_a)(\beta) = 0$ and $\delta_e(q_b)(\beta) = (p_2, q_b)$. The actions do not match, therefore we only take the transition from A_f and go to $(q_4, \{q_a\})$ with p_1. The atom α gives us $\delta_e(q_b)(\alpha) = 1$ and as a result we reject. For $(q_4, \{q_a\})$ and β we have $\delta_f(q_4)(\beta) = (p_1, q_4)$ and $\delta_e(q_a)(\beta) = 0$ and therefore we go to $(q_4, \{q_a\})$. Finally, for α we obtain $\delta_f(q_4)(\alpha) = 1$ and $\delta_e(q_a)(\alpha) = (p_1, q_b)$. Because only A_f accepts, we accept.

We prove the correctness of the construction in the following lemma.

Lemma 4.3. *The* GKAT *automaton A accepts all the words from A_f that do not contain a guarded subword accepted by A_e*

Proof. We prove the lemma by induction on the length of guarded words. We use the following strengthened induction hypothesis. The words in the language of a state (q, S) in A are exactly the words w such that:

1. w is accepted by state q in A_f
2. w is not accepted by any $q_e \in S$ in A_e
3. w does not contain a guarded subword that is accepted by A_e

The second property of the induction hypothesis is needed to prove the third property but not needed for the lemma itself. We start by proving that all words w accepted by a state (q, S) in A satisfy the three criteria.

As a base case we take the guarded word α. We know α is accepted by state (q, S) if and only if $\delta(q, S)(\alpha) = 1$. This is the case when $\delta_f(q)(\alpha) = 1 \;\wedge\; 1 \notin \{\delta_e(q_e)(\alpha) \mid q_e \in S\}$. The first requirement is satisfied because $\delta_f(q)(\alpha) = 1$ states that α is accepted by q in A_f.

The right side of the conjunction, $1 \notin \{\delta_e(q_e)(\alpha) \mid q_e \in S\}$, states that α is not accepted by any state $q_e \in S$ in A_e. Therefore the second requirement is satisfied.

In particular ι_e is always in S and therefore the state ι_e does not accept α. Because the strings accepted by ι_e are exactly the strings accepted by A_e, α is not accepted by A_e. Finally, because α is the only guarded subword of α, α contains no guarded subwords that are accepted by A_e. Therefore the third requirement is satisfied.

Assume as induction hypothesis that the three properties hold for all states (q, S) in A and words w with length less than k. The word $\alpha p w$, where $|w| < k$, is accepted by state (q, S) if and only if $\delta_f(q)(\alpha) = (p, q') \;\wedge\; S' = \{\iota_e\} \,\cup \{q'_e \mid q_e \in S \;\wedge\; \delta_f(q_e)(\alpha) = (p, q'_e)\}$ and w is accepted by (q', S').

The induction hypothesis states that w is accepted by q' in A_f. Combined with $\delta_f(q)(\alpha) = (p, q')$ we can conclude that $\alpha p w$ is accepted by q in A_f. Therefore the first property is satisfied.

If there is a $q_e \in S$ in A_e that accepts $\alpha p w$ then there must be a transition $\delta_e(q_e)(\alpha) = (p, q'_e)$ where q'_e accepts w. The definition of δ states that S' contains for every $q_x \in S$, all states q'_x where $\delta_e(q_x)(\alpha) = (p, q'_x)$. Therefore $q'_e \in S'$. The induction hypothesis shows that w is not accepted by any $q_x \in S'$ in A_e. This is a contradiction and therefore $\alpha p w$ is not accepted by any state $q_e \in S$ in A_e. Therefore the second property is satisfied.

The induction hypothesis states that no guarded subword of w is accepted by A_e. Therefore a guarded subword of $\alpha p w$ that is accepted by A_e would have to start at α. If the guarded prefix is just α itself, property three holds by the same argument as used in the base case.

If the guarded prefix is not just α there would have to be a transition from ι_e such that $\delta_e(\iota_e)(\alpha) = (p, q'_e)$ and some guarded prefix v of w is accepted by q'_e in A_e. Assume that v looks like $\alpha_0 p_0 \alpha_1 p_1 ... \alpha_{n-1} p_{n-1} \alpha_n$. We use v_k to denote the subword $\alpha_0 p_0 ... \alpha_{k-1} p_{k-1}$ and q_{e_k} to denote the state we end up in after reading $\alpha p v_k$ in A_e. Then for every $k \leq n$ we know that $\alpha_k p_k ... \alpha_{n-1} p_{n-1} \alpha_n$ is accepted by q_{e_k} in A_e. Finally we use (q_{f_k}, S_k) to denote the state in which we end after reading $\alpha p v_k$ in A, starting from (q, S).

We know that $\iota_e \in S$ and the definition of δ states that therefore $q_{e_0} \in S_0$. Assume that $q_{e_k} \in S_k$, then again the definition of δ states that $q_{e_{k+1}} \in S_{k+1}$ and therefore it holds for all k that $q_{e_k} \in S_k$.

Because $\alpha p v$ is accepted by A_e we know that $\delta_e(q_{e_n})(\alpha_n) = 1$. Then by the definition of δ and because $q_{e_n} \in S_n$ we deduce that $\delta(q_{f_n}, S_n)(\alpha_n) = 0$. Finally, because reading $\alpha p v$ in (q, S) in A results in 0, we know that $\alpha p w$ is not accepted by (q, S). Therefore the third property holds.

Now we prove the opposite direction, i.e. any guarded word that satisfies these three properties is accepted by (q, S) in A.

As a base case we take the guarded word α. Because the first property holds we know that $\delta_f(q)(\alpha) = 1$. Because the second property holds we know that for all $q_e \in S$, $\delta_e(q_e)(\alpha) \neq 1$ and therefore $1 \notin \{\delta_e(q_e)(\alpha) \mid q_e \in S\}$. The definition of δ then gives us that $\delta(q, S)(\alpha) = 1$ and therefore (q, S) accepts α.

Assume as induction hypothesis that any guarded word w, with $|w| < k$, that satisfies these three properties is accepted by (q, S) in A.

Assume that $\alpha p w$ satisfies the three properties for the state q and the set of states S. For $\alpha p w$ to be accepted by (q, S) there must be a transition $\delta(q, S)(\alpha) = (p, (q', S'))$. First we show that w must then also satisfy these properties for (q', S'). From the first property we deduce that there must be a transition $\delta_f(q)(\alpha) = (p, q')$ where w is accepted by q' in A_f. Therefore the first property holds for w and (q', S').

From the second property we know that for all transitions $\delta_e(q_e)(\alpha) = (p, q'_e)$ where $q_e \in S$, it must be that w is not accepted by q'_e in A_e. From the third property we know that w is not accepted by ι_e in A_e because w is a guarded subword of $\alpha p w$. Finally, we know the second property holds for (q', S') because $S' = \{\iota_e\} \cup \{q'_e \mid q'_e \in S \wedge \delta_e(q_e)(\alpha) = (p, q'_e)\}$. The third property immediately holds because w is a subword of $\alpha p w$. Because all three properties hold for w and (q', S') we know by our induction hypothesis that w is accepted by (q', S') in A.

We concluded that $\delta(q, S)(\alpha) = (p, (q', S'))$ where $S' = \{\iota_e\} \cup \{q'_e \mid q'_e \in S \wedge \delta_e(q_e)(\alpha) = (p, q'_e)\}$ and that w is accepted by (q', S'). Consequently, $\alpha p w$ is accepted by (q, S). Therefore we have proven that all words that satisfy the three properties are accepted by (q, S) in A.

For the initial state of A, $(\iota_f, \{\iota_e\})$, the second property simply states that no words accepted by A_e are in its language. In this case the third property implies the second property. Therefore, the language of the initial state consists exactly of the words accepted by ι_f in A_f that do not contain a guarded subword accepted by A_e. Finally, because the language of the first state is the language of the automaton, we have proved that the language of A is exactly the words from A_f that do not contain a guarded subword accepted by A_e. ■

One important thing to consider is that not all automata have solutions [KT08]. We do not know whether this construction always yields well-nested automata, i.e. solvable automata.

4.2 Non-Equivalence of Removal Operation for GKAT

For KA, the step after creating the automaton for $\hat{f}$ was to prove $\mathsf{KA}, e = 0 \vdash f = \hat{f}$. In this section we show that this does not hold for GKAT. We demonstrate this by applying our construction to some simple automata. We then derive concrete expressions f and $\hat{f}$ for which $\mathsf{GKAT}, e = 0 \nvdash f = \hat{f}$. We accomplish this by showing there is a counter model where the implication $e = 0 \implies f = \hat{f}$ does not hold.

Example 4.4. In this example we apply the automaton construction for $\hat{f}$ to the automata A_f and A_e corresponding to the expressions $f = p_1p_2$ and $e = bp_2c$. These automata are depicted below.

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[r, "1|p_1"] & q_1 \arrow[r, "1|p_2"] & q_2 \arrow[r, Rightarrow] & 1 & & {} \arrow[r] & q_0 \arrow[r, "b|p_2"] & q_1 \arrow[r, Rightarrow] & c
\end{tikzcd}$$

Applying the construction yields the following automaton A.

$$\begin{tikzcd}
(q_0, \{q_0\}) \arrow[r, "1|p_1"] & (q_1, \{q_0\}) \arrow[r, "b|p_2"] \arrow[dr, "\bar{b}|p_2"'] & (q_2, \{q_0, q_1\}) \arrow[r, Rightarrow] & \bar{c} \\
& & (q_2, \{q_0\}) \arrow[r, Rightarrow] & 1
\end{tikzcd}$$

We obtain the expression $\hat{f}$ by computing the solution s for the state $(q_0, \{q_0\})$ using the equivalence from Definition 2.31. This gives us $s(q_0, \{q_0\}) \equiv p_1(p_2\bar{c} +_b p_2)$. Our aim was to prove $\mathsf{GKAT}, e = 0 \vdash f = \hat{f}$. For the concrete expressions of the previous example this corresponds to $\mathsf{GKAT}, bp_2c = 0 \vdash p_1p_2 = p_1(p_2\bar{c} +_b p_2)$. However this equivalence is not provable. We show this with a counter model, where the implication $bp_2c = 0 \implies p_1p_2 = p_1(p_2\bar{c} +_b p_2)$ is false, in Appendix A. If a statement is provable using the axioms of GKAT then it must be true in every model. Thus, the existence of a GKAT model in which the implication does not hold shows that it cannot be provable.

Lemma 4.5. *There exists an expression f such that for f and the corresponding expression $\hat{f}$ it does not hold that:*

$$\mathsf{GKAT}, e = 0 \vdash f = \hat{f}$$

The implication $bp_2c = 0 \implies p_1p_2 = p_1(p_2\bar{c} +_b p_2)$ is, however, true in the language model.

Lemma 4.6. *For any two tests b and c and any two actions p_1 and p_2 the following is true:*

$$[\![bp_2c]\!] = [\![0]\!] \implies [\![p_1p_2]\!] = [\![p_1(p_2\bar{c} +_b p_2)]\!]$$

Proof. We first compute the interpretation of the hypothesis:

$$[\![bp_2c]\!] = [\![0]\!]$$
$$[\![b]\!] \diamond [\![p_2]\!] \diamond [\![c]\!] = \emptyset$$

We know that $[\![p_2]\!] = \{\alpha p_2 \beta \mid \alpha, \beta \in \mathsf{At}\}$. For the fusion product to result in the empty set it must therefore be that $[\![b]\!] = \emptyset$ or $[\![c]\!] = \emptyset$. We now compute the interpretation of the consequent of the implication:

$$\begin{aligned}
[\![p_1p_2]\!] &= [\![p_1(p_2\overline{c} +_b p_2)]\!] \\
[\![p_1]\!] \diamond [\![p_2]\!] &= [\![p_1]\!] \diamond [\![p_2\overline{c} +_b p_2]\!] \\
[\![p_1]\!] \diamond [\![p_2]\!] &= [\![p_1]\!] \diamond ([\![p_2\overline{c}]\!] +_{[\![b]\!]} [\![p_2]\!]) \\
[\![p_1]\!] \diamond [\![p_2]\!] &= [\![p_1]\!] \diamond (([\![b]\!] \diamond [\![p_2\overline{c}]\!]) \cup ([\![\overline{b}]\!] \diamond [\![p_2]\!])) \\
[\![p_1]\!] \diamond [\![p_2]\!] &= [\![p_1]\!] \diamond (([\![b]\!] \diamond [\![p_2]\!] \diamond [\![\overline{c}]\!]) \cup ([\![\overline{b}]\!] \diamond [\![p_2]\!]))
\end{aligned}$$

We now show that both if $[\![b]\!] = \emptyset$ and if $[\![c]\!] = \emptyset$ the consequent of the implication reduces to a tautology. If $[\![b]\!] = \emptyset$ then $[\![\overline{b}]\!] = \mathsf{At}$ and this results in:

$$\begin{aligned}
[\![p_1]\!] \diamond [\![p_2]\!] &= [\![p_1]\!] \diamond ((\emptyset \diamond [\![p_2]\!] \diamond [\![\overline{c}]\!]) \cup (\mathsf{At} \diamond [\![p_2]\!])) \\
[\![p_1]\!] \diamond [\![p_2]\!] &= [\![p_1]\!] \diamond (\emptyset \cup [\![p_2]\!]) \\
[\![p_1]\!] \diamond [\![p_2]\!] &= [\![p_1]\!] \diamond [\![p_2]\!]
\end{aligned}$$

If $[\![c]\!] = \emptyset$ then $[\![\overline{c}]\!] = \mathsf{At}$ and this results in:

$$\begin{aligned}
[\![p_1]\!] \diamond [\![p_2]\!] &= [\![p_1]\!] \diamond (([\![b]\!] \diamond [\![p_2]\!] \diamond \mathsf{At}) \cup ([\![\overline{b}]\!] \diamond [\![p_2]\!])) \\
[\![p_1]\!] \diamond [\![p_2]\!] &= [\![p_1]\!] \diamond (([\![b]\!] \diamond [\![p_2]\!]) \cup ([\![\overline{b}]\!] \diamond [\![p_2]\!])) \\
[\![p_1]\!] \diamond [\![p_2]\!] &= [\![p_1]\!] \diamond [\![p_2]\!]
\end{aligned}$$

Finally, if the hypothesis holds then either $[\![b]\!] = \emptyset$ or $[\![c]\!] = \emptyset$. In both of those cases the consequent of the implication holds, therefore the implication holds. ■

Furthermore, we can prove $\mathsf{GKAT}, bp_2c = 0 \vdash p_1p_2 = p_1(p_2\overline{c} +_b p_2)$ if we add $bec = 0 \implies be = be\overline{c}$ as an axiom.

Lemma 4.7. *If we add $bec = 0 \implies be = be\overline{c}$ as an axiom to* GKAT *then the following holds:*

$$\mathsf{GKAT}, bp_2c = 0 \vdash p_1p_2 = p_1(p_2\overline{c} +_b p_2)$$

Proof.

$$\begin{aligned}
p_1p_2 &\equiv p_1(p_2 +_b p_2) && x +_b x \equiv x \\
&\equiv p_1(bp_2 +_b p_2) && x +_b y \equiv b \cdot x +_b y \\
&\equiv p_1(bp_2\overline{c} +_b p_2) && bp_2c \equiv 0, \implies bp_2 \equiv bp_2\overline{c} \\
&\equiv p_1(p_2\overline{c} +_b p_2) && x +_b y \equiv b \cdot x +_b y
\end{aligned}$$

■

The implication $bec = 0 \implies be = be\overline{c}$ encodes the following fundamental property of Hoare triples. If a program e is executed from the precondition b and it never terminates with postcondition c, then it must be that all terminating executions of e that start with b end with the postcondition $\overline{c}$. In KAT this implication is a theorem. Furthermore, the language model remains sound if this implication is added as an axiom to GKAT.

Lemma 4.8. *For any two tests b and c and expression e, $[\![bec]\!] = [\![0]\!] \implies [\![be]\!] = [\![be\bar{c}]\!]$.*

Proof. We first compute the interpretation of the antecedent:

$$\begin{aligned} [\![bec]\!] &= [\![0]\!] \\ [\![b]\!] \diamond [\![e]\!] \diamond [\![c]\!] &= \emptyset \end{aligned}$$

For the fusion product to result in the empty set it must be that for every guarded string in $[\![e]\!]$ either the first atom does not match with any atom from $[\![b]\!]$ or the last atom does not match with any atom from $[\![c]\!]$. Therefore either $[\![b]\!] \diamond [\![e]\!] = \emptyset$ or $[\![e]\!] \diamond [\![c]\!] = \emptyset$. We now show that in both cases the consequent reduces to a tautology.

$$\begin{aligned} [\![be]\!] &= [\![be\bar{c}]\!] \\ [\![b]\!] \diamond [\![e]\!] &= [\![b]\!] \diamond [\![e]\!] \diamond [\![\bar{c}]\!] \\ [\![b]\!] \diamond [\![e]\!] &= [\![b]\!] \diamond [\![e]\!] \diamond \overline{[\![c]\!]} \end{aligned}$$

If $[\![b]\!] \diamond [\![e]\!] = \emptyset$ we obtain:

$$\begin{aligned} \emptyset &= \emptyset \diamond \overline{[\![c]\!]} \\ \emptyset &= \emptyset \end{aligned}$$

If $[\![e]\!] \diamond [\![c]\!] = \emptyset$ then we know that $X \cap [\![c]\!] = \emptyset$, where X is the set of atoms at the end of a guarded string in $[\![e]\!]$. We also know that $X \subseteq \overline{[\![c]\!]}$ because all atoms not in $\overline{[\![c]\!]}$ are in $[\![c]\!]$ and $[\![c]\!] \cap X = \emptyset$. Therefore $[\![e]\!] \diamond \overline{[\![c]\!]} = [\![e]\!]$ and we obtain:

$$\begin{aligned} [\![b]\!] \diamond [\![e]\!] &= [\![b]\!] \diamond [\![e]\!] \diamond \overline{[\![c]\!]} \\ [\![b]\!] \diamond [\![e]\!] &= [\![b]\!] \diamond [\![e]\!] \end{aligned}$$

Because in both cases the consequent reduces to a tautology the lemma holds. ∎

Finally, Lemma 2.26 states that because the axiom is sound for the language model it is also sound for the relational model.

5 Conclusions and Further Research

The initial objective of this thesis project was to prove the decidability of GKAT under the hypothesis $e = 0$. Prior to our attempt to prove this for GKAT, we proved it for KA because we know KA is decidable under the hypothesis $e = 0$ and KA is simpler to work with than GKAT. We proved this by characterising provability in KA under the hypothesis $e = 0$ as equality in a model K_e with interpretation σ_e. This characterisation consisted mainly of proving the completeness of KA under the hypothesis $e = 0$ with respect to K_e and σ_e. This proof was split into two steps. Firstly, we proved that under the hypothesis $e = 0$, an expression f is equivalent to f with e "removed". We "removed" e by creating an automaton using the product construction that accepts the words from f that do not contain a word from e. Secondly, we proved that the interpretation of an expression by σ_e is equal to a language model interpretation of the same expression with e "removed". The completeness of K_e and σ_e then follows from the completeness of the language model and the equivalence of f and f with e "removed".

We proceeded with adapting the proof to GKAT. We created a product construction and subsequently a construction for an automaton that "removes" e from f. Then we attempted to prove that under the hypothesis $e = 0$, every expression f is equivalent to f with e "removed". However, this led to a counterexample. Therefore the proof is not valid for GKAT.

We concluded the section on GKAT by exploring a possible solution to this flaw. We proposed adding a new axiom to GKAT that would allow us to defuse the counterexample. Furthermore, we proved that adding this axiom would be sound for both the language and the relational model. However, it is unclear whether adding this axiom only allows us to defuse the counterexample or if it also enables us to prove that under the hypothesis $e = 0$ every expression f is equivalent to f with e "removed".

Further research is needed determine whether this equivalence can be proved with this axiom, and if so whether the decidability of GKAT under the hypothesis $e = 0$ can be proved using our strategy. It could also address the question of the decidability of GKAT under different hypotheses, such as $S = 1$, where S is a sum of letters.

References

[AFG$^+$14] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: semantic foundations for networks. POPL '14, page 113–126. Association for Computing Machinery, 2014.

[Ard61] Dean N. Arden. Delayed-logic and finite-state machines. In *2nd Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1961)*, pages 133–151, 1961.

[CKS96] Ernie Cohen, Dexter Kozen, and Frederick Smith. The Complexity of Kleene Algebra with Tests. Technical report, 1996.

[Coh94] Ernie Cohen. Hypotheses in Kleene Algebra. 03 1994.

[DKPP19] Amina Doumane, Denis Kuperberg, Damien Pous, and Pierre Pradic. Kleene Algebra with Hypotheses. In *Foundations of Software Science and Computation Structures*, pages 207–223. Springer International Publishing, 2019.

[Gö31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38(1):173–198, December 1931.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[Hun04] Edward V. Huntington. Sets of Independent Postulates for the Algebra of Logic. *Transactions of the American Mathematical Society*, 5(3):288–309, 1904.

[Kap23] Tobias Kappé. Kleene Algebra - Lecture 4 ESSLLI 2023, 2023. https://tobias.kap.pe/esslli/lectures/lecture-4.pdf [Accessed: 13/05/2025].

[Kle56] S. C. Kleene. *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press, 1956.

[Koz94] D. Kozen. A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events. *Information and Computation*, 110(2):366–390, 1994.

[Koz96] Dexter Kozen. Kleene algebra with tests and commutativity conditions. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 14–33. Springer Berlin Heidelberg, 1996.

[Koz00] Dexter Kozen. On Hoare logic and Kleene algebra with tests. *ACM Trans. Comput. Logic*, 1(1):60–76, July 2000.

[KP00] Dexter Kozen and Maria-Cristina Patron. Certification of Compiler Optimizations Using Kleene Algebra with Tests. In *Computational Logic — CL 2000*, pages 568–582. Springer Berlin Heidelberg, 2000.

[KS97] Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In *Computer Science Logic*, pages 244–259. Springer Berlin Heidelberg, 1997.

[KSS23] Tobias Kappé, Todd Schmid, and Alexandra Silva. A Complete Inference System for Skip-free Guarded Kleene Algebra with Tests. In *Programming Languages and Systems*, pages 309–336. Springer Nature Switzerland, 2023.

[KT08] Dexter Kozen and Wei-Lung Dustin Tseng. The Böhm–Jacopini Theorem Is False, Propositionally. In *Mathematics of Program Construction*, pages 177–192. Springer Berlin Heidelberg, 2008.

[Kuz23] Stepan L. Kuznetsov. On the Complexity of Reasoning in Kleene Algebra with Commutativity Conditions. In *Theoretical Aspects of Computing – ICTAC 2023*, pages 83–99. Springer Nature Switzerland, 2023.

[McC10] W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[Pra80] V. R. Pratt. Dynamic algebras and the nature of induction. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, STOC '80, page 22–28. Association for Computing Machinery, 1980.

[RS59] Michael Rabin and Dana Scott. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*, 3:114–125, 04 1959.

[Sal66] Arto Salomaa. Two Complete Axiom Systems for the Algebra of Regular Events. *J. ACM*, 13(1):158–169, January 1966.

[SFH$^+$19] Steffen Smolka, Nate Foster, Justin Hsu, Tobias Kappé, Dexter Kozen, and Alexandra Silva. Guarded Kleene algebra with tests: verification of uninterpreted programs in nearly linear time. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[Tho68] Ken Thompson. Programming Techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

A GKAT Counter Model

We now present the counter model that proves Lemma 4.5. This counter model was found by my supervisor Tobias Kappé using Mace4 [McC10], the query used is presented in Appendix B. The query defines the axioms of GKAT (without the uniqueness axiom) and Boolean algebra and directs Mace4 to find a model where the implication $\alpha p_2 \beta = 0 \implies p_2 = p_2\overline{\beta} +_\alpha p_2$ is false. In this model the implication that proves Lemma 4.5, $\alpha p_2 \beta = 0 \implies p_1 p_2 = p_1(p_2\overline{\beta} +_\alpha p_2)$, is also false. The counter model is a GKAT with six elements. The elements x_0 through x_3 are tests. The zero element and the one element are confusingly named x_1 and x_0 respectively. The assignment $\alpha = x_0$, $p_2 = x_4$, $\beta = x_3$ and $p_1 = x_2$ results in: $x_0 \cdot x_4 \cdot x_3 = x_1 \implies x_2 \cdot x_4 = x_2 \cdot (x_4 \cdot \overline{x_3} +_{x_0} x_4)$. Evaluating the operations yields:

$$\begin{aligned}
x_0 \cdot x_4 \cdot x_3 = x_1 &\implies x_2 \cdot x_4 = x_2 \cdot (x_4 \cdot \overline{x_3} +_{x_0} x_4) \\
x_4 \cdot x_3 = x_1 &\implies x_4 = x_2 \cdot (x_4 \cdot x_2 +_{x_0} x_4) \\
x_1 = x_1 &\implies x_4 = x_2 \cdot (x_1 +_{x_0} x_4) \\
x_1 = x_1 &\implies x_4 = x_2 \cdot x_1 \\
x_1 = x_1 &\implies x_4 = x_1
\end{aligned}$$

Therefore the implication does not hold.

The model is presented in the tables below. The tables contain the output of the Mace4 query except for one difference. The operation $f + g$ is only defined when f and g are tests. The operations $f^{(b)}$ and $f +_b g$ are only defined when b is a test. The elements x_4 and x_5 are not tests and therefore not every combination of operation and elements is defined. We mark the undefined values with $-$ but the output of the Mace4 query has assigned x_0 to all these values.

x	$\overline{x}$
x_0	x_1
x_1	x_0
x_2	x_3
x_3	x_2
x_4	$-$
x_5	$-$

(a) $\overline{x}$

x	$E(x)$
x_0	x_0
x_1	x_1
x_2	x_2
x_3	x_3
x_4	x_1
x_5	x_3

(b) $E(x)$

Figure 1: Table showing the negation of elements (a) and table showing $E(x)$ (b).

(a) $f+g$

	x_0	x_1	x_2	x_3	x_4	x_5
x_0	x_0	x_0	x_0	x_0	—	—
x_1	x_0	x_1	x_2	x_3	—	—
x_2	x_0	x_2	x_2	x_0	—	—
x_3	x_0	x_3	x_0	x_3	—	—
x_4	—	—	—	—	—	—
x_5	—	—	—	—	—	—

(b) $f \cdot g$

	x_0	x_1	x_2	x_3	x_4	x_5
x_0	x_0	x_1	x_2	x_3	x_4	x_5
x_1	x_1	x_1	x_1	x_1	x_1	x_1
x_2	x_2	x_1	x_2	x_1	x_4	x_4
x_3	x_3	x_1	x_1	x_3	x_1	x_3
x_4	x_4	x_1	x_1	x_1	x_1	x_1
x_5	x_5	x_1	x_1	x_3	x_1	x_3

(c) $f^{(b)}$

	x_0	x_1	x_2	x_3	x_4	x_5
x_0	x_1	x_1	x_1	x_1	x_1	x_1
x_1	x_0	x_0	x_0	x_0	x_0	x_0
x_2	x_3	x_3	x_3	x_3	x_3	x_3
x_3	x_2	x_2	x_2	x_2	x_2	x_2
x_4	—	—	—	—	—	—
x_5	—	—	—	—	—	—

(d) $f +_b g$

	x_0	x_1	x_2	x_3	x_4	x_5
x_0, x_0	x_0	x_0	x_0	x_0	—	—
x_0, x_1	x_0	x_1	x_2	x_3	—	—
x_0, x_2	x_0	x_2	x_2	x_0	—	—
x_0, x_3	x_0	x_3	x_0	x_3	—	—
x_0, x_4	x_0	x_4	x_2	x_5	—	—
x_0, x_5	x_0	x_5	x_0	x_5	—	—
x_1, x_0	x_1	x_0	x_3	x_2	—	—
x_1, x_1	x_1	x_1	x_1	x_1	—	—
x_1, x_2	x_1	x_2	x_1	x_2	—	—
x_1, x_3	x_1	x_3	x_3	x_1	—	—
x_1, x_4	x_1	x_4	x_1	x_4	—	—
x_1, x_5	x_1	x_5	x_3	x_4	—	—
x_2, x_0	x_2	x_0	x_0	x_2	—	—
x_2, x_1	x_2	x_1	x_2	x_1	—	—
x_2, x_2	x_2	x_2	x_2	x_2	—	—
x_2, x_3	x_2	x_3	x_0	x_1	—	—
x_2, x_4	x_2	x_4	x_2	x_4	—	—
x_2, x_5	x_2	x_5	x_0	x_4	—	—
x_3, x_0	x_3	x_0	x_3	x_0	—	—
x_3, x_1	x_3	x_1	x_1	x_3	—	—
x_3, x_2	x_3	x_2	x_1	x_0	—	—
x_3, x_3	x_3	x_3	x_3	x_3	—	—
x_3, x_4	x_3	x_4	x_1	x_5	—	—
x_3, x_5	x_3	x_5	x_3	x_5	—	—
x_4, x_0	x_4	x_0	x_5	x_2	—	—
x_4, x_1	x_4	x_1	x_4	x_1	—	—
x_4, x_2	x_4	x_2	x_4	x_2	—	—
x_4, x_3	x_4	x_3	x_5	x_1	—	—
x_4, x_4	x_4	x_4	x_4	x_4	—	—
x_4, x_5	x_4	x_5	x_5	x_4	—	—
x_5, x_0	x_5	x_0	x_5	x_0	—	—
x_5, x_1	x_5	x_1	x_4	x_3	—	—
x_5, x_2	x_5	x_2	x_4	x_0	—	—
x_5, x_3	x_5	x_3	x_5	x_3	—	—
x_5, x_4	x_5	x_4	x_4	x_5	—	—
x_5, x_5	x_5	x_5	x_5	x_5	—	—

Figure 2: The values of $f+g$ (a) and the values of $f \cdot g$ (b), where the rows denote f and the columns denote g. The values of $f^{(b)}$ (c), where the rows denote b and the columns denote f. The values of $f +_b h$ (d) where the rows denote f and h, and the columns denote b.

B Counter Model Mace4 Query

```
op(400, infix_right, ";").

formulas(sos).

 test(t_zero) & test(t_one).
 test(u) & test(v) -> test(u;v).
 test(u) & test(v) -> test(u+v).
 test(u) -> test(n(u)).

 test(u) & test(v) & test(w) -> u+(v+w) = (u+v)+w.
 test(u) & test(v) -> u+v = v+u.
 test(u) & test(v) -> u;v = v;u.
 test(u) & test(v) & test(w) -> u+(v;w) = (u+v);(u+w).
 test(u) & test(v) & test(w) -> u;(v+w) = (u;v)+(u;w).

 test(u) -> u;n(u) = t_zero.
 test(u) -> u+n(u) = t_one.
 test(u) -> n(n(u)) = u.

 test(u) & test(v) -> n(u+v) = n(u);n(v).
 test(u) & test(v) -> n(u;v) = n(u)+n(v).

 test(u) -> ite(u, x, x) = x.
 test(u) -> ite(u, x, y) = ite(n(u), y, x).
 test(u) & test(v) -> ite(v, ite(u, x, y), z) = ite(u;v, x, ite(v, y, z)).
 test(u) -> ite(u, x, y) = ite(u, u;x, y).
 test(u) -> ite(u, x;z, y;z) = ite(u, x, y);z.

 (x;y);z = x;(y;z).

 t_zero;x = t_zero.
 x;t_zero = t_zero.
 t_one;x = x.
 x;t_one = x.

 test(u) -> loop(u, x) = ite(u, x;loop(u, x), t_one).
 test(u) & test(v) -> loop(u, ite(v, x, t_one)) = loop(u, v;x).

 test(empty(x)).
 test(u) -> empty(u) = u.
 test(u) -> empty(ite(u, x, y)) = u;empty(x)+n(u);empty(y).
 empty(x;y) = empty(x);empty(y).
 test(u) -> empty(loop(u, x)) = n(u).
 test(u) & empty(x) = t_zero & ite(u, x;y, z) = y -> y = loop(u, x);z.

end_of_list.

formulas(goals).

 test(u) & test(v) & u;x;v = t_zero -> x = ite(u, x;n(v), x).

end_of_list.
```