Ricardo Manuel de Oliveira Almeida

Decision Algorithms for Kleene Algebra with Tests and Hoare Logic

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
June, 2012

Ricardo Manuel de Oliveira Almeida

Decision Algorithms for Kleene Algebra with Tests and Hoare Logic

Dissertation submitted to Faculdade de Ciências da Universidade do Porto to obtain a Master's degree in Computer Science

Orientador: Sabine Babette Broda
Co-orientador: Nelma Resende Araújo Moreira

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
June, 2012

Acknowledgments

I would like to thank my supervisors, professors Nelma Moreira and Sabine Broda, for their guidance, support and dedication over the last months. I would also like to thank David Pereira for all the fruitful discussions on Kleene algebra and for his help with the *OCaml* language. My final acknowledgments go to my family for their continuous support.

Abstract

Kleene algebra with tests (KAT) is an equational system for program verification, which is the combination of Boolean algebra (BA) and Kleene algebra (KA), the algebra of regular expressions. In particular, KAT subsumes the propositional fragment of Hoare logic (PHL) which is a formal system for the specification and verification of programs, and that is currently the base of most tools for checking program correctness. Both the equational theory of KAT and the encoding of PHL in KAT are known to be decidable.

In spite of KAT's success in dealing with several software verification tasks, there are very few software applications that implement KAT's equational theory and/or provide adequate decision procedures. In this dissertation we present a new decision procedure for the equivalence of two KAT expressions based on the notion of partial derivatives. We provide some examples of proving the equivalence of two distinct programs using the procedure defined. We also introduce the notion of derivative modulo particular sets of equations. With this we extend the previous procedure for deciding PHL. We present some experimental results, including the proof of correction and the proof of safety of a program.

Contents

List of Tables

Chapter 1

Introduction

The aim of this dissertation is to study, develop and implement a decision algorithm to test the equivalence of two KAT (Kleene algebra with tests) expressions. KAT is an equational algebraic system for reasoning about programs. In particular, KAT subsumes the propositional fragment of Hoare logic (PHL), which is a formal system for the specification and verification of programs. Testing if two KAT expressions are equivalent is tantamount to prove that two programs are equivalent or that a Hoare triple is valid. Deciding the equivalence of KAT expressions is as hard as deciding the equivalence of regular expressions, i.e. PSPACE-complete [9]. In spite of KAT's success in dealing with several software verification tasks, there are very few software applications that implement KAT's equational theory and/or provide adequate decision procedures. Most of them are within (interactive) theorem provers or part of model checking systems. See [1, 13, 8] for some examples.

We start by approaching the decision problem for regular expressions, to which we dedicate Chapter 2. This chapter should be viewed as an introduction to the decision problem we wish to solve. Here we reimplement the algorithm defined by Almeida, Moreira and Reis [3]. The problem of testing the equivalence of two regular expressions is usually solved using automata. A common approach is to transform each regular expression into an equivalent nondeterministic finite automaton (NFA), transforming both automata to deterministic ones (DFA) and then either using a DFA minimization algorithm (such as Hopcroft's [15]) to make the two automata minimal and test if they are isomorphic, or directly applying an equivalence test (such as the almost linear algorithm presented by Hopcroft and Karp [14]) on the deterministic automata. One can also prove the equivalence of two regular expressions in an axiomatic fashion [16]. However, this method is not easily automatized. The approach we follow in Chapter 2

consists in developing a functional approach to the Antimirov and Mosses rewrite system for equivalence of regular expressions [6]. This approach is argued to lead to a better average-case algorithm than those based on the comparison of the equivalent deterministic finite automata, as some experimental results suggest [2].

Similarly to what happens with regular expressions, the equivalence of KAT expressions may be decided using a deductive system and a set of axioms [17] or by minimization of deterministic automata [19]. In Chapter 3 we present a different procedure based on partial derivatives, which is an extension of the procedure for regular expressions in Chapter 2. Kozen [21] extended the notion of Brzozowski derivatives to KAT to prove the existence of a coinductive equivalence procedure. Our approach follows closely that work, but we explicitly define the notion of partial derivatives for KAT, and we effectively provide a (inductive) decision procedure.

In Chapter 4 we extend the procedure from Chapter 3 to prove the correctness of programs and to obtain other equivalence proofs in the presence of assumptions. We present some examples, including examples of proof of correctness of programs and an example of proof of safety of a program.

Some of the results and demonstrations in this dissertation are present in (Almeida, Broda and Moreira, 2012), cf. [4].

Chapter 2

Equivalence of Regular Expressions

2.1 Introduction

The problem of deciding if two regular expressions are equivalent is usually solved using automata. However, some experimental results indicate that a better average-case execution time is achieved if one compares regular expressions directly [3]. In this chapter we follow Almeida, Moreira and Reis, who developed a functional approach to the Antimirov and Mosses rewrite system [6].

We begin with the definition of a Kleene Algebra (KA) and of the language of regular expressions in Section 2.2. In Section 2.3 we describe the decision procedure to test the equivalence of two regular expressions, addressing some essential concepts such as Brzozowski's derivatives and partial derivatives. We close this chapter with the pseudo-code for all relevant implementations (Section 2.4).

2.2 Kleene Algebra and Regular Expressions

A Kleene algebra is an algebraic structure

$$\mathcal{K} = (K, +, \cdot, ^*, 0, 1)$$

satisfying the axioms (2.1-2.15) below.

$$r_1 + (r_2 + r_3) = (r_1 + r_2) + r_3 \tag{2.1}$$
$$r_1 + r_2 = r_2 + r_1 \tag{2.2}$$
$$r + 0 = r \tag{2.3}$$
$$r + r = r \tag{2.4}$$
$$r_1(r_2 r_3) = (r_1 r_2) r_3 \tag{2.5}$$
$$1r = r \tag{2.6}$$
$$r1 = r \tag{2.7}$$
$$r_1(r_2 + r_3) = r_1 r_2 + r_1 r_3 \tag{2.8}$$
$$(r_1 + r_2) r_3 = r_1 r_3 + r_2 r_3 \tag{2.9}$$
$$0r = 0 \tag{2.10}$$
$$r0 = 0 \tag{2.11}$$
$$1 + r r^* = r^* \tag{2.12}$$
$$1 + r^* r = r^* \tag{2.13}$$
$$r_1 + r_2 r_3 \leq r_3 \rightarrow {r_2}^* r_1 \leq r_3 \tag{2.14}$$
$$r_1 + r_2 r_3 \leq r_2 \rightarrow r_1 {r_3}^* \leq r_2 \tag{2.15}$$

In the above, $\leq$ refers to the natural partial order:

$$r_1 \leq r_2 \quad \text{iff} \quad r_1 + r_2 = r_2 \ .$$

The axioms (2.1-2.11) say that the structure is an *idempotent semiring* under $+$, $\cdot$, 0 and 1 and the remaining axioms (2.12-2.15) say that * behaves like the Kleene star operator of formal language theory.

Let $\Sigma = \{p_1, \ldots, p_k\}$, with $k \geq 1$, be an *alphabet*. A *word* w over Σ is any finite sequence of letters. The *empty word* is denoted by 1. Let Σ^* be the set of all words over Σ. A *language* over Σ is a subset of Σ^*. The *left quotient* of a language $L \subseteq \Sigma^*$ by a word $w \in \Sigma^*$ is the language $w^{-1}L = \{x \in \Sigma^* \mid wx \in L\}$. The set of *regular expressions* over Σ, RE, is given by the definition below.

Definition 2.1. *The abstract syntax of a regular expression over an alphabet Σ is given by the following grammar:*

$$r_1, r_2 \in \mathsf{RE} \ := \ 0 \mid 1 \mid p \in \Sigma \mid (r_1 + r_2) \mid (r_1 \cdot r_2) \mid ({r_1}^*)$$

where the operator $\cdot$ (concatenation) and the unnecessary parentheses are often omitted.

Definition 2.2. *The language defined by a regular expression r is a set of words $L \subseteq \Sigma^*$ inductively defined as follows:*

$$\begin{aligned} L(0) &= \emptyset \\ L(1) &= \{1\} \\ L(p) &= \{p\} \\ L(r_1 + r_2) &= L(r_1) \cup L(r_2) \\ L(r_1 \cdot r_2) &= L(r_1) \cdot L(r_2) \\ L(r^*) &= \bigcup_{n \in \mathbb{N}} L(r)^n \end{aligned}$$

where $L_1 \cdot L_2 = \{\, u \cdot v \mid u \in L_1 \wedge v \in L_2 \,\}$, $L^0 = \{1\}$ *and* $L^{n+1} = L \cdot L^n$.

Based on this notion the equivalence of regular expressions is defined.

Definition 2.3. *Two regular expressions r_1,r_2 are* equivalent*, and we write $r_1 = r_2$, if and only if $L(r_1) = L(r_2)$.*

Let *ACI* be the set of axioms that includes (2.1), (2.2) and (2.4). We say that *ACIA* is the set that contains the *ACI* axioms plus (2.5). In this chapter, the expressions of RE are always considered modulo the *ACIA* axioms and the axioms (2.6), (2.7), (2.10) and (2.11).

2.3 Deciding Equivalence in KA

The equivalence of regular expressions may be proven axiomatically, since Kozen has shown that the axioms set (2.1)–(2.15), provided with an usual first-order deduction system, constitutes a complete proof system for equivalence between regular expressions [16]. However, without an automatic proof system this is not very handy.

This section describes the decision algorithm presented by Almeida et. al [3], which addresses the equivalence via computation of the partial derivatives of the expressions. The algorithm is based on the rewrite system of Antimirov and Mosses [6]. Before describing the partial derivatives method, let us recall the classical Brzozowski's derivatives.

2.3.1 Brzozowski's Derivatives

The definition of the Brzozowski's derivatives depends on the notion of the *empty word property*. We say that a regular expression r has the empty word property if the language it defines contains the empty word, i. e., if $1 \in L(r)$. To capture this notion the following function is defined:

$$\varepsilon(r) \;=\; \begin{cases} 1\,, & \text{if } r \text{ has the empty word property} \\ 0\,, & \text{otherwise.} \end{cases}$$

One may define ε inductively as follows:

$$\begin{aligned} \varepsilon :\, \mathsf{RE} &\to \{0,1\} \\ \varepsilon(1) &= 1 \\ \varepsilon(0) &= 0 \\ \varepsilon(p) &= 0 \\ \varepsilon(r_1 + r_2) &= \varepsilon(r_1) + \varepsilon(r_2) \\ \varepsilon(r_1 \cdot r_2) &= \varepsilon(r_1)\varepsilon(r_2) \\ \varepsilon(r^*) &= 1 \end{aligned}$$

The following function computes Brzozowski's derivative of a regular expression r with respect to a symbol p.

$$\begin{aligned} D_p :\, \mathsf{RE} &\to \mathsf{RE} \\ D_p(0) &= 0 \\ D_p(1) &= 0 \\ D_p(p) &= 1 \\ D_{p_1}(p_2) &= 0 \qquad \text{for } p_1 \neq p_2 \\ D_p(r_1 + r_2) &= D_p(r_1) + D_p(r_2) \\ D_p(r_1 r_2) &= \begin{cases} D_p(r_1)r_2 + D_p(r_2) & \text{if } \varepsilon(r_1) = 1 \\ D_p(r_1)r_2 & \text{otherwise} \end{cases} \\ D_p(r^*) &= D_p(r)r^* \end{aligned}$$

These rules are extended to words $w \in \Sigma^*$ as follows:

$$\begin{aligned} \hat{D}_w :\, \mathsf{RE} &\to \mathsf{RE} \\ \hat{D}_1(r) &= r \\ \hat{D}_{wp}(r) &= D_p(\hat{D}_w(r)) \end{aligned}$$

Brzozowski proved that, considering regular expressions modulo the *ACI* axioms and the axioms (2.6), (2.7), (2.10) and (2.11), for every regular expression r, the set of derivatives of r with respect to any word is finite. This is necessary to ensure the termination of the algorithm in Section 2.3.3.

2.3.2 Partial Derivatives

Introduced by Antimirov, a *partial derivative* of a regular expression is regarded as a non-deterministic variation of the Brzozowski's derivative [5]. Let r be a regular expression and p a symbol of Σ. The *set of partial derivatives* of r with respect to p is defined as follows:

$$
\begin{aligned}
\partial_p : \mathsf{RE} &\to \mathcal{P}(\mathsf{RE}) \\
\partial_p(0) &= \emptyset \\
\partial_p(1) &= \emptyset \\
\partial_p(p) &= \{1\} \\
\partial_{p_1}(p_2) &= \emptyset \qquad \text{for } p_1 \neq p_2 \\
\partial_p(r_1 + r_2) &= \partial_p(r_1) \cup \partial_p(r_2) \\
\partial_p(r_1 r_2) &= \begin{cases} \partial_p(r_1) \cdot r_2 \cup \partial_p(r_2) & \text{if } \varepsilon(r_1) = 1 \\ \partial_p(r_1) \cdot r_2 & \text{otherwise.} \end{cases} \\
\partial_p(r^*) &= \partial_p(r) \cdot r^*
\end{aligned}
$$

where for $\Gamma \subseteq \mathsf{RE}$ and $r \in \mathsf{RE}$, $\Gamma \cdot r = \{\, r'r \mid r' \in \Gamma \,\}$ if $r \neq 0$ and $r \neq 1$, and $\Gamma \cdot 0 = \emptyset$ and $\Gamma \cdot 1 = \Gamma$, otherwise.

The notion of partial derivative of a regular expression r can be extended to words $u \in \Sigma^*$ as follows:

$$
\hat{\partial}_1(r) = \{r\} \qquad\qquad \hat{\partial}_{up}(r) = \bigcup_{x \in \hat{\partial}_u(r)} \partial_p(x)
$$

Let $|r|_\Sigma$ be the number of occurrences of symbols of Σ in the regular expression r. It is known that for any expression r, we have $|PD(r)| \leq |r|_\Sigma$, where $PD(r)$ is the set of all syntactically different partial derivatives of r [5].

2.3.3 A Decision Procedure for Regular Expressions Equivalence

We now describe the decision algorithm introduced by Almeida et. al [3], together with all necessary functions. However, the algorithm presented here has a slight modification: every sum is represented by a set containing all subexpressions involved in the operation, rather than by a single regular expression. Consequently, several functions had to be rewritten to accommodate this change (see below). The pseudocode for all relevant implementations is available in Section 2.4.

As mentioned before, this algorithm tests the equivalence of expressions via computation of the sets of partial derivatives of the given expressions. This process relies on the *linearization* of regular expressions. The term linearization is borrowed from Antimirov and Mosses [3]. They defined a linear expression to be a regular expression r of the form $r = p_1 r_1 + p_2 r_2 + \ldots + p_n r_n$, where $p_i \in \Sigma$ and $r_i \in \mathsf{RE}$. The symbols p_i are said to be the *heads* of r. The set of all heads of r is denoted by $\mathsf{hd}(r)$.

Linear expressions are represented by sets. Let $\Sigma \times \mathsf{RE}$ be the set of pairs over an alphabet Σ. A linear expression of the form $p_1 r_1 + \ldots + p_n r_n$ can be represented by a finite set of pairs $l \in \mathcal{P}(\Sigma \times \mathsf{RE})$, named *linear form*, such that $l = \{(p_1, r_1), \ldots, (p_n, r_n)\}$. The concatenation of l with an expression r' is defined by $l \cdot r' = \{(p_1, r_1 \cdot r'), \ldots, (p_n, r_n \cdot r')\}$. Linearization is thus the process of converting a regular expression to a linear form and is given by the following function:

$$
\begin{aligned}
\mathsf{f} : \mathsf{RE} &\to \mathcal{P}(\Sigma \times \mathsf{RE}) \\
\mathsf{f}(0) &= \emptyset \\
\mathsf{f}(1) &= \emptyset \\
\mathsf{f}(p) &= \{(p, 1)\} \\
\mathsf{f}(r_1 + r_2) &= \mathsf{f}(r_1) \cup \mathsf{f}(r_2) \\
\mathsf{f}(p\, r) &= \{(p, r)\} \\
\mathsf{f}({r_1}^* r_2) &= \mathsf{f}(r_1) \cdot {r_1}^* r_2 \cup \mathsf{f}(r_2) \\
\mathsf{f}((r_1 + r_2)\, r_3) &= \mathsf{f}(r_1\, r_3) \cup \mathsf{f}(r_2\, r_3) \\
\mathsf{f}(r^*) &= \mathsf{f}(r) \cdot r^*
\end{aligned}
$$

The function f divides the case of concatenation in three different cases in order to avoid the ε test. Since regular expressions are considered modulo *ACIA* and the axioms (2.6), (2.7), (2.10) and (2.11), every concatenation is necessarily of one of these cases.

The linearization of regular expressions is important to guarantee efficiency, since a single call to $\mathsf{f}(r)$ computes all partial derivatives of r with respect to all symbols in $\mathsf{hd}(r)$.

It is easy to see that the set of partial derivatives of r by p is the set of all r' such that $(p, r') \in \mathsf{f}(r)$. This notion is formally given by the function

$$\mathsf{der}_p : \mathsf{RE} \to \mathcal{P}(\mathsf{RE})$$
$$\mathsf{der}_p(r) = \{\, r' \mid (p, r') \in \mathsf{f}(r) \,\}.$$

To define the decision procedure we need to consider the functions der_p, hd and ε from Section 2.2 applied to sets of regular expressions.

$$\varepsilon : \mathcal{P}(\mathsf{RE}) \to \mathsf{RE}$$
$$\varepsilon(R) = 1 \quad \text{iff} \quad \exists_{r \in R} : \varepsilon(r) = 1$$

The functions der_p and hd are extended as follows:

$$\mathsf{der}_p : \mathcal{P}(\mathsf{RE}) \to \mathcal{P}(\mathsf{RE})$$
$$\mathsf{der}_p(R) = \bigcup_{r \in R} \mathsf{der}_p(r).$$

and

$$\mathsf{hd} : \mathcal{P}(\mathsf{RE}) \to \mathcal{P}(\Sigma)$$
$$\mathsf{hd}(R) = \bigcup_{r \in R} \mathsf{hd}(r).$$

Then, the function $\mathsf{derivatives}$ is defined so that given two (sets of) regular expressions R_1 and R_2 it computes all pairs of sets of partial derivatives with respect to each symbol p of the alphabet of expressions in R_1 and R_2, respectively. For efficiency reasons it is enough to consider only the symbols at the heads of the expressions.

$$\mathsf{derivatives} : \mathcal{P}(\mathsf{RE})^2 \to \mathcal{P}(\mathcal{P}(\mathsf{RE})^2)$$
$$\mathsf{derivatives}(R_1, R_2) = \{\, (\mathsf{der}_p(R_1), \mathsf{der}_p(R_2)) \mid p \in \mathsf{hd}(R_1 \cup R_2) \,\}$$

The function equiv returns True if and only if, given two (sets of) expressions r_1 and r_2, we have $r_1 = r_2$. For two sets R_1 and R_2 the function returns True if $\varepsilon(R_1) = \varepsilon(R_2)$ and if, for every p, the partial derivative of R_1 w.r.t. p is equivalent to the partial derivative of R_2 w.r.t. p.

$$\mathsf{equiv} : \mathcal{P}(\mathcal{P}(\mathsf{RE})^2) \times \mathcal{P}(\mathcal{P}(\mathsf{RE})^2) \to \{\mathsf{True}, \mathsf{False}\}$$
$$\mathsf{equiv}(\emptyset, H) = \mathsf{True}$$
$$\mathsf{equiv}(\{(R_1, R_2)\} \cup S, H) = \begin{cases} \mathsf{False} & \text{if } \varepsilon(R_1) \neq \varepsilon(R_2) \\ \mathsf{equiv}(S \cup S', H') & \text{otherwise,} \end{cases}$$

where

$$S' = \{d \mid d \in \mathsf{derivatives}(R_1, R_2) \text{ and } d \notin H'\} \quad \text{and} \quad H' = \{(R_1, R_2)\} \cup H.$$

The function equiv accepts two sets S and H as arguments. At each step, S contains the pairs of (sets of) expressions that still need to be checked for equivalence, while H contains the pairs of (sets of) expressions that have already been tested. The use of the set H is important to ensure that the derivatives of the same pair of (sets of) expressions are not computed more than once, and thus prevent a possibly infinite loop.

To compare two expressions r_1 and r_2, the initial call must be $\mathsf{equiv}(\{(\{r_1\}, \{r_2\})\}, \emptyset)$. At each step the function takes a pair (R_1, R_2) and verifies if $\varepsilon(R_1) = \varepsilon(R_2)$. If the test fails, then $r_1 \neq r_2$ and the function halts, returning False. If the test succeeds, then the function adds (R_1, R_2) to H and then replaces in S the pair (R_1, R_2) by the pairs of its corresponding derivatives, provided that these are not in H already. The return value of equiv will be the result of recursively calling equiv with the new sets as arguments. If the function ever receives $\emptyset$ as S, then the initial call ensures that $r_1 = r_2$, since all derivatives have been successfully tested, and the function returns True.

The proof of correctness and termination of the algorithm can be found in [3].

2.4 Pseudo-code of Relevant Implementations

In this section we present the pseudo-code for all relevant functions presented in this chapter and for the representation of regular expressions. The language chosen for implementation was *OCaml* [25].

2.4.1 Representation of Regular Expressions

The data structure we used to represent regular expressions is defined as follows:

```
type re = Zero
        | One
        | Symb of int
        | Or of (re set * bool)
        | Conc of re list
        | Star of re
```

Every symbol p_i of Σ is represented by its respective index i.

We represent sums by structures *(set, flag)*, where *set* contains the expressions involved in the operation and *flag* indicates if any of the expressions in *set* has the empty word property. The value of this *flag* is defined during the parsing of the expressions. We regard this as an optimization to facilitate the test $\varepsilon(r_1) = \varepsilon(r_2)$ in the function equiv (see Section 2.3.3). Concatenations are kept in lists of expressions.

2.4.2 Implementation of the Decision Procedure

In this subsection we present the pseudo-code for all relevant functions involved in the decision procedure. There is a slight difference between the implementation and the formal definition from Section 2.3.3, concerning how the sets of partial derivatives are computed. Here we use an additional function, called g, that first invokes the function f to calculate the linear form of an expression r but then rearranges the computed set so that all expressions with common heads are kept in the same pair. This corresponds to the determinization process of regular expressions [3]. Thus, when applied to an expression r, instead of building a set containing all expressions r' such that $r' \in \mathsf{f}(r)$, the function der_p simply returns the set s such that $(p, s) \in \mathsf{g}(r)$, or $Zero$ if that set does not exist.

We begin with the pseudo-code for the function f, which converts an expression to a linear form.

$$
\begin{array}{l}
\text{let } f\ exp\ =\ \text{match } exp \text{ with} \\
\quad Zero \ \longrightarrow\ \emptyset \\
\quad \mid One \ \longrightarrow\ \emptyset \\
\quad \mid Symb\ p \ \longrightarrow\ \{(p, One)\} \\
\quad \mid Or\ (set,\ flag) \ \longrightarrow\ \cup\ (map\ f\ (flattenConcats\ (fromSetToList\ set))) \\
\quad \mid Star\ exp' \ \longrightarrow\ map\ (concat\ exp)\ (\ f\ (flattenConcats\ exp')) \\
\quad \mid Conc\ (\ Symb\ p\ )\ ::\ rs \ \longrightarrow\ \{(p,\ buildConc\ rs)\} \\
\quad \mid Conc\ (Or\ (set,\ flag))\ ::\ rs \ \longrightarrow\ \cup\ (map\ f\ (flattenConcats\ concatenations)) \\
\qquad \text{where} \\
\qquad\quad concatenations \ \leftarrow\ map\ (concat'\ (buildConc\ rs))\ (fromSetToList\ set) \\
\quad \mid Conc\ (\ Star\ exp'\ )\ ::\ rs \ \longrightarrow\ (map\ (concat\ exp)\ (\ f\ (flattenConcats\ exp'))) \cup \\
(\ f\ (buildConc\ rs))
\end{array}
$$

The function $flattenConcats$ inspects expressions, recursively looking for concatenation lists inside concatenation lists and flattening them, so that all expressions involved

in the concatenation are at the same level.

The function *concat* accepts an expression exp and a pair (p, exp') as arguments and returns $(p, exp \cdot exp')$.

The function *buildConc* constructs a concatenation of expressions from a list rs given as argument, provided there are at least two expressions in the list. If rs contains only one expression, then *buildConc* simply returns that element as the correct regular expression instead of constructing a concatenation.

The function *concat′* receives two expressions as arguments and concatenates the first one to the end of the second one.

The following is the pseudo-code for the function g.

$$\begin{array}{l} \text{let } g\ exp = fromListToSet\ [(p, buildOr\ (g'\ p\ l_r)) \mid (p, \_) \leftarrow l_r] \\ \quad \text{where} \\ \quad\quad l_r \leftarrow fromSetToList\ (f\ r) \end{array}$$

and

$$\text{let } g'\ p\ ls = [r \mid (p', r) \leftarrow ls\ \ \&\&\ \ p = p']$$

As the name suggests, *buildOr* is a function that receives a list of expressions and builds an *Or* with *(set,flag)*, where *set* contains the elements in the list and *flag* indicates whether there is at least one element in set that has the empty word property. But if the list contains only one expression, then *buildOr* simply returns that expression and no *Or* construct is created.

Now we present the pseudo-code for the function derivatives, which uses a comprehension list to compute the sets of all partial derivatives of the expressions.

$$\begin{array}{l} \text{let } derivatives\ set_1\ set_2 = [(p, der_p\ ls_1, der_p\ ls_2) \mid (p, \_) \leftarrow (append\ ls_1\ ls_2)] \\ \quad \text{where} \\ \quad\quad ls_1 \leftarrow fromSetToList\ set_1 \\ \quad\quad ls_2 \leftarrow fromSetToList\ set_2 \end{array}$$

and

$$\begin{array}{lll} \text{let } der_p\ l_r = \text{match } l_r \text{ with} & & \\ \quad (p', e) :: ls & \longrightarrow & \textbf{if } p' = p \textbf{ then } e \textbf{ else } der_p\ ls \\ \quad \mid [\,] & \longrightarrow & Zero \end{array}$$

Finally we present the pseudo-code for the function equiv. The function uses an aux-

iliary function $hasEmptyWordProp$ that, as the name suggests, verifies if a given expression has the empty word property.

let $equiv\ S\ H\ =$
if $S{=}\emptyset$ **then**
  (True, H)
else if $(hasEmptyWordProp\ r_1)$!= $(hasEmptyWordProp\ r_2)$ **then**
  (False, H)
  where
    $(r_1, r_2) \leftarrow S$.firstElement
else
  $equiv\ (S \setminus (r_1, r_2) \cup S')\ H'$
  where
    $H' \leftarrow H \cup (r_1, r_2)$
    $S' \leftarrow fromListToSet\ [(der_1, der_2) \mid (\_, der_1, der_2) \leftarrow derivatives\ (g\ r_1)\ (g\ r_2)$
  && $(der_1, der_2) \notin H']$
end if

Chapter 3

Equivalence of KAT Expressions

3.1 Introduction

In this section we present a decision algorithm to test equivalence of expressions in KAT. This decision procedure is an extension of the algorithm described in Section 2.3.3 for deciding equivalence of regular expressions, that does not use the axiomatic system. Equivalence of expressions is decided through an iterated process of testing the equivalence of their partial derivatives.

We begin this chapter with the definition of a Kleene algebra with tests (KAT) in Section 3.2. In Section 3.2.1 we define the syntax and the semantics of a KAT expression and define the set of guarded strings denoted by a KAT expression. In Section 3.3.1 we present a definition of the derivative of a set of guarded strings with respect to a symbol. In Section 3.3.2 we give the definitions for the derivatives of an expression with respect to a symbol and with respect to a sequence of symbols. In Section 3.3.3 we introduce a decision algorithm to test the equivalence of two expressions. The pseudo-code for all relevant implementations is available in Section 3.4. This chapter closes with a presentation of results of several tests performed with the algorithm.

3.2 Kleene Algebra with Tests

A Kleene algebra with tests (KAT) is a Kleene algebra with an embedded Boolean subalgebra

$$\mathcal{K} = (K, B, +, \cdot, ^*, 0, 1, ^-)$$

where $^-$ denotes negation and is an unary operator defined only on B, such that

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

Thus, a KAT is an algebraic structure that satisfies the KA axioms (2.1)–(2.15) (see Chapter 2) and the axioms for a Boolean algebra.

3.2.1 KAT Expressions and Guarded Strings

Let $\Sigma = \{p_1, \ldots, p_k\}$ be a non-empty set of (primitive) *action* symbols and $T = \{t_1, \ldots, t_l\}$ be a non-empty set of (primitive) *test* symbols. The set of boolean expressions over T is denoted by Bexp and the set of KAT expressions by Exp, with elements $b_1, b_2, \ldots$ and $e_1, e_2, \ldots$, respectively. The abstract syntax of KAT expressions over an alphabet $\Sigma \cup T$ is given by the following grammar,

$$\begin{aligned} b \in \mathsf{Bexp} &:= 0 \mid 1 \mid t \in T \mid (\bar{b}) \mid (b_1 + b_2) \mid (b_1 \cdot b_2) \\ e \in \mathsf{Exp} &:= p \in \Sigma \mid b \in \mathsf{Bexp} \mid (e_1 + e_2) \mid (e_1 \cdot e_2) \mid ({e_1}^*). \end{aligned}$$

As usual, we often omit the operator $\cdot$ in concatenations and in conjunctions, as well as the unnecessary parentheses. The standard language-theoretic models of KAT are regular sets of *guarded strings* over alphabets Σ and T [19]. Let $\overline{T} = \{\bar{t} \mid t \in T\}$ and let At be the set of *atoms*, i. e., of all truth assignments to T,

$$\mathsf{At} = \{b_1 \ldots b_l \mid b_i \text{ is either } t_i \text{ or } \overline{t_i} \text{ for } 1 \leq i \leq l \text{ and } t_i \in T\}.$$

We reserve the symbols $\alpha_1, \alpha_2, \ldots$ for atoms. For an atom α and a BA expression b, we write $\alpha \leq b$ if $\alpha \to b$ is a propositional tautology. For every atom α and every test t, we either have $\alpha \leq t$ or $\alpha \leq \bar{t}$.

Then the set of *guarded strings* over Σ and T is

$$\mathsf{GS} = (\mathsf{At} \cdot \Sigma)^* \cdot \mathsf{At}.$$

Guarded strings will be denoted by $x, y, \ldots$. For $x = \alpha_1 p_1 \alpha_2 p_2 \cdots p_{n-1} \alpha_n \in \mathsf{GS}$, where $n \geq 1$, $\alpha_i \in \mathsf{At}$ and $p_i \in \Sigma$, we define $\mathsf{first}(x) = \alpha_1$ and $\mathsf{last}(x) = \alpha_n$. If $\mathsf{last}(x) = \mathsf{first}(y)$, then the *fusion product* xy is defined by concatenating x and y,

omitting the extra occurrence of the common atom. If $\mathsf{last}(x) \neq \mathsf{first}(y)$, then xy does not exist. For sets $X, Y \subseteq \mathsf{GS}$ of guarded strings, the set $X \diamond Y$ defines the set of all xy such that $x \in X$ and $y \in Y$. We have that $X^0 = \mathsf{At}$ and $X^{n+1} = X \diamond X^n$, for $n \geq 0$.

Every KAT expression $e \in \mathsf{Exp}$ denotes a set of *guarded strings*, $\mathsf{GS}(e) \subseteq \mathsf{GS}$. Given a KAT expression e we define $\mathsf{GS}(e)$ inductively as follows,

$$\mathsf{GS}(p) = \{\, \alpha_1 p \alpha_2 \mid \alpha_1, \alpha_2 \in \mathsf{At} \,\} \qquad p \in \Sigma \tag{3.1}$$

$$\mathsf{GS}(b) = \{\, \alpha \in \mathsf{At} \mid \alpha \leq b \,\} \qquad b \in \mathsf{Bexp} \tag{3.2}$$

$$\mathsf{GS}(e_1 + e_2) = \mathsf{GS}(e_1) \cup \mathsf{GS}(e_2) \tag{3.3}$$

$$\mathsf{GS}(e_1 e_2) = \mathsf{GS}(e_1) \diamond \mathsf{GS}(e_2) \tag{3.4}$$

$$\mathsf{GS}(e^*) = \bigcup_{n \geq 0} \mathsf{GS}(e)^n. \tag{3.5}$$

Based on this notion the equivalence of two KAT expressions is defined.

Definition 3.1. *Two KAT expressions e_1,e_2 are* equivalent*, and we write $e_1 = e_2$, if and only if $\mathsf{GS}(e_1) = \mathsf{GS}(e_2)$.*

Kozen has shown that one has $e_1 = e_2$ modulo the KAT axioms if and only if they are equivalent [22]. Two sets of KAT expressions $E, F \subseteq \mathsf{Exp}$ are equivalent if and only if $\mathsf{GS}(E) = \mathsf{GS}(F)$, where

$$\mathsf{GS}(E) = \bigcup_{e \in E} \mathsf{GS}(e). \tag{3.6}$$

3.3 Deciding Equivalence in KAT

3.3.1 Derivatives

Given a set of guarded strings R, its derivative with respect to $\alpha p \in \mathsf{At} \cdot \Sigma$, denoted by $\mathsf{D}_{\alpha p}(R)$, is defined as being the *left quotient* of R by αp. As such, one considers the following *derivative* functions,

$$\mathsf{D} : \mathsf{At} \cdot \Sigma \to \mathcal{P}(\mathsf{GS}) \to \mathcal{P}(\mathsf{GS}) \qquad \mathsf{E} : \mathsf{At} \to \mathcal{P}(\mathsf{GS}) \to \{0, 1\}$$

consisting of components,

$$\mathsf{D}_{\alpha p} : \mathcal{P}(\mathsf{GS}) \to \mathcal{P}(\mathsf{GS}) \qquad \mathsf{E}_\alpha : \mathcal{P}(\mathsf{GS}) \to \{0, 1\}$$

defined as follows. For $\alpha \in \mathsf{At}$, $p \in \Sigma$ and $R \subseteq \mathsf{GS}$,

$$\mathsf{D}_{\alpha p}(R) \;=\; \{\, y \in \mathsf{GS} \mid \alpha p y \in R \,\} \qquad \text{and} \qquad \mathsf{E}_\alpha(R) \;=\; \begin{cases} 1 & \text{if } \alpha \in R \\ 0 & \text{otherwise.} \end{cases}$$

3.3.2 Partial Derivatives

Given $\alpha \in \mathsf{At}$, $p \in \Sigma$ and $e \in \mathsf{Exp}$, the set $\Delta_{\alpha p}(e)$ of partial derivatives of e with respect to αp is inductively defined as follows,

$$\begin{aligned}
\Delta_{\alpha p} &: \mathsf{Exp} \to \mathcal{P}(\mathsf{Exp}) \\
\Delta_{\alpha p}(p') &= \begin{cases} \{1\} & \text{if } p = p' \\ \emptyset & \text{otherwise} \end{cases} && (3.7)\\
\Delta_{\alpha p}(b) &= \emptyset && (3.8)\\
\Delta_{\alpha p}(e_1 + e_2) &= \Delta_{\alpha p}(e_1) \cup \Delta_{\alpha p}(e_2) && (3.9)\\
\Delta_{\alpha p}(e_1 e_2) &= \begin{cases} \Delta_{\alpha p}(e_1) \cdot e_2 & \text{if } \mathsf{E}_\alpha(e_1) = 0 \\ \Delta_{\alpha p}(e_1) \cdot e_2 \cup \Delta_{\alpha p}(e_2) & \text{if } \mathsf{E}_\alpha(e_1) = 1 \end{cases} && (3.10)\\
\Delta_{\alpha p}(e^*) &= \Delta_{\alpha p}(e) \cdot e^*, && (3.11)
\end{aligned}$$

where for $\Gamma \subseteq \mathsf{Exp}$ and $e \in \mathsf{Exp}$, $\Gamma \cdot e = \{\, e'e \mid e' \in \Gamma \,\}$ if $e \neq 0$ and $e \neq 1$, and $\Gamma \cdot 0 = \emptyset$ and $\Gamma \cdot 1 = \Gamma$, otherwise. We note that $\Delta_{\alpha p}(e)$ corresponds to an equivalence class of $D_{\alpha p}(e)$ (the syntactic Brzozowski derivative, defined in [21]) modulo axioms (2.1)–(2.4), (2.6), (2.7) and (2.9)–(2.11). Kozen calls such a structure a *right presemiring*.

The following syntactic definition of $\mathsf{E}_\alpha : \mathsf{At} \to \mathsf{Exp} \to \{0,1\}$ simply evaluates an expression with respect to the truth assignment α [21].

$$\begin{aligned}
\mathsf{E}_\alpha(p) &= 0 & \qquad \mathsf{E}_\alpha(e_1 + e_2) &= \mathsf{E}_\alpha(e_1) + \mathsf{E}_\alpha(e_2) \\
\mathsf{E}_\alpha(b) &= \begin{cases} 1 & \text{if } \alpha \leq b \\ 0 & \text{otherwise} \end{cases} & \qquad \mathsf{E}_\alpha(e_1 e_2) &= \mathsf{E}_\alpha(e_1)\mathsf{E}_\alpha(e_2) \\
& & \qquad \mathsf{E}_\alpha(e^*) &= 1.
\end{aligned}$$

One can show that,

$$\mathsf{E}_\alpha(e) = \begin{cases} 1 & \text{if } \alpha \leq e \\ 0 & \text{if } \alpha \not\leq e \end{cases} = \begin{cases} 1 & \text{if } \alpha \in \mathsf{GS}(e) \\ 0 & \text{if } \alpha \notin \mathsf{GS}(e). \end{cases}$$

The next proposition shows that for all KAT expressions e the set of guarded strings corresponding to the set of partial derivatives of e w.r.t. $\alpha p \in \mathsf{At} \cdot \Sigma$ is the derivative of $\mathsf{GS}(e)$ by αp.

Proposition 3.1. *For all KAT expressions e, all atoms α and all symbols p,*

$$\mathsf{D}_{\alpha p}(\mathsf{GS}(e)) = \mathsf{GS}(\Delta_{\alpha p}(e)).$$

Proof. The proof is obtained by induction on the structure of e. The base cases, $e \in \mathsf{Bexp}$ or $e \in \Sigma$ are trivial.

- If $e = e_1 + e_2$ then

$$\begin{aligned}
\mathsf{D}_{\alpha p}(\mathsf{GS}(e)) &= \mathsf{D}_{\alpha p}(\mathsf{GS}(e_1)) \cup \mathsf{D}_{\alpha p}(\mathsf{GS}(e_2)). \\
&\quad \text{applying the induction hypotheses} \\
&= (\mathsf{GS}(\Delta_{\alpha p}(e_1))) \cup (\mathsf{GS}(\Delta_{\alpha p}(e_2))) \\
&= \mathsf{GS}(\Delta_{\alpha p}(e_1) \cup \Delta_{\alpha p}(e_2)) \\
&\quad \text{applying (3.9)} \\
&= \mathsf{GS}(\Delta_{\alpha p}(e_1 + e_2)) \\
&= \mathsf{GS}(\Delta_{\alpha p}(e)).
\end{aligned}$$

- If $e = e_1 e_2$ then

$$\begin{aligned}
\mathsf{D}_{\alpha p}(\mathsf{GS}(e)) &= \mathsf{D}_{\alpha p}(\mathsf{GS}(e_1) \diamond \mathsf{GS}(e_2)) \\
&= \begin{cases} \mathsf{D}_{\alpha p}(\mathsf{GS}(e_1)) \diamond \mathsf{GS}(e_2) & \text{if } \alpha \notin \mathsf{GS}(e_1) \\ \mathsf{D}_{\alpha p}(\mathsf{GS}(e_1)) \diamond \mathsf{GS}(e_2) \cup \mathsf{D}_{\alpha p}(\mathsf{GS}(e_2)) & \text{if } \alpha \in \mathsf{GS}(e_1) \end{cases} \\
&\quad \text{applying the induction hypothesis} \\
&= \begin{cases} (\cup_{e' \in \Delta_{\alpha p}(e_1)} \mathsf{GS}(e')) \diamond \mathsf{GS}(e_2) & \text{if } \mathsf{E}_\alpha(e_1) = 0 \\ (\cup_{e' \in \Delta_{\alpha p}(e_1)} \mathsf{GS}(e')) \diamond \mathsf{GS}(e_2) \cup \mathsf{GS}(\Delta_{\alpha p}(e_2)) & \text{if } \mathsf{E}_\alpha(e_1) = 1 \end{cases} \\
&\quad \text{applying (3.4)} \\
&= \begin{cases} \cup_{e' \in \Delta_{\alpha p}(e_1)} \mathsf{GS}(e' e_2) & \text{if } \mathsf{E}_\alpha(e_1) = 0 \\ (\cup_{e' \in \Delta_{\alpha p}(e_1)} \mathsf{GS}(e' e_2)) \cup \mathsf{GS}(\Delta_{\alpha p}(e_2)) & \text{if } \mathsf{E}_\alpha(e_1) = 1 \end{cases} \\
&\quad \text{applying (3.6)} \\
&= \begin{cases} \mathsf{GS}(\Delta_{\alpha p}(e_1) \cdot e_2) & \text{if } \mathsf{E}_\alpha(e_1) = 0 \\ \mathsf{GS}(\Delta_{\alpha p}(e_1) \cdot e_2) \cup \mathsf{GS}(\Delta_{\alpha p}(e_2)) & \text{if } \mathsf{E}_\alpha(e_1) = 1 \end{cases} \\
&\quad \text{applying (3.10)} \\
&= \mathsf{GS}(\Delta_{\alpha p}(e_1 e_2)) \;=\; \mathsf{GS}(\Delta_{\alpha p}(e)).
\end{aligned}$$

- If $e = {e_1}^*$ then

$$\begin{aligned}
\mathsf{D}_{\alpha p}(\mathsf{GS}(e)) &= \mathsf{D}_{\alpha p}(\underset{n\geq 0}{\cup} \mathsf{GS}(e_1)^n) \\
&\quad \text{applying (3.4)} \\
&= \mathsf{D}_{\alpha p}(\underset{n\geq 0}{\cup} \mathsf{GS}(e_1^n)) \\
&= \underset{n\geq 0}{\cup} \mathsf{D}_{\alpha p}(\mathsf{GS}(e_1^n)) \\
&\quad \text{applying the induction hypothesis} \\
&= \underset{n\geq 0}{\cup} \mathsf{GS}(\Delta_{\alpha p}(e_1^n)) \\
&= \mathsf{GS}(\Delta_{\alpha p}(e_1)) \cup \mathsf{GS}(\Delta_{\alpha p}(e_1)\cdot e_1) \cup \mathsf{GS}(\Delta_{\alpha p}(e_1)\cdot e_1^2) \cup \ldots \\
&\quad \text{applying (3.4)} \\
&= \underset{n\geq 0}{\cup} \mathsf{GS}(\Delta_{\alpha p}(e_1)) \diamond \mathsf{GS}(e_1^n) \\
&\quad \text{applying (3.5)} \\
&= \mathsf{GS}(\Delta_{\alpha p}(e_1)) \diamond \mathsf{GS}({e_1}^*) \\
&\quad \text{applying (3.4)} \\
&= \mathsf{GS}(\Delta_{\alpha p}(e_1)\cdot {e_1}^*) \\
&\quad \text{applying (3.11)} \\
&= \mathsf{GS}(\Delta_{\alpha p}({e_1}^*)) \;=\; \mathsf{GS}(\Delta_{\alpha p}(e))
\end{aligned}$$

  where $e^n = e^{n-1}e$ and $e^0 = 1$.

□

The notion of partial derivative of an expression with respect to $\alpha p \in \mathsf{At}\cdot\Sigma$ can be extended to words $x \in (\mathsf{At}\cdot\Sigma)^*$, as follows,

$$\begin{aligned}
\hat{\Delta} : (\mathsf{At}\cdot\Sigma)^* \;\to\; \mathsf{Exp} &\to\; \mathcal{P}(\mathsf{Exp}) \\
\hat{\Delta}_1(e) &= \{e\} \\
\hat{\Delta}_{w\alpha p}(e) &= \Delta_{\alpha p}(\hat{\Delta}_w(e)).
\end{aligned}$$

Here, the notion of (partial) derivative has been extended to sets of KAT expressions $E \subseteq \mathsf{Exp}$, by defining, as expected, $\Delta_{\alpha p}(E) = \cup_{e\in E}\Delta_{\alpha p}(e)$, for $\alpha p \in \mathsf{At}\cdot\Sigma$. Analogously, we also consider $\hat{\Delta}_x(E)$ and $\hat{\Delta}_R(E)$, for $x \in (\mathsf{At}\cdot\Sigma)^*$ and $R \subseteq (\mathsf{At}\cdot\Sigma)^*$.

The fact that for any $e \in \mathsf{Exp}$ the set $\hat{\Delta}_{(\mathsf{At}\cdot\Sigma)^*}(e)$ is finite ensures the termination of the decision procedure presented in the next section.

3.3.3 A Decision Procedure for KAT Expressions Equivalence

In this section we describe an algorithm for testing the equivalence of two KAT expressions using partial derivatives. This algorithm is an extension of the algorithm presented in Section 2.3.3 for regular expressions. The pseudo-code for all relevant implementations can be found in Section 3.4.

Analogously to what we did in the previous chapter, we define the function f that given an expression e computes the set of pairs $(\alpha p, e')$, such that for each $\alpha p \in \mathsf{At} \cdot \Sigma$, the corresponding e' is a partial derivative of e with respect to αp.

$$\begin{aligned}
\mathsf{f} : \mathsf{Exp} &\rightarrow \mathcal{P}(\mathsf{At} \cdot \Sigma \times \mathsf{Exp}) \\
\mathsf{f}(p) &= \{\, (\alpha p, 1) \mid \alpha \in \mathsf{At} \,\} \\
\mathsf{f}(b) &= \emptyset \\
\mathsf{f}(e_1 + e_2) &= \mathsf{f}(e_1) \cup \mathsf{f}(e_2) \\
\mathsf{f}(e_1 e_2) &= \mathsf{f}(e_1) \cdot e_2 \cup \{\, (\alpha p, e) \in \mathsf{f}(e_2) \mid \mathsf{E}_\alpha(e_1) = 1 \,\} \\
\mathsf{f}(e^*) &= \mathsf{f}(e) \cdot e^*
\end{aligned}$$

where, as before, $\Gamma \cdot e = \{\, (\alpha p, e'e) \mid (\alpha p, e') \in \Gamma \,\}$ if $e \neq 0$ and $e \neq 1$, and $\Gamma \cdot 0 = \emptyset$ and $\Gamma \cdot 1 = \Gamma$, otherwise. Also, we denote by $\mathsf{hd}(\mathsf{f}(e)) = \{\alpha p \mid (\alpha p, e') \in \mathsf{f}(e)\}$ the set of *heads* (i. e. first components of each element) of $\mathsf{f}(e)$. The function $\mathsf{der}_{\alpha p}$, defined in (3.12), collects all the partial derivatives of an expression e w.r.t. αp, that were computed by function f.

$$\mathsf{der}_{\alpha p}(e) = \{e' \mid (\alpha p, e') \in \mathsf{f}(e)\} \tag{3.12}$$

Proposition 3.2. *For all $e, e' \in \mathsf{Exp}$, $\alpha \in \mathsf{At}$ and $p \in \Sigma$ one has, $\mathsf{der}_{\alpha p}(e) = \Delta_{\alpha p}(e)$.*

Proof. We proceed by structural induction. The proof splits into cases according to the structure of e. The cases when $e \in \Sigma$ or $e \in \mathsf{Bexp}$ follow trivially from the definitions of the derivative functions.

- If $e = e_1 + e_2$ then

$$\begin{aligned}
\mathsf{der}_{\alpha p}(e) &= \{\, e' \mid (\alpha p, e') \in \mathsf{f}(e_1) \cup \mathsf{f}(e_2) \,\} \\
&= \{\, e' \mid (\alpha p, e') \in \mathsf{f}(e_1) \,\} \cup \{\, e' \mid (\alpha p, e') \in \mathsf{f}(e_2) \,\} \\
&\quad \text{applying (3.12)} \\
&= \mathsf{der}_{\alpha p}(e_1) \cup \mathsf{der}_{\alpha p}(e_2) \\
&\quad \text{applying the induction hypotheses} \\
&= \Delta_{\alpha p}(e_1) \cup \Delta_{\alpha p}(e_2) \\
&\quad \text{applying (3.9)} \\
&= \Delta_{\alpha p}(e_1 + e_2) \;=\; \Delta_{\alpha p}(e).
\end{aligned}$$

- If $e = e_1 e_2$ then

$$\begin{aligned}
\mathsf{der}_{\alpha p}(e) &= \begin{cases} \{\, e' \mid (\alpha p, e') \in \mathsf{f}(e_1) \cdot e_2 \,\} & \text{if } \mathsf{E}_\alpha(e_1) = 0 \\ \{\, e' \mid (\alpha p, e') \in \mathsf{f}(e_1) \cdot e_2 \,\} \cup \{\, e' \mid (\alpha p, e') \in \mathsf{f}(e_2) \,\} & \text{if } \mathsf{E}_\alpha(e_1) = 1 \end{cases} \\
&\quad \text{applying (3.12)} \\
&= \begin{cases} \mathsf{der}_{\alpha p}(e_1) \cdot e_2 & \text{if } \mathsf{E}_\alpha(e_1) = 0 \\ \mathsf{der}_{\alpha p}(e_1) \cdot e_2 \cup \mathsf{der}_{\alpha p}(e_2) & \text{if } \mathsf{E}_\alpha(e_1) = 1 \end{cases} \\
&\quad \text{applying the induction hypotheses} \\
&= \begin{cases} \Delta_{\alpha p}(e_1) \cdot e_2 & \text{if } \mathsf{E}_\alpha(e_1) = 0 \\ \Delta_{\alpha p}(e_1) \cdot e_2 \cup \Delta_{\alpha p}(e_2) & \text{if } \mathsf{E}_\alpha(e_1) = 1 \end{cases} \\
&\quad \text{applying (3.10)} \\
&= \Delta_{\alpha p}(e_1 e_2) = \Delta_{\alpha p}(e).
\end{aligned}$$

- If $e = {e_1}^*$ then

$$\begin{aligned}
\mathsf{der}_{\alpha p}(e) &= \{\, e' \mid (\alpha p, e') \in \mathsf{f}(e_1) \cdot {e_1}^* \,\} \\
&= \{\, e' \mid (\alpha p, e') \in \mathsf{f}(e_1) \,\} \cdot {e_1}^* \\
&\quad \text{applying (3.12)} \\
&= \mathsf{der}_{\alpha p}(e_1) \cdot {e_1}^* \\
&\quad \text{applying the induction hypotheses} \\
&= \Delta_{\alpha p}(e_1) \cdot {e_1}^* \\
&\quad \text{applying (3.11)} \\
&= \Delta_{\alpha p}({e_1}^*) \;=\; \Delta_{\alpha p}(e).
\end{aligned}$$

Thus, we have proved that for every KAT expression e the two functions $\mathsf{der}_{\alpha p}$ and $\Delta_{\alpha p}$ produce the same derivatives, for the same αp. □

Similarly to what we did in Section 2.3.3, to define the decision procedure we need to consider the functions hd and $\mathsf{der}_{\alpha p}$ applied to sets of expressions. The functions $\mathsf{der}_{\alpha p}$ and hd are extended as follows:

$$\begin{aligned} \mathsf{der}_{\alpha p} &: \mathcal{P}(\mathsf{Exp}) \to \mathcal{P}(\mathsf{Exp}) \\ \mathsf{der}_{\alpha p}(E) &= \underset{e \in E}{\cup} \mathsf{der}_{\alpha p}(e) \end{aligned}$$

and

$$\begin{aligned} \mathsf{hd} &: \mathcal{P}(\mathsf{Exp}) \to \mathcal{P}(\mathsf{At} \cdot \Sigma) \\ \mathsf{hd}(E) &= \underset{e \in E}{\cup} \mathsf{hd}(e). \end{aligned}$$

Then, we define the function $\mathsf{derivatives}$ that given two sets of KAT expressions E_1 and E_2 computes all pairs of sets of partial derivatives of E_1 and E_2 w.r.t. $\alpha p \in \mathsf{At} \cdot \Sigma$, respectively.

$$\begin{aligned} \mathsf{derivatives} &: \mathcal{P}(\mathsf{Exp})^2 \to \mathcal{P}(\mathcal{P}(\mathsf{Exp})^2) \\ \mathsf{derivatives}(E_1, E_2) &= \{(\mathsf{der}_{\alpha p}(E_1), \mathsf{der}_{\alpha p}(E_2)) \mid \alpha p \in \mathsf{hd}(E_1 \cup E_2)\} \end{aligned}$$

Finally, we present the function equiv that tests if two (sets of) KAT expressions are equivalent. For two sets of KAT expressions E_1 and E_2 the function returns True if, for every α, $\mathsf{E}_\alpha(E_1) = \mathsf{E}_\alpha(E_2)$ and if, for every αp, the partial derivative of E_1 w.r.t. αp is equivalent to the partial derivative of E_2 w.r.t. αp.

$$\begin{aligned} \mathsf{equiv} : \ & \mathcal{P}(\mathcal{P}(\mathsf{Exp})^2) \times \mathcal{P}(\mathcal{P}(\mathsf{Exp})^2) \to \{\mathsf{True}, \mathsf{False}\} \\ & \mathsf{equiv}(\emptyset, H) = \mathsf{True} \\ & \mathsf{equiv}(\{(E_1, E_2)\} \cup S, H) = \begin{cases} \mathsf{False} & \text{if } \exists \alpha \in \mathsf{At} : \mathsf{E}_\alpha(E_1) \neq \mathsf{E}_\alpha(E_2) \\ \mathsf{equiv}(S \cup S', H') & \text{otherwise} \end{cases} \end{aligned}$$

where

$$S' = \{d \mid d \in \mathsf{derivatives}(E_1, E_2) \text{ and } d \notin H'\} \quad \text{and} \quad H' = \{(E_1, E_2)\} \cup H.$$

The function equiv accepts two sets S and H as arguments. At each step, S contains the pairs of (sets of) expressions that still need to be tested for equivalence, while H contains the pairs of (sets of) expressions that have already been tested. The use of the set H is important to ensure that the derivatives of the same pair of (sets of) expressions are not computed more than once, and thus prevent a possibly infinite loop.

To compare two expressions e_1 and e_2, the initial call must be $\mathsf{equiv}(\{(\{e_1\},\{e_2\})\},\emptyset)$. At each step the function takes a pair (E_1, E_2) and verifies if there exists an atom α such that $\mathsf{E}_\alpha(E_1) \neq \mathsf{E}_\alpha(E_2)$. If such an atom exists, then $e_1 \neq e_2$ and the function halts, returning False. If no such atom exists, then the function adds (E_1, E_2) to H and then replaces in S the pair (E_1, E_2) by the pairs of its corresponding derivatives, provided that these are not in H already. The return value of equiv will be the result of recursively calling equiv with the new sets as arguments. If the function ever receives $\emptyset$ as S, then the initial call ensures that $e_1 = e_2$, since all derivatives have been successfully tested, and the function returns True.

Next, we will show that the function equiv terminates. For every KAT expression e, we define the set $\mathsf{PD}(e)$ and show that, for every KAT expression e, the set of partial derivatives of e is a subset of $\mathsf{PD}(e)$, which on the other hand is clearly finite. The set $\mathsf{PD}(e)$ coincides with the *closure* of a KAT expression e, defined by Kozen, and is also similar to Mirkin's prebases [24].

$$\begin{aligned} \mathsf{PD}(b) &= \{b\} \\ \mathsf{PD}(p) &= \{p, 1\} \end{aligned} \qquad \begin{aligned} \mathsf{PD}(e_1 + e_2) &= \{e_1 + e_2\} \cup \mathsf{PD}(e_1) \cup \mathsf{PD}(e_2) \\ \mathsf{PD}(e_1 e_2) &= \{e_1 e_2\} \cup \mathsf{PD}(e_1) \cdot e_2 \cup \mathsf{PD}(e_2) \\ \mathsf{PD}(e^*) &= \{e^*\} \cup \mathsf{PD}(e) \cdot e^*. \end{aligned}$$

Lemma 3.1. *Let $e, e' \in \mathsf{Exp}$, $\alpha \in \mathsf{At}$ and $p \in \Sigma$. If $e' \in \mathsf{PD}(e)$, then $\Delta_{\alpha p}(e') \subseteq \mathsf{PD}(e)$.*

Proof. The proof is obtained by induction on the structure of e. We exemplify with the case $e = e_1 e_2$. Let $e' \in \mathsf{PD}(e_1 e_2) = \{e_1 e_2\} \cup \mathsf{PD}(e_1) \cdot e_2 \cup \mathsf{PD}(e_2)$.

- If $e' \in \{e_1 e_2\}$, then $\Delta_{\alpha p}(e') \subseteq \Delta_{\alpha p}(e_1) \cdot e_2 \cup \Delta_{\alpha p}(e_2)$. But $e_1 \in \mathsf{PD}(e_1)$ and $e_2 \in \mathsf{PD}(e_2)$, so applying the induction hypothesis twice, we obtain $\Delta_{\alpha p}(e') \subseteq \mathsf{PD}(e_1) \cdot e_2 \cup \mathsf{PD}(e_2) \subseteq \mathsf{PD}(e)$.
- If $e' \in \mathsf{PD}(e_1) \cdot e_2$, then $e' = e_1' e_2$ such that $e_1' \in \mathsf{PD}(e_1)$. So $\Delta_{\alpha p}(e') \subseteq \Delta_{\alpha p}(e_1') \cdot e_2 \cup \Delta_{\alpha p}(e_2) \subseteq \mathsf{PD}(e_1) \cdot e_2 \cup \mathsf{PD}(e_2) \subseteq \mathsf{PD}(e)$.
- Finally, if $e' \in \mathsf{PD}(e_2)$, again by the induction hypothesis we have $\Delta_{\alpha p}(e') \subseteq \mathsf{PD}(e_2) \subseteq \mathsf{PD}(e)$.

□

Proposition 3.3. *For all* $x \in (\mathsf{At} \cdot \Sigma)^*$*, one has* $\hat{\Delta}_x(e) \subseteq \mathsf{PD}(e)$.

Proof. We prove this lemma by induction on the length of x. If $|x| = 0$, i. e. $x = 1$, then $\hat{\Delta}_1(e) = \{e\} \subseteq \mathsf{PD}(e)$. If $x = w\alpha p$, then $\hat{\Delta}_{w\alpha p} = \cup_{e' \in \hat{\Delta}_w(e)} \Delta_{\alpha p}(e')$. By induction hypothesis, we know that $\hat{\Delta}_w(e) \subseteq \mathsf{PD}(e)$. By Lemma 3.1, if $e' \in \mathsf{PD}(e)$, then $\Delta_{\alpha p}(e') \subseteq \mathsf{PD}(e)$. Consequently, $\cup_{e' \in \hat{\Delta}_w(e)} \Delta_{\alpha p}(e') \subseteq \mathsf{PD}(e)$. □

Corollary 3.1. *For all KAT expressions* e*, the set* $\hat{\Delta}_{(\mathsf{At} \cdot \Sigma)^*}(e)$ *is finite.*

It is obvious that the previous results also apply to sets of KAT expressions.

Proposition 3.4. *The function* equiv *is terminating.*

Proof. When the set S is empty it follows directly from the definition of the function that it terminates. We argue that when S is not empty the function also terminates based on these two aspects:

- In order to ensure that the set of partial derivatives of a pair of (sets of) expressions are not computed more than once, the set H is used to store the ones which have already been calculated.
- Each function call removes one pair (E_1, E_2) from the set S and appends the set of partial derivatives of (E_1, E_2), which have not been calculated yet, to S. As a consequence of Corollary 3.1, the set of partial derivatives, by any word, of a set of expressions is finite, and so eventually S becomes $\emptyset$.

Thus, since at each call the function analyzes one pair from S, after a finite number of calls the function terminates. □

The next proposition states the correctness of our algorithm.

Proposition 3.5. *For all KAT expressions* e_1 *and* e_2*,*

$$\mathsf{GS}(e_1) = \mathsf{GS}(e_2) \quad \Leftrightarrow \quad \begin{cases} \mathsf{E}_\alpha(e_1) = \mathsf{E}_\alpha(e_2), & \forall \alpha \in \mathsf{At} \quad \textit{and} \\ \mathsf{GS}(\Delta_{\alpha p}(e_1)) = \mathsf{GS}(\Delta_{\alpha p}(e_2)), & \forall \alpha p \in \mathsf{At} \cdot \Sigma. \end{cases}$$

Proof. Let us first prove the $\Leftarrow$ implication. If $\mathsf{GS}(e_1) \neq \mathsf{GS}(e_2)$, then there is $x \in \mathsf{GS}$, such that $x \in \mathsf{GS}(e_1)$ and $x \notin \mathsf{GS}(e_2)$ (or vice-versa). If $x = \alpha$, then we have $\mathsf{E}_\alpha(e_1) = 1 \neq 0 = \mathsf{E}_\alpha(e_2)$ and the test fails. If $x = \alpha p w$, such that $w \in (\mathsf{At} \cdot \Sigma)^* \cdot \mathsf{At}$,

then since $\alpha pw \in \mathsf{GS}(e_1)$ and $\alpha pw \notin \mathsf{GS}(e_2)$, we have that $w \in \mathsf{GS}(\Delta_{\alpha p}(e_1))$ and $w \notin \mathsf{GS}(\Delta_{\alpha p}(e_2))$. Thus, $\mathsf{GS}(\Delta_{\alpha p}(e_1)) \neq \mathsf{GS}(\Delta_{\alpha p}(e_2))$.

Let us now prove the $\Rightarrow$ implication. For $\alpha \in \mathsf{At}$, there is either $\alpha \in \mathsf{GS}(e_1)$ and $\alpha \in \mathsf{GS}(e_2)$, thus $\mathsf{E}_\alpha(e_1) = \mathsf{E}_\alpha(e_2) = 1$; or $\alpha \notin \mathsf{GS}(e_1)$ and $\alpha \notin \mathsf{GS}(e_2)$, thus $\mathsf{E}_\alpha(e_1) = \mathsf{E}_\alpha(e_2) = 0$. For $\alpha p \in \mathsf{At} \cdot \Sigma$, by Proposition 3.1, one has $\mathsf{GS}(\Delta_{\alpha p}(e_1)) = \mathsf{GS}(\Delta_{\alpha p}(e_2))$ if and only if $\mathsf{D}_{\alpha p}(\mathsf{GS}(e_1)) = \mathsf{D}_{\alpha p}(\mathsf{GS}(e_2))$. This follows trivially from $\mathsf{GS}(e_1) = \mathsf{GS}(e_2)$. □

3.4 Pseudo-code of Relevant Implementations

In this section we present the pseudo-code for the representation of KAT expressions and for all relevant functions presented in this chapter.

3.4.1 Representation of KAT Expressions

The data structure we used to represent KAT expressions is defined as follows:

$$
\begin{array}{rcl}
\text{type } exp & = & P \text{ of int} \\
& | & B \text{ of bexp} \\
& | & Or \text{ of exp set} \\
& | & Conc \text{ of exp list} \\
& | & Star \text{ of exp} \\
\text{type } bexp & = & Zero \\
& | & One \\
& | & T \text{ of int} \\
& | & NegBexp \text{ of bexp} \\
& | & Or \text{ of bexp set} \\
& | & And \text{ of bexp set}
\end{array}
$$

Each symbol p_i of Σ or t_i of T is represented by its index i.

We represent sums, disjunctions and conjunctions by sets, since this is a natural way of ensuring the idempotence property of the operations. Concatenations are kept in lists of expressions.

3.4.2 Implementation of the Decision Procedure

This section presents the pseudo-code for all relevant functions involved in the decision procedure. Similarly to what we explained in the previous chapter, the implementation of the algorithm treats the computation of the partial derivatives differently from what follows from its formal definition. We use a function g that, for each expression e, first computes $\mathsf{f}(e)$ and then rearranges the set so that all expressions with common heads are stored in the same pair. Thus, when applied to an expression e, instead of computing a set containing all expressions e' such that $e' \in \mathsf{f}(e)$, the function $\mathsf{der}_{\alpha p}$ simply returns the set s such that $(\alpha p, s) \in \mathsf{g}(e)$, or $Zero$ if such set does not exist.

The set of atoms, at, is computed after the two expressions have been provided but before calling the decision procedure.

We begin with the pseudo-code for the function f.

$$
\begin{array}{l}
\text{let } f\ at\ exp\ =\ \text{match } exp \text{ with} \\
\qquad\quad Bexp\ b \ \longrightarrow\ \emptyset \\
\qquad |\ \ Prog\ p \ \longrightarrow\ fromListToSet\ [(atom, p, One)\ \mid\ atom \leftarrow at] \\
\qquad |\ \ Or\ set \ \longrightarrow\ \cup\ (map\ (f\ at)\ (fromSetToList\ set)) \\
\qquad |\ \ Star\ exp' \ \longrightarrow\ map\ (concat\ exp)\ (f\ at\ e') \\
\qquad |\ \ Conc\ exp_1 :: es \ \longrightarrow\ set_1\ \cup\ set_2 \\
\qquad\quad \text{where} \\
\qquad\qquad exp_2 \ \leftarrow\ buildConc\ es \\
\qquad\qquad set_1 \ \leftarrow\ map\ (concat\ exp_2)\ (f\ at\ exp_1) \\
\qquad\qquad set_2 \ \leftarrow\ f'\ at\ exp_1\ exp_2
\end{array}
$$

The function *concat* accepts an expression exp and a tuple $(atom, p, exp')$ as arguments and returns $(atom, p, exp \cdot exp')$.

The function *buildConc* constructs a concatenation of expressions from a list es given as argument, provided there are at least two expressions in the list. If es contains only one expression, then the function simply returns that element as the correct expression, instead of constructing a concatenation.

When called with arguments at, exp_1 and exp_2, the function f' behaves essentially like $f\ at\ exp_2$, except that it only allows the set to have tuples in which the atom α satisfies $\mathsf{E}_\alpha(exp_1) = 1$.

The following is the pseudo-code for the function g.

$$\text{let } g \; at \; exp = fromListToSet\; [(atom, p, buildOr\; (g'\; atom\; p\; l_e)) \mid (atom, p, \_) \leftarrow l_e]$$
$$\text{where}$$
$$l_e \leftarrow fromSetToList\; (f\; at\; exp)$$

and

$$\text{let } g'\; atom\; p\; ls = [e \mid (atom', p', e) \leftarrow ls \;\; \&\& \;\; atom = atom' \;\; \&\& \;\; p = p']$$

As the name suggests, *buildOr* is a function that receives a list of expressions and builds *Or set*, where *set* contains the elements in the list. But if the list contains only one element, then *buildOr* simply returns that element as the correct expression and no *Or* construct is created.

Now we present the pseudo-code for the function derivatives, which uses a comprehension list to compute the set of all partial derivatives of the expressions.

$$\text{let } derivatives\; set_1\; set_2 = [(atom, p\; , der_{atom\; p}\; ls_1, der_{atom\; p}\; ls_2) \mid (atom, p, \_) \leftarrow (append\; ls_1\; ls_2)]$$
$$\text{where}$$
$$ls_1 \leftarrow fromSetToList\; set_1$$
$$ls_2 \leftarrow fromSetToList\; set_2$$

and

$$\text{let } der_{atom\; p}\; l_e = \text{match } l_e \text{ with}$$
$$(atom', p', e) :: ls \longrightarrow \textbf{if } atom' = atom \;\; \&\& \;\; p' = p \textbf{ then } e \textbf{ else } der_{atom\; p}\; ls$$
$$\mid [\,] \longrightarrow Zero$$

Finally, we present the pseudo-code for the function equiv. This function uses an auxiliary function *hAll* that takes two expressions e_1 and e_2 and returns True if, for every atom α, we have $\mathsf{E}_\alpha(e_1) = \mathsf{E}_\alpha(e_2)$ and False otherwise.

let *equiv* S H =
if S=$\emptyset$ **then**
  (True, H)
else if ($hAll\; e_1\; e_2$) = False **then**
  (False, H)
  where
    (e_1, e_2) $\leftarrow$ S.firstElement
else
  $equiv\; (S \setminus (e_1, e_2) \cup S')\; H'$

where

$H' \leftarrow H \cup (e_1, e_2)$

$S' \leftarrow fromListToSet\ [(der_1, der_2) \mid (\_, \_, der_1, der_2) \leftarrow derivatives\ (g\ at\ e_1)\ (g\ at\ e_2)$ && $(der_1, der_2) \notin H']$

end if

3.5 Experimental Results

This section presents the results of some performance tests we did with our algorithm. We generated uniformly random expressions using **FAdo**, an open source software library for the symbolic manipulation of automata, regular expressions and other models of computation [10]. Following is the grammar in Polish notation we used to generate KAT expressions. The values k and l can be any integers and are specified for each sample.

$$
\begin{array}{lcl}
r & \rightarrow & + \ r \ c \mid c \\
c & \rightarrow & \cdot \ c \ s \mid s \\
s & \rightarrow & ^* s \mid q \mid p \mid t \mid \overline{t} \mid b \mid r \\
p & \rightarrow & p_1 \mid p_2 \mid \ldots \mid p_k \\
t & \rightarrow & t_1 \mid t_2 \mid \ldots \mid t_l \\
b & \rightarrow & 0 \mid 1 \\
q & \rightarrow & + \ q \ d \mid d \\
d & \rightarrow & \cdot \ d \ e \mid e \\
e & \rightarrow & \overline{q} \mid t \mid \overline{t} \mid b \mid q
\end{array}
$$

Figure 3.1: KAT Grammar in Polish notation used at the generation of uniformly random expressions.

Next we describe the two types of tests we performed. The tests were all executed on the same computer, an Intel® Xeon® 5140 at 2.33 GHz with 4 GB of RAM, running a minimal 64 bit Linux system.

In the first subsection we present some test samples of syntactically different, but at the same time equivalent, expressions. In the second subsection we present test samples of syntactically equal expressions and of unequivalent expressions.

3.5.1 Testing Equivalent Expressions

We considered some rewrite rules for KAT expressions as presented in the following scheme, where $t \in T$, $b_i \in \mathsf{Bexp}$ and $e_i \in \mathsf{Exp}$. Our purpose was to, given a generated expression e, compute a syntactically different, but yet equivalent, expression e' and then test our algorithm with these two expressions.

$$
\begin{array}{lrcl}
\textbf{(K1)} & (e_1 + e_2 + \ldots + e_n)^* & \rightarrow & (e_n{}^* \ldots e_2{}^* e_1{}^*)^* e_n{}^* \\
\textbf{(K2)} & e^* & \rightarrow & e^* e^* \\
\textbf{(K3)} & e_1(e_2 + \ldots + e_n) & \rightarrow & e_1 e_2 + \ldots + e_1 e_n \\
\textbf{(B1)} & 1 \rightarrow t + \bar{t} \qquad 0 & \rightarrow & t\bar{t} \\
 & \multicolumn{3}{c}{\text{and}} \\
 & t + \bar{t} \rightarrow 1 \qquad t\bar{t} & \rightarrow & 0 \\
\textbf{(B2)} & b_1 + b_2 b_3 \ldots b_n & \rightarrow & (b_1 + b_2)(b_1 + b_3) \ldots (b_1 + b_n)
\end{array}
$$

The rule **K1** is a generalization of the *denesting* rule [17]. The rule **K2** is an equivalence that follows directly from the definition of the star operator *. The rule **K3** is based on the axiom (2.8). The rule **B1** is based on the complementation axioms of BA. The rule **B2** is based on the BA axiom of the distributivity of the disjunction over the conjunction.

The test consisted in rewriting each expression e according to the rules above. A rule $e_1 \rightarrow e_2$ applied to a given expression e reads as follows: every occurrence in e of a subexpression of the form e_1 was replaced by the corresponding expression e_2. Expressions in which none of the substitutions could be instantiated, and therefore would result in the same expressions after application of the rules, were ignored in this test. Moreover, substitutions that would result in an expression e' with a syntactic tree larger than 2,000 were skipped.

For each expression, our procedure applied the rules **K1**, **K2**, **B1**, **B2** and **K3**, following this sequence. If the application of the rule **K1** on e resulted in an expression e' in which new substitutions were possible, the same rule was applied again, repeatedly until either no more substitutions were possible or the maximum length of 2,000 was reached. Since each substitution increased the length of the expression and the procedure stopped at a given length, this procedure was guaranteed to halt. The same was done for the rules **B2** and **K3**.

When there were no more rules to apply, the function equiv was called with the

expressions e and e'. As expected, all tests were successful.

The following table schematizes the results obtained. All test samples were performed for 10,000 expressions e. The length $|e|$ is the number of symbols in the syntactic tree of the expressions before applying the rules. The cardinalities of Σ and T (k and l, respectively) give the number of different actions and tests, respectively, available at the random generation of the expressions. The average cardinality of H gives the number of pairs of expressions that, on average, were necessary to derive to verify that e and e' were equivalent. The last columns indicate how many times, on average, each rule was applied on e to obtain e'.

				Number of Times Used (Average)				
$\|e\|$	k	l	$\|H\|$	K1	K2	K3	B1	B2
50	5	5	7.17	0.69	5.85	4.08	5.35	0.54
100	5	5	17.03	1.87	13.63	16.48	26.84	3.46

Table 3.1: Experimental results for tests of equivalent KAT expressions.

3.5.2 Tests with Random Expressions

The test samples detailed in this subsection consist in generating uniformly random expressions and testing the equivalence of each expression with itself and of every two consecutive expressions. We present some measures of the test, namely the number of computed derivatives, the number of computed atoms and the time (in seconds) at each test.

Each sample has 10,000 KAT expressions of a given length $|e|$ (number of symbols in the syntactic tree of $e \in \mathsf{Exp}$). The size of each sample is more than enough to ensure results statistically significant with 95% confidence level within a 5% error margin. For each sample we performed two experiments: (1) we tested the equivalence of each KAT expression against itself; (2) we tested the equivalence of two consecutive KAT expressions. For each pair of KAT expressions we measured: the size of the set H produced by equiv (that measures the number of iterations) and the number of primitive tests in each expression ($|e|_T$). Table 3.2 summarizes the results obtained. Each row corresponds to a sample, where the three first columns characterize the sample, respectively, the number of primitive actions (k), the number of primitive tests (l), and the length of each KAT expression generated. Column four has the number of primitive tests in each expression ($|e|_T$). Columns five and six give the

average size of H in the experiment (1) and (2), respectively. Column seven is the ratio of the equivalent pairs in experiment (2). Finally, columns eight and nine are the average times, in seconds, of each comparison in the experiments (1) and (2). These experiments aimed to test the feasibility of the procedure. As expected, the main *bottleneck* is the number of different primitive tests in the KAT expressions.

1	2	3	4	5	6	7	8	9
k	l	$\|e\|$	$\|e\|_T$	$H(1)$	$H(2)$	$=(2)$	Time(1)	Time(2)
5	5	50	9.98	7.35	0.53	0.0042	0.0097	0.00087
5	5	100	19.71	15.74	0.76	0.0048	0.0875	0.00223
10	10	50	11.12	8.30	0.50	0.0008	0.5050	0.30963
10	10	100	21.93	16.78	0.67	0.0018	20.45	1.31263
15	15	50	11.57	8.47	0.47	0.0010	6.4578	55.22

Table 3.2: Experimental results for uniformly random generated KAT expressions.

We note that in all test samples the first test took longer than the second one (since all possible derivatives of the expressions had to be calculated), except in the last sample, where Time(2) was much larger than Time(1). We interpret this discrepancy with the fact that $|\mathsf{At}|$, when computed for the first test, was $2^{l'_1}$ (where l'_1 is the number of different tests occurring in the expression e_1), whereas At in the second test (in which all different tests occurring in e_1 or e_2 had to be considered) was $2^{l'_1+l'_2}$. Thus, in average $|\mathsf{At}|$ was much larger in the second test than in the first test, which contributed to the large value of Time(2).

3.6 Encoding Programs as KAT Expressions

In this section we introduce the encoding of programs as KAT expressions and give some examples of proving the equivalence of two different programs.

We consider programs written in a minimal **while** language, where a program P is given by the following grammar:

$$
\begin{array}{rcll}
\mathsf{P} \in \mathsf{Prog} & = & x := v; & \\
& | & \mathsf{P}_1; \mathsf{P}_2 \qquad \mathsf{P}_1, \mathsf{P}_2 \in \mathsf{Prog} & \\
& | & \textbf{if } b \textbf{ then } \mathsf{P}_1 \textbf{ else } \mathsf{P}_2 \qquad b \in \mathsf{Bexp}, \mathsf{P}_1, \mathsf{P}_2 \in \mathsf{Prog} & \\
& | & \textbf{while } b \textbf{ do } \mathsf{P}_1 \qquad b \in \mathsf{Bexp}, \mathsf{P}_1 \in \mathsf{Prog} &
\end{array}
$$

In the above, x is a variable of the language and v is a value or another variable. The assignment rule is not codifiable in KAT, therefore we simulate it considering an atomic program p. If e_1 and e_2 are the encodings of programs P_1 and P_2, respectively, the encoding of more complex constructs of a **while** program involving P_1 and P_2 is given by the following rules.

$$\mathsf{P}_1 \,;\, \mathsf{P}_2 \;\Rightarrow\; e_1 e_2 \tag{3.13}$$

$$\textbf{if } b \textbf{ then } \mathsf{P}_1 \textbf{ else } \mathsf{P}_2 \;\Rightarrow\; be_1 + \bar{b}e_2 \tag{3.14}$$

$$\textbf{while } b \textbf{ do } \mathsf{P}_1 \;\Rightarrow\; (be_1)^*\bar{b} \tag{3.15}$$

Should we require a conditional test where the **else** clause is not necessary, we may omit it in the expression. We regard it as a conditional test with a dummy **else** clause 1. So we would have:

$$\textbf{if } b \textbf{ then } \mathsf{P} \;\Rightarrow\; be_p \;+\; \bar{b}\,. \tag{3.16}$$

A justification of the definitions (3.13 - 3.16) has been provided by D. Kozen and J. Tiuryn [23].

Next we present three examples of proving the equivalence of two simple programs using the algorithm we defined. For each example we present the output of the procedure, including the History set (H) containing the pairs of derivatives computed to decide about the equivalence of the programs.

Example 3.1. *Consider the two programs* P_1 *and* P_2 *below.*

P_1*:*
```
do {
    if t_1 then
        p_1;
} while t_1
```

P_2*:*
```
while t_1 do
    p_1;
```

Encoding P_1 *and* P_2 *in KAT we obtain the expressions* e_1 *and* e_2*, respectively:*

$$e_1 \;=\; (t_1p_1 + \overline{t_1})(t_1(t_1p_1 + \overline{t_1}))^*\overline{t_1}$$

and

$$e_2 \;=\; (t_1p_1)^*\overline{t_1}.$$

Applying the decision procedure to the expressions e_1 *and* e_2 *we obtain:*

> Answer : True
> History : $\{(\mathsf{t}_1(\bar{\mathsf{t}}_1+\mathsf{t}_1\mathsf{p}_1)^*\bar{\mathsf{t}}_1, \quad (\mathsf{t}_1\mathsf{p}_1)^*\bar{\mathsf{t}}_1),$
$((\bar{\mathsf{t}}_1+\mathsf{t}_1\mathsf{p}_1)(\mathsf{t}_1(\bar{\mathsf{t}}_1+\mathsf{t}_1\mathsf{p}_1))^*\bar{\mathsf{t}}_1, \quad (\mathsf{t}_1\mathsf{p}_1)^*\bar{\mathsf{t}}_1)\}$

Example 3.2. *In the following,* P_3 *and* P_4 *correspond to the programs (34) and (35), respectively, in [17], adapted to the notation we use.*

P_3*:*
```
while t_1 do begin
      p_1;
      while t_2 do p_2;
   end
```

P_4*:*
```
if t_1 then begin
      p_1;
      while (t_1+t_2) do
          if t_2 then p_2; else p_1;
   end
```

The KAT expressions corresponding to the programs P_3 *and* P_4 *are respectively*

$$e_3 \;=\; (t_1p_1(t_2p_2)^*\overline{t_2})^*\overline{t_1}$$

and

$$e_4 \;=\; t_1p_1((t_1+t_2)(t_2p_2+\overline{t_2}p_1))^*\overline{t_1+t_2}+\overline{t_1}.$$

Applying the decision procedure to the expressions e_3 *and* e_4 *we obtain:*

> Answer : True
> History : $\{((\mathsf{t}_1\mathsf{p}_1(\mathsf{t}_2\mathsf{p}_2)^*\overline{\mathsf{t}_2})^*\overline{\mathsf{t}_1}, \quad \overline{\mathsf{t}_1}+\mathsf{t}_1\mathsf{p}_1((\mathsf{t}_1+\mathsf{t}_2)(\mathsf{t}_2\mathsf{p}_2+\overline{\mathsf{t}_2}\mathsf{p}_1))^*\overline{\mathsf{t}_1+\mathsf{t}_2}),$
$((\mathsf{t}_2\mathsf{p}_2)^*\overline{\mathsf{t}_2}(\mathsf{t}_1\mathsf{p}_1(\mathsf{t}_2\mathsf{p}_2)^*\overline{\mathsf{t}_2})^*\overline{\mathsf{t}_1}, \quad ((\mathsf{t}_1+\mathsf{t}_2)(\mathsf{t}_2\mathsf{p}_2+\overline{\mathsf{t}_2}\mathsf{p}_1))^*\overline{\mathsf{t}_1+\mathsf{t}_2})\}$

Example 3.3. *The following annotated programs are from [17], Section 3.5.*

P_5*:*
```
p_3;
t_1t_2+~t_1~t_2
while t_1 do begin
  p_1;p_3;
  t_1t_2+~t_1~t_2
end
p_2;
```

P_6*:*
```
p_3;
t_1t_2+~t_1~t_2
while t_2 do begin
  p_1;p_3;
  t_1t_2+~t_1~t_2
end
p_2;
```

The KAT expressions corresponding to the programs P_5 *and* P_6 *are respectively*

$$e_5 \;=\; (t_1t_2+\overline{t_1t_2})(t_1p_1(t_1t_2+\overline{t_1t_2}))^*\overline{t_1}$$

and

$$e_6 \;=\; (t_1t_2 + \overline{t_1t_2})(t_2p_1(t_1t_2 + \overline{t_1t_2}))^*\overline{t_2}\ .$$

Calling our program with the expressions e_5 and e_6 we obtain:

> Answer : True
> History : $\{((\mathsf{t_1t_2} + \overline{\mathsf{t_1t_2}})(\mathsf{t_1p_1}(\mathsf{t_1t_2} + \overline{\mathsf{t_1t_2}}))^*\overline{\mathsf{t_1}},\quad ((\mathsf{t_1t_2} + \overline{\mathsf{t_1t_2}})(\mathsf{t_2p_1}(\mathsf{t_1t_2} + \overline{\mathsf{t_1t_2}}))^*\overline{\mathsf{t_2}})\}$

Chapter 4

Deciding Hoare Logic with KAT

4.1 Introduction

Hoare logic was first introduced in 1969, cf. [12], and is a formal system widely used for the specification and verification of programs. Hoare logic uses triples to reason about the correctness of programs. A triple is an assertion of the form $\{b_1\}\mathsf{P}\{b_2\}$ with P being a program, and b_1 and b_2 logic formulas. We read such an assertion as *if* b_1 *holds before the execution of* P*, then* b_2 *will necessarily hold at the end of the execution.* A deductive system of Hoare logic provides inference rules for deriving valid triples, where rules depend on the program constructs. We consider programs written in the **while** language as referred in Section 3.6.

Let us now consider the traditional Hoare logic system for partial correctness. Its inference rules are defined as follows:

Assignment

$$\{b[x \mapsto e]\} \;\; x := e \;\; \{b\}$$

Composition

$$\frac{\{b_1\}\ \mathsf{P}_1\ \{b_2\} \quad \{b_2\}\ \mathsf{P}_2\ \{b_3\}}{\{b_1\}\ \mathsf{P}_1\ ;\ \mathsf{P}_2\ \{b_3\}}$$

Conditional

$$\frac{\{b_1 \wedge b_2\}\ \mathsf{P}_1\ \{b_3\} \quad \{\neg b_1 \wedge b_2\}\ \mathsf{P}_2\ \{b_3\}}{\{b_2\}\ \textbf{if}\ b_1\ \textbf{then}\ \mathsf{P}_1\ \textbf{else}\ \mathsf{P}_2\ \{b_3\}}$$

While

$$\frac{\{b_1 \wedge b_2\}\ \mathsf{P}_1\ \{b_2\}}{\{b_2\}\ \textbf{while}\ b_1\ \textbf{do}\ \mathsf{P}_1\ \{\neg b_1 \wedge b_2\}}$$

Weakening

$$\frac{b_1' \to b_1 \quad \{b_1\}\ \mathsf{P}_1\ \{b_2\} \quad b_2 \to b_2'}{\{b_1'\}\ \mathsf{P}_1\ \{b_2'\}}$$

A triple $\{b_1\}\mathsf{P}\{b_2\}$ is called a *partial correctness assertion* and can be deduced using the inference rules above, which can be done manually. However, we would like to make this process automatic. In order to do that we need a system that has the *sub-formula* property, i. e., a system in which the premises of a rule do not contain occurrences of assertions (formulas) that do not occur in the conclusion of the rule. The traditional Hoare logic system clearly does not satisfy this property, but the variation system considered by M. Frade and J. Pinto serves our purpose [11]. Since the program P must be annotated (which can be done using a WP algorithm [11]), the correction assertions are no longer triples. The inference rules for this system are the following:

Skip

$$\frac{b_1 \to b_2}{\{b_1\}\ \textbf{skip}\ \{b_2\}}$$

Assignment

$$\frac{b_1 \to b_2[x \mapsto e]}{\{b_1\}\ \ x := e\ \ \{b_2\}}$$

Composition

$$\frac{\{b_1\}\ \mathsf{P}_1\ \{b_2\} \quad \{b_2\}\ \mathsf{P}_2\ \{b_3\}}{\{b_1\}\ \mathsf{P}_1\ ; \{b_2\}\ \mathsf{P}_2\ \{b_3\}}$$

Conditional

$$\frac{\{b_1 \wedge b_2\}\ \mathsf{P}_1\ \{b_3\} \quad \{\neg b_1 \wedge b_2\}\ \mathsf{P}_2\ \{b_3\}}{\{b_2\}\ \textbf{if}\ b_1\ \textbf{then}\ \mathsf{P}_1\ \textbf{else}\ \mathsf{P}_2\ \{b_3\}}$$

While

$$\frac{\{b_1 \wedge b_i\}\ \mathsf{P}_1\ \{b_i\} \quad b_2 \to b_i \quad (b_i \wedge \neg b_1) \to b_3}{\{b_2\}\ \textbf{while}\ b_1\ \textbf{do}\ \{b_i\}\mathsf{P}_1\ \{b_3\}}$$

4.2 Encoding Propositional Hoare Logic in KAT

The propositional fragment of Hoare logic (PHL), i. e., the fragment without the rule for assignment, can be encoded in KAT [18]. The encoding of an annotated **while** program P and of our inference system follow the same lines. The **skip** command is encoded by a distinguished primitive symbol $p_{\mathbf{skip}}$. We extend the encoding rules presented in 3.6 to **while** programs with annotations.

$$\begin{aligned} \mathsf{P}_1 \,;\{b\}\, \mathsf{P}_2 &\;\Rightarrow\; e_1 b e_2 \\ \textbf{if } b \textbf{ then } \mathsf{P}_1 \textbf{ else } \mathsf{P}_2 &\;\Rightarrow\; b e_1 + \bar{b} e_2 \\ \textbf{while } b \textbf{ do } \{b_i\}\, \mathsf{P}_1 &\;\Rightarrow\; (b b_i e_1)^* \bar{b} \end{aligned}$$

Since the assignment construct is not codifiable in KAT, we simulate such instructions by considering not only verification conditions but also atomic PCA's $\{b_1'\}x := v\{b_2'\}$.

A PCA of the form $\{b_1\}\mathsf{P}\{b_2\}$ is encoded in KAT as an equational identity of the form

$$b_1 e = b_1 e b_2 \qquad \text{or equivalently by} \qquad b_1 e \overline{b_2} = 0,$$

where e is the encoding of the program P. Intuitively, we are saying that if b_1 holds, then it is redundant to test b_2 after the execution of p because it will necessarily hold. The rules above are encoded as the following equational implications:

Composition

$$b_1 e_1 = b_1 e_1 b_2 \wedge b_2 e_2 = b_2 e_2 b_3 \;\;\rightarrow\;\; b_1 e_1 b_2 e_2 = b_1 e_1 b_2 e_2 b_3 \tag{4.1}$$

Conditional

$$b_1 b_2 e_1 = b_1 b_2 e_1 b_3 \wedge \overline{b_1} b_2 e_2 = \overline{b_1} b_2 e_2 b_3 \;\;\rightarrow\;\; b_2(b_1 e_1 + \overline{b_1} e_2) = b_2(b_1 e_1 + \overline{b_1} e_2) b_3 \tag{4.2}$$

While

$$b_1 b_i e_1 = b_1 b_i e_1 b_i \wedge b_2 \leq b_i \wedge b_i \overline{b_1} \leq b_3 \;\;\rightarrow\;\; b_2 (b_1 b_i e_1)^* \overline{b_1} = b_2 (b_1 b_i e_1)^* \overline{b_1} b_3 \tag{4.3}$$

Now, suppose we want to prove the PCA $\{b_1\}\mathsf{P}\{b_2\}$. Since the inference system for Hoare logic that we are using possesses the sub-formula property, one can generate mechanically in a backward fashion the verification conditions that ensure the validity of the PCA.

Since in the KAT encoding we do not have the rule for assignment, besides verification conditions (proof obligations) of the form $b'_1 \to b'_2$ we will also have assumptions of the form $b'_1 p \overline{b'_2} = 0$.

One can generate a set of assumptions, $\Gamma = \mathsf{Gen}(b_1 e \overline{b_2})$, backwards from $b_1 e \overline{b_2} = 0$, where Gen is inductively defined by:

$$\begin{array}{rcll}
\mathsf{Gen}(b_1\ p_{\mathbf{skip}}\ \overline{b_2}) & = & \{b_1 \leq b_2\} & \\
\mathsf{Gen}(b_1\ p\ \overline{b_2}) & = & \{b_1\ p\ \overline{b_2}\} & p_{\mathbf{skip}} \neq p \in \Sigma \\
\mathsf{Gen}(b_1\ e_1\ b_2\ e_2\ \overline{b_3}) & = & \mathsf{Gen}(b_1\ e_1\ \overline{b_2}) \cup \mathsf{Gen}(b_2\ e_2\ \overline{b_3}) & \\
\mathsf{Gen}(b_1\ (b_2 e_1 + \overline{b_2} e_2)\ \overline{b_3}) & = & \mathsf{Gen}(b_1 b_2\ e_1\ \overline{b_3}) \cup \mathsf{Gen}(b_1 \overline{b_2}\ e_2\ \overline{b_3}) & \\
\mathsf{Gen}(b_1\ ((b_2 b_i e)^* \overline{b_2})\ \overline{b_3}) & = & \mathsf{Gen}(b_i b_2\ e\ \overline{b_i}) \cup \{b_1 \leq b_i, b_i \overline{b_2} \leq b_3\} &
\end{array}$$

Note that Γ is necessarily of the form

$$\Gamma = \{b_1 p_1 \overline{b'_1} = 0, \ldots,\ b_m p_m \overline{b'_m} = 0\} \cup \{c_1 \leq c'_1, \ldots, c_n \leq c'_n\},$$

where $p_1, \ldots, p_m \in \Sigma$ and such that all b's and c's are Bexp expressions. In Section 4.3, we show how one can prove the validity of $b_1 e_P \overline{b_2} = 0$ in the presence of such a set of assumptions Γ, but first let us illustrate the encoding and generation of the assumption set with a few small examples.

4.2.1 Some Small Examples

In this section we present some examples of encoding small annotated programs and their corresponding sets of assumptions as KAT expressions.

Example 4.1. *Consider the program* $\mathsf{P_1}$ *in Table 4.1, that calculates the triple of a number. We wish to prove that, at the end of the execution, the variable* y *contains the result of* $3x$*. Thus, we aim to verify the assertion* $\{\mathsf{True}\}\ \mathsf{P_1}\ \{y = 3x\}$*.*

Program $\mathsf{P_1}$	Annotated Program $\mathsf{P'_1}$	Symbols used in the encoding
	$\{x + x + x = 3x\}$	t_1
$y := x;$	$y := x;$	p_1
$y := x + x + y;$	$\{x + x + y = 3x\}$	t_2
	$y := x + x + y;$	p_2

Table 4.1: A program to calculate the triple of a number

In order to apply the inference rules we need to annotate program P_1*, obtaining program* P'_1*, which can be done using a WP algorithm. Manually, applying the inference rules for deriving PCA's in a backward fashion to* $\{\mathsf{True}\}\ \mathsf{P}'_1\ \{y = 3x\}$*, one easily generates the corresponding set of assumptions provided by the annotated version of the program. Thus the assumption set is*

$$\Gamma_{P_1} = \left\{ \begin{array}{l} \{\mathsf{True}\} \rightarrow \{x+x+x=3x\}, \{x+x+x=3x\}y := x\{x+x+y=3x\}, \\ \{x+x+y=3x\}y := x+x+y\{y=3x\} \end{array} \right\}.$$

On the other hand, using the correspondence of KAT primitive symbols and atomic parts of the annotated program P'_1*, as in Table 4.1, and additionally encoding* True *as* t_0 *and* $y = 3x$ *as* t_3*, the encoding of* $\{\mathsf{True}\}\ \mathsf{P}'_1\ \{y = 3x\}$ *in KAT is*

$$t_0 t_1 p_1 t_2 p_2 \overline{t_3} = 0. \tag{4.4}$$

The corresponding set of assumptions Γ *in KAT is*

$$\Gamma = \{t_0 \leq t_1, t_1 p_1 \overline{t_2} = 0, t_2 p_2 \overline{t_3} = 0\}. \tag{4.5}$$

Example 4.2. *Consider the program* P_2 *in Table 4.2, that finds the greatest of two numbers. We wish to prove that, at the end of the execution, the variable* MAX *contains the greatest number between* x *and* y*, i. e. to verify the assertion* $\{\mathsf{True}\}\ \mathsf{P}_2\ \{MAX = max(x, y)\}$*.*

Program P_2	Symbols used in the encoding
if $x >= y$ then	t_1
$\mathsf{MAX} := x$;	p_1
else	
$\mathsf{MAX} := y$;	p_2

Table 4.2: A program to find the greatest of two numbers

Since this is a simple if-then-else case, the program is the same annotated or not annotated. Applying the inference rules for deriving PCA's to $\{\mathsf{True}\}\ \mathsf{P}_2\ \{MAX = max(x, y)\}$*, the set of assumptions we obtain is*

$$\Gamma_{P_2} = \left\{ \begin{array}{l} (\mathsf{True} \wedge x \geq y) \rightarrow MAX = max(x, y), \\ (\mathsf{True} \wedge \neg(x \geq y)) \rightarrow MAX = max(x, y) \end{array} \right\}.$$

Using the correspondence of KAT primitive symbols and atomic parts of program P_2*, as in Table 4.2, and additionally encoding* True *as* t_0 *and* $MAX = max(x, y)$ *as* t_2*, the encoding of* $\{\mathsf{True}\}\ \mathsf{P}_2\ \{MAX = max(x, y)\}$ *in KAT is*

$$t_0(t_1p_1 + \overline{t_1}p_2)\overline{t_3} = 0. \tag{4.6}$$

The corresponding set of assumptions Γ *in KAT is*

$$\Gamma = \{t_0t_1p_1\overline{t_3} = 0, t_0\overline{t_1}p_2\overline{t_3} = 0\}. \tag{4.7}$$

Example 4.3. *Consider the program* P_3 *in Table 4.3, that calculates the factorial of a non-negative integer. We wish to prove that, at the end of the execution, the variable* y *contains the factorial of* x*, i. e. to verify the assertion* $\{\mathsf{True}\}\ \mathsf{P}_3\ \{y = x!\}$*.*

Program P_3	Annotated Program P'_3	Symbols used in the encoding
	$y := 1;$	p_1
	$\{y = 0!\}$	t_1
$y := 1;$	$z := 0;$	p_2
$z := 0;$	$\{y = z!\}$	t_2
while $\neg z = x$ do	while $\neg z = x$ do	t_3
{	{	
z := z+1;	{y=z!}	t_2
y := y×z;	z := z+1;	p_3
}	{y×z = z!}	t_4
	y := y×z;	p_4
	}	

Table 4.3: A program for the factorial

In order to apply the inference rules we need to annotate program P_3*, obtaining program* P'_3*. Applying the inference rules for deriving PCA's in a backward fashion to* $\{\mathsf{True}\}\ \mathsf{P}'_3\ \{y = x!\}$*, the set of assumptions we obtain is*

$$\Gamma_{P_3} = \left\{ \begin{array}{l} \{\mathsf{True}\}y := 1\{y = 0!\}, \{\mathsf{y = 0!}\}z := 0\{y = z!\}, \\ \{\mathsf{y = z!} \wedge \neg(\mathsf{z = x})\}z := z + 1\{y \times z = z!\}, \{y \times z = z!\}y := y \times z\{y = z!\}, \\ y = z! \rightarrow y = z!, (y = z! \wedge \neg\neg(z = x)) \rightarrow y = x! \end{array} \right\}.$$

Using the correspondence of KAT primitive symbols and atomic parts of the annotated program P'_3*, as in Table 4.3, and additionally encoding* True *as* t_0 *and* $y = x!$ *as* t_5*,*

the encoding of $\{\mathsf{True}\}\ \mathsf{P}'_3\ \{y = x!\}$ *in KAT is*

$$t_0p_1t_1p_2t_2(t_3t_2p_3t_4p_4)^*\overline{t_3}t_5 = 0. \tag{4.8}$$

The corresponding set of assumptions Γ *in KAT is*

$$\Gamma = \{t_0p_1\overline{t_1} = 0, t_1p_2\overline{t_2} = 0, t_2t_3p_3\overline{t_4} = 0, t_4p_4\overline{t_2} = 0, t_2 \leq t_2, t_2\overline{t_3} \leq t_5\}. \tag{4.9}$$

4.3 Deciding Hoare Logic

Rephrasing the observations in the last section, we are interested in proving in KAT the validity of implications of the form

$$b_1p_1\overline{b'_1} = 0\ \wedge\cdots\wedge\ b_mp_m\overline{b'_m} = 0\ \wedge\ c_1\overline{c'_1} = 0\ \wedge\cdots\wedge c_n\overline{c'_n} = 0\ \ \rightarrow\ \ bp\overline{b'} = 0. \tag{4.10}$$

Note that $c\overline{c'} = 0$ is just a different way of writing $c \leq c'$. The expression (4.10) can be reduced to proving the equivalence of two KAT expressions, since it has been shown, cf. [18], that for all KAT expressions $r_1, \ldots, r_n, e_1, e_2$ over $\Sigma = \{p_1, \ldots, p_k\}$ and $T = \{t_1, \ldots, t_l\}$, an implication of the form

$$r_1 = 0\ \wedge\cdots\wedge\ r_n = 0\ \ \rightarrow\ \ e_1 = e_2$$

is a theorem of KAT if and only if

$$e_1 + uru = e_2 + uru \tag{4.11}$$

where $u = (p_1 + \cdots + p_k)^*$ and $r = r_1 + \ldots + r_n$. Testing this last equality can of course be done by applying our algorithm to $e_1 + uru$ and $e_2 + uru$. However, in the next subsection, we present an alternative method of proving the validity of implications of the form 4.10. This method has the advantage of prescinding from the expressions u and r, above.

4.3.1 Equivalence of KAT Expressions Modulo a Set of Assumptions

In the presence of a finite set of assumptions of the form

$$\Gamma = \{b_1p_1\overline{b'_1} = 0, \ldots, b_mp_m\overline{b'_m} = 0\} \cup \{c_1 \leq c'_1, \ldots, c_n \leq c'_n\} \tag{4.12}$$

we have to restrict ourselves to atoms that satisfy the restrictions in Γ. Thus, let

$$\mathsf{At}^\Gamma = \{\ \alpha \in \mathsf{At} \mid \alpha \leq c \to \alpha \leq c',\ \text{for all } c \leq c' \in \Gamma\ \}. \tag{4.13}$$

Given a KAT expression e, the *set of guarded strings modulo* Γ, $\mathsf{GS}^\Gamma(e)$, is inductively defined as follows.

$$\begin{array}{rcl}
\mathsf{GS}^\Gamma(p) & = & \{\ \alpha p \beta \mid \alpha, \beta \in \mathsf{At}^\Gamma \wedge \forall_{bp\overline{b'}=0\,\in\,\Gamma}\ (\alpha \leq b \to \beta \leq b')\ \} \\
\mathsf{GS}^\Gamma(b) & = & \{\ \alpha \in \mathsf{At}^\Gamma \mid \alpha \leq b\ \} \\
\mathsf{GS}^\Gamma(e_1 + e_2) & = & \mathsf{GS}^\Gamma(e_1) \cup \mathsf{GS}^\Gamma(e_2) \\
\mathsf{GS}^\Gamma(e_1 e_2) & = & \mathsf{GS}^\Gamma(e_1) \diamond \mathsf{GS}^\Gamma(e_2) \\
\mathsf{GS}^\Gamma(e^*) & = & \cup_{n\geq 0} \mathsf{GS}^\Gamma(e)^n.
\end{array}$$

The following proposition characterizes the *equivalence* modulo a set of assumptions Γ, and ensures the correctness of the new Hoare logic decision procedure.

Proposition 4.1. *Let e_1 and e_2 be KAT expressions and Γ a set of assumptions as in (4.12). Then,*

$$\mathsf{KAT}, \Gamma \vdash e_1 = e_2 \qquad \textit{iff} \qquad \mathsf{GS}^\Gamma(e_1) = \mathsf{GS}^\Gamma(e_2).$$

Proof. By (4.11) one has $\mathsf{KAT}, \Gamma \vdash e_1 = e_2$ if and only if $e_1 + uru = e_2 + uru$ is provable in KAT, where $u = (p_1 + \cdots + p_k)^*$ and $r = b_1 p_1 \overline{b'_1} + \cdots + b_m p_m \overline{b'_m} + c_1 \overline{c'_1} + \cdots + c_n \overline{c'_n}$. The second equality is equivalent to $\mathsf{GS}(e_1 + uru) = \mathsf{GS}(e_2 + uru)$, i. e. $\mathsf{GS}(e_1) \cup \mathsf{GS}(uru) = \mathsf{GS}(e_2) \cup \mathsf{GS}(uru)$. In order to show the equivalence of this last equality and $\mathsf{GS}^\Gamma(e_1) = \mathsf{GS}^\Gamma(e_2)$, it is sufficient to show that for every KAT expression e one has $\mathsf{GS}^\Gamma(e) = \mathsf{GS}(e) \setminus \mathsf{GS}(uru)$ (note that $A \cup C = B \cup C \ \Leftrightarrow\ A \setminus C = B \setminus C$).

First we analyze under which conditions a guarded string x is an element of $\mathsf{GS}(uru)$. Given the values of u and r, it is easy to see that $x \in \mathsf{GS}(uru)$ if and only if in x occurs an atom α such that $\alpha \leq c$ and $\alpha \not\leq c'$ for some $c \leq c' \in \Gamma$, or x has a substring $\alpha p \beta$, such that $\alpha \leq b$ and $\alpha \not\leq b'$ for some $bp\overline{b'} \in \Gamma$. This means that $x \notin \mathsf{GS}(uru)$ if and only if every atom in x is an element of At^Γ and every substring $\alpha p \beta$ of x satisfies $(\alpha \leq b \to \beta \leq b')$, for all $bp\overline{b'} = 0 \in \Gamma$. From this remark and by the definitions of At^Γ and GS^Γ, we conclude that $\mathsf{GS}^\Gamma(e) \cap \mathsf{GS}(uru) = \emptyset$. Note also that, since $\mathsf{GS}^\Gamma(e)$ is a restriction of $\mathsf{GS}(e)$, one has $\mathsf{GS}^\Gamma(e) \subseteq \mathsf{GS}(e)$. Now it suffices to show that for every $x \in \mathsf{GS}(e) \setminus \mathsf{GS}(uru)$, one has $x \in \mathsf{GS}^\Gamma(e)$. This can be easily proved by induction on the structure of e. □

We now define the set of partial derivatives of a KAT expression modulo a set of assumptions Γ. Let $e \in \mathsf{Exp}$. If $\alpha \notin \mathsf{At}^\Gamma$, then $\Delta^\Gamma_{\alpha p}(e) = \emptyset$. For $\alpha \in \mathsf{At}^\Gamma$, let

$$\begin{aligned}
\Delta^\Gamma_{\alpha p}(p') &= \begin{cases} \{\Pi b' \mid bp\overline{b'} = 0 \in \Gamma \wedge \alpha \leq b\ \} & \text{if } p = p' \\ \emptyset & \text{if } p \neq p' \end{cases} \\
\Delta^\Gamma_{\alpha p}(b) &= \emptyset \\
\Delta^\Gamma_{\alpha p}(e_1 + e_2) &= \Delta^\Gamma_{\alpha p}(e_1) \cup \Delta^\Gamma_{\alpha p}(e_2) \\
\Delta^\Gamma_{\alpha p}(e_1 e_2) &= \begin{cases} \Delta^\Gamma_{\alpha p}(e_1) \cdot e_2 & \text{if } \mathsf{E}_\alpha(e_1) = 0 \\ \Delta^\Gamma_{\alpha p}(e_1) \cdot e_2 \cup \Delta^\Gamma_{\alpha p}(e_2) & \text{if } \mathsf{E}_\alpha(e_1) = 1 \end{cases} \\
\Delta^\Gamma_{\alpha p}(e^*) &= \Delta^\Gamma_{\alpha p}(e) \cdot e^*.
\end{aligned}$$

Note, that by definition, $\Pi\ b' = 1$ if there is no $bp = bpb' \in \Gamma$ such that $\alpha \leq b$ and $\alpha \in \mathsf{At}^\Gamma$. The next proposition states the correctness of the definition of $\Delta^\Gamma_{\alpha p}$.

Proposition 4.2. *Let Γ be a set of assumptions as above, $e \in \mathsf{Exp}$, $\alpha \in \mathsf{At}$, and $p \in \Sigma$. Then,*

$$\mathsf{D}_{\alpha p}(\mathsf{GS}^\Gamma(e)) = \mathsf{GS}^\Gamma(\Delta^\Gamma_{\alpha p}(e)).$$

Proof. The proof is obtained by induction on the structure of e. We only show the case $e = p$, since the other cases are similar to those in the proof of Proposition 3.1. If $\alpha \notin \mathsf{At}^\Gamma$, then $\mathsf{GS}^\Gamma(p) = \emptyset = \mathsf{D}_{\alpha p}(\mathsf{GS}^\Gamma(p))$. Also, $\Delta^\Gamma_{\alpha p}(p) = \emptyset = \mathsf{GS}^\Gamma(\Delta^\Gamma_{\alpha p}(p))$. Otherwise, if $\alpha \in \mathsf{At}^\Gamma$, then $\mathsf{GS}^\Gamma(p) = \{\alpha p \beta \mid \alpha, \beta \in \mathsf{At}^\Gamma \wedge \forall_{bp\overline{b'}=0\ \in\ \Gamma}\ (\alpha \leq b \rightarrow \beta \leq b')\}$, thus $\mathsf{D}_{\alpha p}(\mathsf{GS}^\Gamma(p)) = \{\beta \in \mathsf{At}^\Gamma \mid \beta \leq b' \text{ for all } bp\overline{b'} = 0\ \in\ \Gamma \text{ such that } \alpha \leq b\}$. On the other hand, $\Delta^\Gamma_{\alpha p}(p) = \{\Pi b' \mid bp\overline{b'} = 0 \in \Gamma \wedge \alpha \leq b\}$. Thus, $\mathsf{GS}^\Gamma(\Delta^\Gamma_{\alpha p}(p)) = \mathsf{GS}^\Gamma(c)$, where $c = \prod_{bp\overline{b'}=0\ \in\ \Gamma, \alpha \leq b} b'$. We conclude that $\mathsf{GS}^\Gamma(c) = \{\beta \in \mathsf{At}^\Gamma \mid \beta \leq b' \text{ for all } bp\overline{b'} = 0\ \in\ \Gamma \text{ such that } \alpha \leq b\}$. □

4.3.2 Testing Equivalence Modulo a Set of Assumptions

The decision procedure for testing equivalence presented before can be easily adapted. Given a set of assumptions Γ, the set At^Γ is obtained by discarding in At all atoms that satisfy c but do not satisfy c', for all $c \leq c' \in \Gamma$. The function f has to account for the new definition of $\Delta^\Gamma_{\alpha p}$.

We compared this new algorithm, equiv^Γ, with equiv when deciding the PCA's presented in Section 4.2.1. The results are as follows:

Example 4.1

First, we constructed expressions r and u from Γ as described above and proved the equivalence of expressions $t_0t_1p_1t_2p_2\overline{t_3} + uru$ and $0 + uru$, with function equiv. In this case $|H| = 10$. In other words, equiv needed to derive 10 pairs of expressions in order to reach a conclusion about the correction of program $\mathsf{P_1}$. Then, we applied function equiv^Γ directly to the pair $(t_0t_1p_1t_2p_2\overline{t_3}, 0)$ and Γ. In this case, $|H| = 3$.

Example 4.2

Again, we constructed the expressions r and u from Γ and proved the equivalence of expressions $t_0(t_1p_1 + \overline{t_1}p_2)\overline{t_3} + uru$ and $0 + uru$, with function equiv. In this case $|H| = 6$. Then, we applied function equiv^Γ directly to the pair $(t_0(t_1p_1 + \overline{t_1}p_2)\overline{t_3}, 0)$ and Γ. In this case, $|H| = 2$.

Example 4.3

In the third example, we proved the equivalence of expressions $t_0p_1t_1p_2t_2(t_3t_2p_3t_4p_4)^*\overline{t_3t_5}+ uru$ and $0+uru$, with function equiv. In this case $|H| = 17$. Then, we applied function equiv^Γ directly to the pair $(t_0p_1t_1p_2t_2(t_3t_2p_3t_4p_4)^*\overline{t_3t_5}, 0)$ and Γ. In this case, $|H| = 5$.

Thus, in the three examples the second method required the computation of approximately less than a third of the number of derivatives required by the first method.

4.4 Commutativity Conditions

Besides the problem of deciding Hoare logic, there are other proofs of equivalence in KAT that can only be achieved in the presence of assumptions. In this section we present the equivalence of KAT expressions that depends on *commutativity conditions*.

Definition 4.1. *In any KAT, given a symbol p of Σ and a logic expression b of* Bexp*, we say that p and b commute if and only if*

$$bp = pb.$$

Intuitively, we are saying that if the program p does not affect the value of b, then it is indifferent to test b before or after the execution of p, because its value will be the same.

A different way of writing $bp = pb$ is $bp\bar{b} = 0 \wedge \bar{b}pb = 0$.

In the first subsection we present a small example of two programs that can only be proved equivalent if such a commutativity condition holds. In the second subsection we present a more complex example in which commutativity conditions are essential in proving the safety of a code fragment that acquires and releases a lock on a resource.

4.4.1 A Simple Example

The following correspond to the programs (21) and (22) from [17].

```
P1: if t_1 then
      begin p_1;p_2; end
    else
      begin p_1;p_3; end
```

```
P2: p_1;
    if t_1 then
      p_2;
    else
      p_3;
```

The KAT expressions corresponding to the programs P_1 and P_2 are respectively

$$e_1 \;=\; t_1p_1p_2 + \overline{t_1}p_1p_3$$

and

$$e_2 \;=\; p_1(t_1p_2 + \overline{t_1}p_3)\ .$$

In the program above, we do not know if the value of t_1 is preserved by p_1, and therefore if we test the equivalence giving the program nothing more than the expressions e_1 and e_2 the result is False, as one would expect. The solution is to give the following commutativity condition as an assumption: $\Gamma \;=\; \{t_1p_1\overline{t_1}{=}0, \overline{t_1}p_1t_1{=}0\}$. In this case, calling the function equiv^Γ with the expressions e_1 and e_2 and the axioms set Γ, the program correctly returns True.

4.4.2 Proving the Safety of a Program

The following code fragment is from [7]. It consists of a loop that alternately acquires and releases a lock on a resource.

If the driver currently holds the lock and tries to reacquire it, the driver will hang. The same happens if the driver tries to release the lock when it does not hold it. We wish to prove that the program is safe in that the driver never attempts to acquire the

Program P	Symbols used in the encoding
`do {`	
`  KeAcquireSpinLock();`	p_1
`  nPacketsOld = nPackets;`	p_2
`  if (request) {`	t_1
`  request = request->next;`	p_3
`  KeReleaseSpinLock();`	p_4
`  nPackets++;`	p_5
`  }`	
`} while (nPackets != nPacketsOld)`	$\overline{t_2}$
`KeReleaseSpinLock();`	p_4

Table 4.4: A code fragment from a device driver

lock when it is already in the locked state and never attempts to release the lock when it is not in the locked state. The definition of commutativity conditions and other assumptions is essential to the proof of safety and to do that we follow the approach taken by Kozen [20].

First, we need to encode the program as a KAT expression. This requires extending the list (3.13)–(3.15) of encoding rules for program constructs with a new rule [20].

$$\textbf{do } p; \textbf{while } b \;\Rightarrow\; p(bp)^*\overline{b}\,.$$

Now P can be encoded as the following expression.

$$e \;=\; p_1p_2(t_1p_3p_4p_5 + \overline{t_1})(\overline{t_2}p_1p_2(t_1p_3p_4p_5 + \overline{t_1}))^*t_2p_4.$$

Let t_0 be a new test representing the assertion that the driver is in the locked state. We wish to guarantee both that before the driver acquires a lock it is not in the locked state (t_0 is False) and that before it releases the lock it is in the locked state (t_0 is True). So, the expression corresponding to the annotated program is as follows:

$$e_A \;=\; \overline{t_0}p_1p_2(t_1p_3t_0p_4p_5 + \overline{t_1})(\overline{t_2t_0}p_1p_2(t_1p_3t_0p_4p_5 + \overline{t_1}))^*t_2t_0p_4$$

To show that the program is safe, we need to prove that the following equivalence holds:

$$\overline{t_0}e \;=\; \overline{t_0}e_A. \tag{4.14}$$

We prefix both encodings with $\overline{t_0}$, since the first critical operation performed by the driver is acquiring the lock.

Assumptions	Interpretation
$p_1 = p_1 t_0$	*Acquiring the lock acquires it*
$p_4 = p_4 \overline{t_0}$	*Releasing the lock releases it*
$t_2 p_5 = t_2 p_5 \overline{t_2}$	*If two variables are equal and we increment one, then they are no longer equal*
$p_2 = p_2 t_2$	*Assigning the value of one variable to another makes them equal*
$t_0 p_2 = p_2 t_0$	*commutativity condition*
$t_0 p_3 = p_3 t_0$	*commutativity condition*
$t_0 p_5 = p_5 t_0$	*commutativity condition*
$t_2 p_3 = p_3 t_2$	*commutativity condition*
$t_2 p_4 = p_4 t_2$	*commutativity condition*
$t_2 p_1 = p_1 t_2$	*commutativity condition*

Table 4.5: Assumptions used to prove the safety of a device driver

Dexter Kozen has proved the equation (4.14) using **KAT-ML** [20], an interactive theorem prover for KAT's designed for ML [1]. Here we showed the equivalence by calling function equiv^{Γ} with expressions $\overline{t_0} e$ and $\overline{t_0} e_A$ and Γ containing the assumptions in Table 4.5.

Chapter 5

Conclusion and Future Work

In this dissertation we studied and developed a decision algorithm for the equivalence of KAT expressions. We started by reimplementing a decision procedure for regular expressions which, as expected, revealed to be an adequate starting point to solve the problem we wanted. We presented some experimental results of testing the equivalence of uniformly random KAT expressions, focusing on measures such as the number of derivatives computed and the time taken by each test.

We explained how programs written in a minimal imperative language can be encoded as KAT expressions. This allowed us to address the proof of equivalence of two distinct programs, which we exemplified with a few programs.

Even though it was not in our initial plans, the idea of adapting the decision procedure to solve the Hoare logic problems - and, in general, other proofs of equivalence that can only be shown in the presence of assumptions - became possible. Thus, we extended the decision procedure to accommodate assumptions, which allowed us to prove not only the correctness but also the safety of a program.

During the study and development of the algorithms some motivations for future work arose. As we saw, when testing the equivalence of two KAT expressions the main *bottleneck* is the number of atoms computed, which directly depends on the number of different tests in the expressions. Therefore, it would be interesting to study a method that reduces the number of atoms used in the test. Alternatively, resorting this problem to an external SAT solver would also make the use of this method in formal verification more feasible. Concerning Hoare logic, it would be interesting to treat the assignment rule within a decidable first-order theory and to integrate the decision procedure in an SMT solver.

References

[1] Kamal Aboul-Hosn and Dexter Kozen. KAT-ML: An interactive theorem prover for Kleene algebra with tests. *Journal of Applied Non-Classical Logics*, 16(1–2):9–33, 2006.

[2] Marco Almeida. *Equivalence of regular languages: an algorithmic approach and complexity analysis*. PhD thesis, Faculdade de Ciências das Universidade do Porto, 2011.

[3] Marco Almeida, Nelma Moreira, and Rogério Reis. Antimirov and Mosses's rewrite system revisited. *International Journal of Foundations of Computer Science*, 20(04):669 – 684, 2009.

[4] Ricardo Almeida, Sabine Broda, and Nelma Moreira. Deciding KAT and Hoare logic with derivatives. *accepted for publication in EPTCS, GandALF 2012 (Third International Symposium on Games, Automata, Logics and Formal Verification)*, 2012.

[5] Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoret. Comput. Sci.*, 155(2):291–319, 1996.

[6] Valentin M. Antimirov and Peter D. Mosses. Rewriting extended regular expressions. In G. Rozenberg and A. Salomaa, editors, *Developments in Language Theory*, pages 195 – 209. World Scientific, 1994.

[7] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[8] Thomas Braibant and Damien Pous. An efficient Coq tactic for deciding Kleene algebras. In *Proc. 1st ITP*, volume 6172 of *LNCS*, pages 163–178. Springer, 2010.

[9] Ernie Cohen, Dexter Kozen, and Frederick Smith. The complexity of Kleene algebra with tests. Technical Report TR96-1598, Computer Science Department, Cornell University, July 1996.

[10] Project FAdo. FAdo: tools for formal languages manipulation. `http://fado.dcc.fc.up.pt`, Access date:1.1.2012.

[11] Maria João Frade and Jorge Sousa Pinto. Verification conditions for source-level imperative programs. *Computer Science Review*, 5(3):252–277, 2011.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12(10):576–580, 1969.

[13] Peter Hófner and Georg Struth. Automated reasoning in Kleene algebra. In F. Pfenning, editor, *CADE 2007*, number 4603 in LNAI, pages 279–294. Springer-Verlag, 2007.

[14] J.E. Hopcroft and R.M. Karp. *A linear algorithm for testing equivalence of finite automata.* Technical report (Cornell University. Dept. of Computer Science). Defense Technical Information Center, 1971.

[15] John E. Hopcroft. An *nlogn* algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[16] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.

[17] Dexter Kozen. Kleene algebra with tests. *Trans. on Prog. Lang. and Systems*, 19(3):427–443, May 1997.

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

[19] Dexter Kozen. Automata on guarded strings and applications. *Matématica Contemporânea*, 24:117–139, 2003.

[20] Dexter Kozen. Kleene algebras with tests and the static analysis of programs. Technical Report TR2003-1915, Computer Science Department, Cornell University, November 2003.

[21] Dexter Kozen. On the coalgebraic theory of Kleene algebra with tests. Computing and Information Science Technical Reports `http://hdl.handle.net/1813/10173`, Cornell University, May 2008.

[22] Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In D. van Dalen and M. Bezem, editors, *Proc. 10th Int. Workshop Computer Science Logic (CSL'96)*, volume 1258 of *Lecture Notes in Computer Science*, pages 244–259, Utrecht, The Netherlands, September 1996. Springer-Verlag.

[23] Dexter Kozen and Jerzy Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 789–840. North Holland, Amsterdam, 1990.

[24] Boris Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering Cybernetics*, 5:110–116, 1966.

[25] The Caml team. Ocaml. `http://caml.inria.fr/ocaml/`, Access date: 1.5.2012.