GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Master's Thesis

submitted in partial fulfillment of the
requirements for the course "Applied Computer Science"

A functional solver for word equations with regular constraints

Maximilian David Eipper

Institute of Computer Science

Bachelor's and Master's Theses
of the Center for Computational Sciences
at the Georg-August-Universität Göttingen

31st of March 2023

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
🌍 www.informatik.uni-goettingen.de

First Supervisor: Prof. Dr. Florin Manea
Second Supervisor: Dr. Henrik Brosenne

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Göttingen, 31st of March 2023

Abstract

We present a prefix-based approach for solving regex-constrained word equations combining the Nielsen transformation and Brzozowski derivatives. Differently from most approaches, we solve the word equation and the regex constraints in a combined manner. We provide a Haskell implementation of a solver using our approach. Our implementation outperforms state-of-the-art solvers on regex-constrained, quadratic equations. Since our approach is only viable within this particular niche, we conclude it is best suited as a solver tactic, as opposed to a standalone solver.

Contents

1 Introduction **2**

2 Foundations **4**

2.1 Basics . . . 4
2.2 Word Equations . . . 5
2.3 Constrained Variables . . . 5
2.4 Methods for Solving Word Equations . . . 6
2.5 Nielsen Transformation . . . 7
2.6 Regular Expressions . . . 10
2.6.1 Brzozowski Derivatives . . . 10
2.6.2 Antimirov Derivatives . . . 11
2.6.3 Transition Regexes . . . 12

3 Approach **13**

3.1 Algorithm . . . 13
3.1.1 Nullability . . . 14
3.1.2 Satisfiability . . . 14
3.1.3 a as a prefix of x . . . 15
3.1.4 y as a prefix of x . . . 15
3.1.5 Modified Algorithm . . . 16
3.2 Correctness . . . 17
3.3 Complexity . . . 19
3.4 Extensibility . . . 20

4 Implementation **21**

4.1 Haskell . . . 21
4.2 String Solvers . . . 22
4.3 SMT-LIB . . . 22
4.4 Architecture . . . 22

4.4.1 User Definable Regex Theory . . . 23
4.4.2 Algorithmic Core . . . 24
4.4.3 SMT-LIB Interface . . . 25
4.4.4 Solver Frontend . . . 26

5 Benchmarks **27**
5.1 Building the Benchmark Sets . . . 27
5.1.1 SMTQuery . . . 27
5.1.2 benchmarkset-1 . . . 28
5.1.3 benchmarkset-2 . . . 29
5.1.4 benchmarkset-2 \ benchmarkset-1 . . . 30
5.2 Other String Solvers . . . 31
5.2.1 CVC4 . . . 31
5.2.2 CVC5 . . . 31
5.2.3 Z3 . . . 31
5.2.4 Woorpje . . . 32
5.2.5 Noodler . . . 32
5.3 ZaligVinder . . . 33

6 Results **34**
6.1 Benchmarks . . . 34
6.1.1 benchmarkset-1 . . . 34
6.1.2 benchmarkset-2 . . . 36
6.1.3 benchmarkset-2 \ benchmarkset-1 . . . 37
6.2 Interpretation . . . 38

7 Conclusion **40**

Bibliography **43**

A Benchmarks **44**
A.1 benchmarkset-1 . . . 44
A.2 benchmarkset-2 . . . 46
A.3 benchmarkset-2 \ benchmarkset-1 . . . 48

Chapter 1

Introduction

The field of string solving deals with finding solutions for string-related problems, where *strings* are sequences of symbols. These problems can take many forms, but they often involve finding patterns, checking some properties of the strings or transforming the strings in some way.

This includes but is not restricted to:

- sorting strings by an ordering relation, such as lexicographic order
- finding out if a string contains a specific word
- finding out if a string follows a specific pattern
- compressing/decompressing a string to store it efficiently
- checking the syntax of a string to ensure that it follows certain rules, such as those of a programming language or a formal language
- solving equations where the variables take string values

String solving is an active area of research with many open problems still to be explored. It has a wide range of applications in computer science as well as in related fields.

In this work, we will focus on the last problem: Solving equations of strings. This problem does not deal with numeric equations but instead with so-called *word equations* in which the variables take symbolic values from the monoid Σ^*. Consider the equation $abxb = ybx$ where a and b are terminal symbols from Σ and x and y are variables that can take any string value. This equation is satisfied for $x = b, y = ab$. Let us consider another equation: $xa = byx$. It does not have a solution because no matter which values we choose for x and y the two sides are never equal.

Usually, we add additional rules – so-called *constraints* – to the variables. One such constraint could be that x must be at most 10 characters long or y must satisfy the regular expression $ab*$.

Word equations are used to model problems in cyber security and formal verification. In the former, problems can be modeled in a way so that the solutions to the equations correspond with attack vectors that can be used to breach security.

This work will deal with one specific subproblem of solving word equations. We will solve word equations where the variables have regular expression constraints. For this, we take Nielsen's algorithm [1] for solving word equations without constraints and extend it with Brzozowski's notion of regex derivatives [2] to solve equations with regex constraints. To test the viability of our approach, we provide a Haskell implementation of a string solver that follows this method. We benchmark against the state-of-the-art solvers CVC [3,4] and Z3 [5] to find out under which circumstances our approach can compete with them.

Usually, string solvers solve word equations and regex constraints separately. We are aware of only one approach – Noodler [6] – that solves equations and constraints in a joined manner by using an automata structure. Our contribution is to provide a second approach that solves equations and constraints together by using the Nielsen transformation and Brzozowski derivatives. The goal of this work is to present an algorithm, provide an implementation and evaluate its performance.

This thesis is structured as follows. In chapter 2, we introduce some basic notation and give a rough overview of string solving and related strategies. Subsequently, we have a detailed look at the Nielsen transformation and Brzozowski's algorithm for regex matching. In chapter 3, we present our modification to the Nielsen transformation to extend it to regex-constrained variables and prove its correctness. In chapter 4, we give a technical report on our Haskell implementation of the algorithm presented in chapter 3. In chapter 5, we explain the setup of our benchmarks. First, we present three benchmark sets. Then, we introduce the other string solvers we benchmark against. Finally, we explain how the experiment is going to be conducted. In chapter 6, we document and interpret the results of our benchmarks. Finally, in chapter 7, we summarize our work, draw a conclusion from our findings in chapter 6 and give an outlook on future work.

Chapter 2

Foundations

This chapter will introduce all theoretical foundations for our approach. We first introduce some notation. Then, we give an overview of the field of word equation solving, going into detail on the Nielsen transformation. Finally, we introduce some regex theories, focusing mainly on Brzozowski's work.

2.1 Basics

Words are sequences over a fixed length alphabet Σ. We write $w = w_1w_2...w_n, w_i \in \Sigma$. We denote the set of all words over Σ as Σ^*, which forms a monoid under concatenation and the empty word ε. We define the length of a word $|w_1w_2...w_n| = n$. While we use Σ to denote the alphabet of terminal symbols, we use X to denote the set of variables. A word equation is an expression of the form $\alpha = \beta$ where $\alpha, \beta \in (\Sigma \cup X)^*$. If not specified differently, we use $a, b \in \Sigma, a \neq b$ and $x, y \in X, x \neq y$.

We define the replacement function $\varphi_{\cdot\to\cdot}(\cdot)$ so that $\varphi_{x\to\alpha}(\beta)$ denotes the result of replacing every occurrence of x with α in β. E.g. $\varphi_{x\to ay}(xbbxy) = aybbayy$.

$$\begin{aligned}
\varphi_{x\to\alpha}(x) &= \alpha \\
\varphi_{x\to\alpha}(y) &= y \\
\varphi_{x\to\alpha}(a) &= a \\
\varphi_{x\to\alpha}(\beta_1\beta_2...\beta_n) &= \varphi_{x\to\alpha}(\beta_1)\varphi_{x\to\alpha}(\beta_2)...\varphi_{x\to\alpha}(\beta_n)
\end{aligned}$$

Additionally, we introduce the prefix deletion function φ_{DEL} with

$$\varphi_{DEL}(a\alpha) = \alpha$$

Finally, for readability, we define the reverse order function composition $\circ'$ with

$$f \circ' g = g \circ f$$

2.2 Word Equations

A word equation is an expression of the form $\alpha_1 x_1 \alpha_2 x_2 ... x_{n-1} \alpha_n = \beta_1 y_1 \beta_2 y_2 ... y_{m-1} \beta_m$ where $\alpha_i, \beta_i \in \Sigma^*, x_i, y_i \in X$. Now, the goal is to find out if there is an assignment function $[\![\cdot]\!] : X \to \Sigma^*$ s.t. $\alpha_1 [\![x_1]\!] \alpha_2 [\![x_2]\!] ... [\![x_{n-1}]\!] \alpha_n = \beta_1 [\![y_1]\!] \beta_2 [\![y_2]\!] ... [\![y_{m-1}]\!] \beta_m \in \Sigma^*$. In other words, we are interested in finding a function that maps the variables to words in a way that makes both sides of the equation equal. Sometimes we might just be interested in finding out whether such a mapping exists without constructing it.

We consider two special cases of word equations: *Quadratic* word equations and *regular* word equations. Quadratic word equations are equations in which each variable occurs at most twice. For example, $xxy = ayz$ is quadratic, while $xxy = axz$ is not. Regular word equations are an even stricter subclass of quadratic word equations: Each variable may occur at most once on each side. $xy = ax$ is regular, while $xx = ay$ is not.

2.3 Constrained Variables

Until now, variables have always been *unconstrained* in the way that they can take any values from Σ^*. This is not sufficient for many real-world applications. Modeling dates or passwords, for example, requires the variables to follow special formats, to have specific lengths or to be selected over only a subalphabet of Σ.

Instead of $x \in \Sigma^*$, we shall write $x \in \Sigma^k$ to denote variables whose values must be of length k. Similarly, we write $x \in \Sigma'^*$ for $\Sigma' \subset \Sigma$ to denote variables that only take symbols from a restricted subset of Σ. For now, the only other kind of constraint we are interested in is regular expressions.

Let the notion $x \in ab*$ stand for the constraint that x must match the regular expression $ab*$. This means x can take the values $a, ab, abb, ...$ but not b or ε. This work will focus on implementing regex constraints for variables in the Nielsen transformation. The reason for this is that regular expressions are generally expressive and that they prove powerful enough to cover most constraints presented in this chapter, albeit with drawbacks in usability or blowup in complexity. Subalphabets $\Sigma' = \{\sigma'_1, \sigma'_2, ...\sigma'_n\}$ can be modeled as one exhaustive disjunction over all terminal symbols of that subalphabet with $x \in \sigma'_1 | \sigma'_2 | ... | \sigma'_n$. This means that this simple constraint can only be modeled with linear complexity. Constant length constraints $x \in a^k$ can be modeled with $x \in \underbrace{aa...a}_{k}$. Again, we see linear blowup. As regular languages are less powerful than context-free languages, complicated counting constraints where multiple variables depend on the same length constant k cannot be modelled.

2.4 Methods for Solving Word Equations

To solve such equation problems, we can employ multiple different methods. This can involve manipulating the equation in various ways, such as substituting values for variables, rearranging terms, rewriting the equation in some way to simplify it or applying rules of algebra. Keep in mind that we are on a monoid, so basic tricks like adding an inverse do not work. Depending on the complexity of the equation and the methods used to solve it, solving a word equation can be a simple or a challenging task.

There are many different approaches for solving word equations, and the appropriate approach will depend on the specific problem and the requirements of the solution. Some common techniques for solving word equations include

- guessing the variables by brute force
- guessing the variables by educated guessing
- simplifying the equation
- isolating variables
- fixing the position
- approximating lengths
- left side elimination

Fixing the positions and *approximating lengths* are approaches that try to approximate the variable position or length by some variation of binary search. *Left side elimination* makes use of the basic fact that string equality can be recursively defined:

Definition 1. *Let $a_1a_2...a_n, b_1b_2...b_m$ be two strings $\in \Sigma^*$. The recursively defined equality $a_1a_2...a_n = b_1b_2...b_m$ holds iff the prefixes a_1 and b_1 are equal and the remaining suffixes $a_2...a_n$ and $b_2...b_m$ are equal. I.e. iff $a_1 = b_1$ and $a_2...a_n = b_2...b_m$. The base case for this recursive comparison is $\varepsilon = \varepsilon$.*

Theorem 1. *The string equality defined in Definition 1 – like other equalities – describes an equivalence relation. The following properties hold for any $\alpha, \beta, \gamma \in \Sigma^*$:*

1. *Reflexivity: $\alpha = \alpha$*
2. *Symmetry: $\alpha = \beta \Rightarrow \beta = \alpha$*
3. *Transitivity: $\alpha = \beta \wedge \beta = \gamma \Rightarrow \alpha = \gamma$*

2.5 Nielsen Transformation

One method for left side elimination is the Nielsen transformation [1]. It is defined for equations of terminals and unconstrained variables. It extends the recursive definition 1 of string equality for terminal words over Σ^* to terminal and variable words over $(\Sigma \cup X)^*$ by making use of a case analysis. Keep in mind that two strings are only equal if their prefixes are equal.

Let $a, b \in \Sigma, a \neq b$ two distinct terminals, $x, y \in X, x \neq y$ two distinct variables and $\alpha, \beta \in (\Sigma \cup X)^*$ two (possibly equal) strings. Then for the Nielsen transformation, there are the following cases:

1. Both sides start with the same terminal: $a\alpha = a\beta$.
   We can eliminate this terminal and reduce the equality problem to $\alpha = \beta$.

2. Both sides start with a different terminal: $a\alpha = b\beta$
   The strings cannot be equal, because $a \neq b$.

3. One side starts with a variable: $x\alpha = \beta$
   We try to set this variable to ε and check if the equation can be solved.

   If x is empty we have to assume that every occurrence of x is empty. We have to remove x from both sides of the equation.

   $$\begin{aligned} x\alpha &= \beta \\ \varphi_{x\to\varepsilon}(x\alpha) &= \varphi_{x\to\varepsilon}(\beta) \\ \varphi_{x\to\varepsilon}(\alpha) &= \varphi_{x\to\varepsilon}(\beta) \end{aligned}$$

   Here, we assumed without loss of generality that the side starting with a variable is the left hand side of the equation (We may assume this because the equality relation is symmetrical).

4. One side starts with a variable and one starts with a terminal: $x\alpha = a\beta$

   This is only possible if x starts with a or if x is empty. The case that x is empty is already covered in case 3. Therefore, we assume that x starts with a. If x starts with a, we must assume that a is a prefix for x for every occurrence of x. In other words, we introduce a new variable x', s.t. $x = ax'$. We then rewrite the equation:

   $$\begin{aligned} x\alpha &= a\beta \\ \varphi_{x\to ax'}(x\alpha) &= \varphi_{x\to ax'}(a\beta) \\ ax'\varphi_{x\to ax'}(\alpha) &= a\varphi_{x\to ax'}(\beta) \end{aligned}$$

   As introducing a new variable for each substitution quickly becomes tedious, we "recycle" the variables: Instead of replacing x with ax' we replace x with ax. This is especially useful, because $xa = bx$ is isomorph to $x'a = bx'$ as we will see later. In this case, our rewrite looks like this:

   $$ax\varphi_{x\to ax}(\alpha) = a\varphi_{x\to ax}(\beta)$$

5. Both sides start with the same variable: $x\alpha = x\beta$
   Both sides have the same prefix. We can just delete the prefix x. Whatever solution for x solves the equation, can be determined later. We are thus left with $\alpha = \beta$.

6. Both strings start with different variables: $x\alpha = y\beta$
   This means that one variable must be the prefix of the other variable. Without loss of generality we assume y to be the prefix of x.

$$\begin{aligned} x\alpha &= y\beta \\ \varphi_{x\to yx}(x\alpha) &= \varphi_{x\to yx}(y\beta) \\ yx\varphi_{x\to yx}(\alpha) &= y\varphi_{x\to yx}(\beta) \end{aligned}$$

An equation $\alpha = \beta$ is solvable iff $\varepsilon = \varepsilon$ can be derived from repeated application of these rules.

Keep in mind that some of the cases are not necessarily exclusive and can overlap (e.g. 3 and 4). Just because one case does not yield a solution does not mean another case will yield no solution either. For example $x = a$ matches the form $x\alpha = \beta$, but case 3 cannot be used to reduce $x = a$ to $\varepsilon = \varepsilon$, instead case 4 must be used. This must be taken into account when implementing a solver.

We can model the satisfiability problem as a graph reachability problem. Each node holds a word equation. If one word equation can be rewritten into another word equation by one of our rewrite rules, we draw a directed edge from its node to the others' node. The question whether $\alpha = \beta$ can be rewritten to $\varepsilon = \varepsilon$ is equivalent to the question whether the node holding $\varepsilon = \varepsilon$ can be reached from the node holding $\alpha = \beta$.

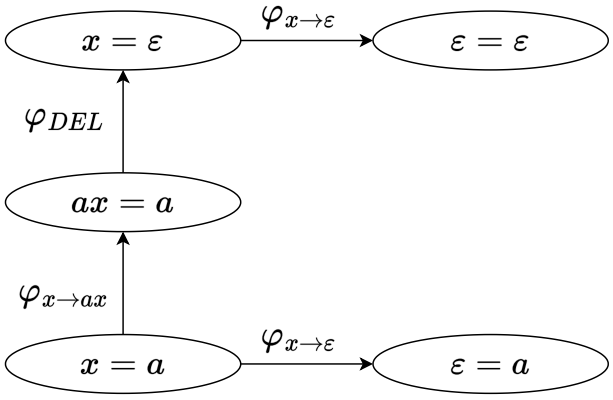

Figure 2.1: Graph for the satisfiable word equation $x = a$

Let us consider the simple and satisfiable word equation $x = a$. Figure 2.1 shows the graph of the rewrites. We start at the bottom left with $x = a$. The equation can be rewritten, either by deleting x and thus rewriting the equation by $\varphi_{x\to\varepsilon}$ (case 3), or by assuming that a must be a prefix of

x, triggering the rewrite $\varphi_{x\to ax}$ (case 4). In the latter case we arrive at the equation $ax = a$. We delete the prefix a from both sides (case 1), arriving at $x = \varepsilon$. By deleting x (case 3), we reach $\varepsilon = \varepsilon$. We reach $\varepsilon = \varepsilon$ from $x = a$ via the directed path characterized by the chain of rewrites $\varphi = \varphi_{x\to ax} \circ' \varphi_{DEL} \circ' \varphi_{x\to\varepsilon}$, because $\varphi(x) = \varepsilon = \varphi(a)$. We say $x = a$ is satisfiable with the solution φ.

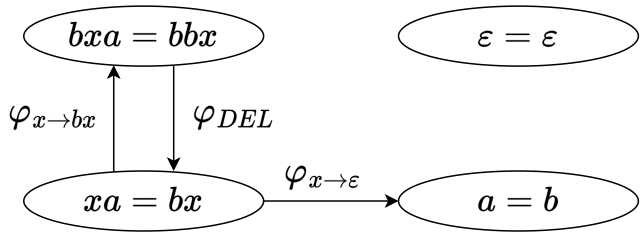

Figure 2.2: Graph for the unsatisfiable word equation $xa = bx$

Let us now consider the case of an unsatisfiable equation: $xa = bx$ (figure 2.2). We start at the bottom left and try to reach the top right node $\varepsilon = \varepsilon$. Again, we can either delete x arriving at the dead end $a = b$, or we assume that x has the prefix b. In this case we reach the equation $bxa = bbx$. We delete the shared prefix b from both sides of the equation, bringing us back to $xa = bx$. No other rewrite rule applies. We cannot reach $\varepsilon = \varepsilon$. Therefore, the equation $xa = bx$ is unsatisfiable.

We reduced the satisfiability of word equations to a graph reachability problem. We can solve the reachability problem with any graph discovery algorithm. This includes depth-first search (DFS) and breadth-first search (BFS). Interestingly, this also includes Dijkstra's [7] algorithm for finding the shortest path. It generalizes BFS with a cost function. This cost function could be a heuristic, ranking each word equation by how promising it is to be reducible to $\varepsilon = \varepsilon$. Given a good cost function, Dijkstra could solve satisfiable equations faster than BFS. Note, however, that unsatisfiable equations would still take as long as with BFS, because Dijkstra's algorithm still has to perform an exhaustive graph search, if it cannot find a path. While this is an interesting idea, in this work we will not focus on finding heuristics.

To answer the question of satisfiability, we just need to find out whether some path of rewrites from $\alpha = \beta$ to $\varepsilon = \varepsilon$ exists. If we are interested in finding a satisfying variable assignment, we can backtrack over such a path. If we are interested in finding all assignments, we backtrack over all such paths.

Day et al. [8] proved that the satisfiability problem for regular word equations is in NP and that for quadratic word equations the algorithm terminates in finite time.

This algorithm only works for unconstrained variables. In the next section we will have a look at regular expressions and how we can use them to constrain variables.

2.6 Regular Expressions

A regular expression p over the alphabet Σ describes a specific subset of Σ^*. We use $L(p)$ to describe the set of words that are described by p. [2]

We recursively define regular expressions to be either one of the base cases

1. A symbol $a \in \Sigma$. $L(a) = \{a\}$
2. The empty regex λ that only matches the empty string ε. $L(\lambda) = \{\varepsilon\}$
3. The never matching regex $\emptyset$ that matches no string. $L(\emptyset) = \{\}$. This will later prove useful to model error states.

Let p, q be regular expressions over Σ. We now add the following recursive definitions to also be regular expressions

4. The concatenation pq. This matches all words ww' where $w \in L(p), w' \in L(q)$
5. The iteration $p*$. This regex matches zero or more occurrences of p. $L(p*) = L(\lambda) \cup L(p) \cup L(pp) \cup ...$
6. The negation p'. This matches precicely when p does not match. $L(p') = \Sigma^* \setminus L(p)$
7. The conjunction $p \,\&\, q$. $L(p \,\&\, q) = L(p) \cap L(q)$
8. The disjunction $p \mid q$. $L(p \mid q) = L(p) \cup L(q)$

We denote the set of all regular expressions over Σ as $R(\Sigma)$.

2.6.1 Brzozowski Derivatives

In his 1964 work, Brzozowski [2] introduced a methodology for regular expression matching that does not construct an intermediary automaton. For this he defined a notion of *derivatives* of a regular expression. The derivative of a regular expression p respective to the symbol a is the remainder of p after successfully matching the first character a. Let us consider $a \in \Sigma, w \in \Sigma^*, p \in R(\Sigma)$. We define the derivative $D_a p$ in a way s.t. $aw \in L(p) \Leftrightarrow w \in L(D_a p)$.

Take for example the string ab and the regex $a(b|c)$. To decide whether $ab \in a(b|c)$, we take the prefix of ab, i.e. a and then calculate the derivative $D_a(a(b|c)) = b|c$. We reduced the matching problem $ab \in a(b|c)$ to $b \in b|c$. Now, we take the derivative $D_b(b|c) = \lambda|\emptyset$. We now reduced our problem to $\varepsilon \in \lambda|\emptyset$. Note that $\varepsilon \in L(\lambda) \cup L(\emptyset)$. Therefore, $ab \in a(b|c)$.

We always solve the matching problem by deriving character-by-character and then deciding whether the remaining regular expression matches ε: Let $w = w_1 w_2 ... w_n \in \Sigma^*$, then $w \in p \Leftrightarrow \varepsilon \in D_{w_n}(...(D_{w_2}(D_{w_1} p)))$. Thus, we only need a method of determining whether a regex p is nullable,

i.e. whether $\varepsilon \in p$. For this we define the nullability function $\nu : R(\Sigma) \to \{false, true\}$ with $\nu(p) \Leftrightarrow \varepsilon \in p$.

$$\begin{array}{rcl}
\nu(\lambda) & = & true \\
\nu(\emptyset) & = & false \\
\nu(a) & = & false \\
\nu(pq) & = & \nu(p) \wedge \nu(q) \\
\nu(p*) & = & true \\
\nu(p') & = & \neg\nu(p) \\
\nu(p \,\&\, q) & = & \nu(p) \wedge \nu(q) \\
\nu(p \mid q) & = & \nu(p) \vee \nu(q)
\end{array}$$

We can now define $D_a p$ as

$$\begin{array}{rcll}
D_a a & = & \lambda & \\
D_a b & = & \emptyset & \\
D_a \lambda & = & \emptyset & \\
D_a \emptyset & = & \emptyset & \\
D_a(pq) & = & \begin{cases} (D_a p)q \mid D_a q & \text{if } \nu(p) \\ (D_a p)q & \text{if } \neg\nu(p) \end{cases} & \\
D_a(p*) & = & (D_a p)p* & \\
D_a(p') & = & (D_a p)' & \\
D_a(p \,\&\, q) & = & D_a p \,\&\, D_a q & \\
D_a(p \mid q) & = & D_a p \mid D_a q &
\end{array}$$

Brzozowski also presented an algorithm to construct a DFA from a regular expression. Brzozowski's notion of derivatives inspired an entire field of research and regex theories. In the following sections we will touch on two of these.

2.6.2 Antimirov Derivatives

In his 1995 work, Antimirov [9] introduced the notion of *partial derivatives*. Just like Brzozowski's derivatives can be used to construct a DFA, Antimirov's partial derivatives can be used to construct an NFA. Antimirov does this by defining the partial derivative of p respective to a as $\partial_a : R(\Sigma) \to 2^{R(\Sigma)}$ as opposed to Brzozowski's $D_a : R(\Sigma) \to R(\Sigma)$. Antimirov also presents an algorithm for constructing a DFA that is more compact than Brzozowki's.

2.6.3 Transition Regexes

Stanford et al. [10] present the notion of *symbolic derivatives*. Instead of operating on a regex, they return functions that later evaluate to regexes. By doing this, they delay branching and decision making for as long as possible. In their benchmarks they outperform Brzozowski derivatives and perform especially well on regular expressions with logical branches ($\&$ and $|$).

Both of these approaches are complete regex theories and can be used for regex matching. In this work, to keep it simple, we will only focus on the classic Brzozowski derivatives. In the next chapter, we will describe how to incorporate them into the Nielsen transformation.

Chapter 3

Approach

The previous chapter introduced the Nielsen transformation, a prefix-based method to solve the satisfiability problem for unconstrained word equations, and Brzozowski's regex derivatives, a prefix-based method for regex matching. In this chapter, we present our approach of bringing both methods together, effectively extending the Nielsen transformation with regex derivatives to solve the satisfiability problem for regex-constrained word equations.

We now restrict the variables $x \in X$ by adding constraints $c(x) \in R(\Sigma)$. An assignment $[\![\cdot]\!]$ must satisfy these constraints. In other words, $[\![x]\!] \in c(x)$ for all x. This approach can handle unconstrained variables as well: For any unconstrained variable x, set $c(x)$ to be the catch-all regex $\emptyset'$ with $L(\emptyset') = \Sigma^* \setminus L(\emptyset) = \Sigma^*$.

Variables are now selected over (usually strict) subsets of Σ^*. This has severe consequences, as questions like "can x be empty?", "is there a solution for x?" or "can y be a prefix of x?" suddenly become non-trivial.

3.1 Algorithm

We modify Nielsen's algorithm to also cover regex-constrained variables. While some rewrite rules stay the same (i.e. if both sides start with a terminal symbol), a few rules change when moving from unconstrained to regex-constrained variables. This is because unconstrained variables are always *satisfiable* (i.e. a solution exists) and *nullable* (i.e. can be equal to ε), while for constrained variables we now have to add checks. In the sections 3.1.1 - 3.1.4, we therefore take into account the ways in which constrained variables differ from unconstrained variables. In section 3.1.5, we bring them together and present our modified version of the Nielsen transformation.

3.1.1 Nullability

While unconstrained variables, selected from Σ^*, are always *nullable*, i.e. they can be empty, this does not hold for regex-constrained variables. Consider $x, c(x) = a \mid b$. Then x is not nullable, because $\varepsilon \notin c(x)$. Thus, no assignment $[\![\cdot]\!]$ exists with $[\![x]\!] = \varepsilon$. We need to take this into account because case 3 of the Nielsen transformation deletes variables without checking for nullability first.

3.1.2 Satisfiability

Unconstrained variables always have some solution within Σ^*. Constrained variables on the other hand can have *unsatisfiable* constraints. Take for example $x, c(x) = a\&b$. Then, there exists no assignment $[\![\cdot]\!]$ with $[\![x]\!] \in c(x)$, because $L(c(x)) = L(a) \cap L(b) = \{a\} \cap \{b\} = \{\}$. This breaks the current setup of the Nielsen transformation: Until now we always assumed that whenever we encounter a variable, there exists some value it can take. We need to take into account, that whenever we work with a variable, we have to check it for satisfiability first. This applies to the cases 3 - 6.

We introduce the satisfiability function $\sigma : R(\Sigma) \to \{false, true\}$ with $\sigma(p) \Leftrightarrow L(p) \neq \{\}$.

$$
\begin{array}{rcc}
\sigma(\emptyset) & = & false \\
\sigma(\lambda) & = & true \\
\sigma(a) & = & true \\
\sigma(pq) & = & \sigma(p) \wedge \sigma(q) \\
\sigma(p*) & = & true \\
\sigma(p') & = & unknown \\
\sigma(p \,\&\, q) & = & \sigma(p) \wedge \sigma(q) \\
\sigma(p \mid q) & = & \sigma(p) \vee \sigma(q)
\end{array}
$$

For $\sigma(p')$ we cannot determine the satisfiability. Consider the satisfiable regex $a|b$. Its negation $(a|b)'$ is satisfiable, because $aa \in (a|b)'$. Now consider the unsatisfiable regex $a\&b$. Again, its negation $(a\&b)'$ is satisfiable, because $a \in (a\&b)'$. So for these two regular expressions – one satisfiable, the other unsatisfiable – both negations are satisfiable. Still, let us consider $(a|b)*$ over the alphabet $\Sigma = \{a, b\}$. Then $L((a|b)*) = \Sigma^*$ and subsequently $L(((a|b)*)') = \{\}$, so the negation is unsatisfiable. The satisfiability of the negation p' can therefore not be inferred from the satisfiability of p. To be safe, we assume that p' is always satisfiable: $\sigma(p') = true$. Note, that this does not lead to incorrect results: A variable wrongly declared as satisfiable can only be deleted from the word equation later on, if it matches ε and is therefore satisfiable.

3.1.3 a as a prefix of x

If we now encounter case 4 of the Nielsen transformation, i.e. $x\alpha = a\beta$, we assume that x has the prefix a. For unconstrained variables this is always possible. For constrained variables we first need to find out if x is allowed to start with a. In other words, we need to check that $\sigma(D_a(c(x))$. When rewriting with $\varphi_{x \to ax'}$, we also need to introduce the new constraint for $c(x') = D_a(c(x))$. If we recycle variable names, we update the constraint $c(x)$ instead.

3.1.4 y as a prefix of x

For unconstrained variables, things are simpler: y can always be a prefix of x. For regex-constrained variables, it is not trivially possible to check whether one can be another variable's prefix because we have no simple way of finding out if $c(x)$ can begin with some $w \in L(c(y))$. We work around this issue by reformulating it. y can be a prefix of x if it is either empty (i.e. $y = \varepsilon$), or if $x = ax', y = ay'$ and y' is a prefix of x' for some $a \in \Sigma$. The case $y = \varepsilon$ can be safely disregarded because it is already covered in case 3 of the Nielsen transformation. The other case just fixes a shared prefix a for both variables and postpones the rest of the prefix problem for a later step of the Nielsen transformation. For this to work, both x and y need to be allowed to start with a, in other words $\sigma(D_a(c(x))) \wedge \sigma(D_a(c(y)))$ needs to hold.

As we are interested in finding all $a \in \Sigma$ that fulfill this constraint, we define the prefix-function $\pi : R(\Sigma) \to 2^\Sigma, \pi(p) = \{a \in \Sigma \mid \sigma(D_a p)\}$ that can be algorithmically computed as follows:

$$
\begin{array}{rcl}
\pi(\lambda) & = & \{\} \\
\pi(\emptyset) & = & \{\} \\
\pi(a) & = & \{a\} \\
\pi(pq) & = & \begin{cases} \pi(p) \cup \pi(q) & \text{if } \nu(p) \\ \pi(p) & \text{if } \neg\nu(p) \end{cases} \\
\pi(p*) & = & \pi(p) \\
\pi(p') & = & unknown \\
\pi(p \,\&\, q) & = & \pi(p) \cap \pi(q) \\
\pi(p \mid q) & = & \pi(p) \cup \pi(q)
\end{array}
$$

For $\pi(p')$ we cannot determine the possible prefixes, so instead we say any symbol is possible and define $\pi(p') = \Sigma$.

For all $a \in \pi(c(x)) \cap \pi(c(y))$ (and those can be linearly many), we replace y with ay' by use of $\varphi_{y \to ay'}$ and introduce the new constraint $c(y') = D_a(c(y))$. It suffices to only cover $y = ay'$ and leave out $x = ax'$, because the resulting word equation $x\varphi_{y \to ay'}(\alpha) = ay'\varphi_{y \to ay'}(\beta)$ starts with x and a and thus triggers case 4 immediately.

Note, that strictly speaking π is not needed. Instead of iterating over $a \in \pi(c(x)) \cap \pi(c(y)) \subset \Sigma$,

we could iterate over all $a \in \Sigma$ because unsatisfiable derivatives get filtered out later anyways. For large alphabets Σ this comes with a blowup linear in $|\Sigma|$ for every application of case 6. This accumulates to an exponential blowup for repeated applications of case 6. We try to minimize this blowup by using π. This blowup does not exist in the unconstrained version of the Nielsen transformation. It is only introduced by our method of fixing shared prefixes a.

3.1.5 Modified Algorithm

To accommodate regex-constrained variables, we must modify the case analysis of the Nielsen transformation. We integrate the ideas from the previous chapters into the four cases 3 - 6 that deal with variable changes. The two non-terminal cases 1 and 2 stay the same. The updated case analysis looks like this:

Let $a, b \in \Sigma, a \neq b$ be two distinct terminals, $x, y \in X, x \neq y$ two distinct variables with constraints $c(x), c(y) \in R(\Sigma)$ and $\alpha, \beta \in (\Sigma \cup X)^*$ two (possibly equal) strings. Then for the Nielsen transformation, there are the following cases:

1. Both sides start with the same terminal: $a\alpha = a\beta$.
   We can eliminate this terminal and reduce the equality problem to $\alpha = \beta$.

2. Both sides start with a different terminal: $a\alpha = b\beta$
   The strings cannot be equal because $a \neq b$.

3. One side starts with a variable: $x\alpha = \beta$
   If x is satisfiable and nullable, i.e. if $\sigma(c(x)) \wedge \nu(c(x))$, we delete x from the equation using $\varphi_{x \to \varepsilon}$.

$$\begin{aligned} x\alpha &= \beta \\ \varphi_{x\to\varepsilon}(x\alpha) &= \varphi_{x\to\varepsilon}(\beta) \\ \varphi_{x\to\varepsilon}(\alpha) &= \varphi_{x\to\varepsilon}(\beta) \end{aligned}$$

4. One side starts with a variable, and one starts with a terminal: $x\alpha = a\beta$
   If x is satisfiable and if it can start with a, i.e. $\sigma(c(x)) \wedge \sigma(D_a(c(x)))$, we rewrite the equation using $\varphi_{x \to ax'}$ and we introduce the updated constraint $c(x') = D_a(c(x))$.

$$\begin{aligned} x\alpha &= a\beta \\ \varphi_{x\to ax'}(x\alpha) &= \varphi_{x\to ax'}(a\beta) \\ ax'\varphi_{x\to ax'}(\alpha) &= a\varphi_{x\to ax'}(\beta) \end{aligned}$$

5. Both sides start with the same variable: $x\alpha = x\beta$
   If x is satisfiable, i.e. if $\sigma(c(x))$, we delete the prefix x from both sides of the equation leaving us with $\alpha = \beta$.

6. Both strings start with different variables: $x\alpha = y\beta$
   We fix a prefix for y. For all $a \in \pi(c(x)) \cap \pi(c(y))$, we rewrite using $\varphi_{y \to ay'}$.

$$\begin{aligned} x\alpha &= y\beta \\ \varphi_{y \to ay'}(x\alpha) &= \varphi_{y \to ay'}(y\beta) \\ x\varphi_{y \to ay'}(\alpha) &= ay'\varphi_{y \to ay'}(\beta) \end{aligned}$$

The rest of the algorithm stays unchanged: We try to find a directed path from $\alpha = \beta$ to $\varepsilon = \varepsilon$.

3.2 Correctness

Theorem 2. *The algorithm formulated in 3.1 is correct. A solution for $\alpha = \beta$ exists, iff there is a path from $\alpha = \beta$ to $\varepsilon = \varepsilon$ in the graph of rewrites.*

Proof. We will prove both sides of the equivalence separately.

Let $\varphi = \varphi_1 \circ' \varphi_2 \circ' ...\varphi_n$ be the path of rewrites so that $\varphi(\alpha) = \varepsilon = \varphi(\beta)$.

Now, ignore all prefix deletions φ_{DEL} and group all rewrites by variable. For each variable x we get $\varphi_x = \varphi_{x \to ax'} \circ' \varphi_{x' \to a'x''} \circ' ... \circ' \varphi_{x''' \to \varepsilon} = \varphi_{x \to aa'a''...}$. Note, that each φ_x must end with $\varphi_{x''' \to \varepsilon}$, because otherwise $\varphi(\alpha) \neq \varepsilon$ or $\varphi(\beta) \neq \varepsilon$. We define the assignment function $[\![x]\!] = \varphi_x(x) = aa'a''...$ and the overall rewrite without prefix deletion $\Phi = \circ'_{x \in X} \varphi_x$ so that $\Phi(\alpha) = \Phi(\beta) \in \Sigma^*$. Then $\alpha = \beta$ holds with the assignment function $[\![\cdot]\!]$. Remember that rewrites of the form $\varphi_{x \to ax'}$ only exist if $\sigma(D_a(c(x)))$, and $\varphi_{x \to \varepsilon}$ only exists if $\nu(c(x))$, because otherwise our algorithm does not allow this rewrite. Therefore, $\nu(D_{a'''}(...(D_{a'}(D_a(c(x))))) \Leftrightarrow \varepsilon \in D_{a'''}(...(D_{a'}(D_a(c(x)))) \Leftrightarrow aa'...a''' \in c(x) \Leftrightarrow [\![x]\!] \in c(x)$. It follows that $\alpha = \beta$ has a solution $[\![\cdot]\!]$ that satisfies the regex constraints.

Let us now consider the other direction. Let $[\![\cdot]\!]$ be the assignment function of a solution for $\alpha = \beta$ that satisfies the regex constraints, i.e. $[\![x]\!] \in c(x)$ for all variables x. We will now prove there exists a path $\varphi = \varphi_1 \circ' \varphi_2 \circ' ...\varphi_n$ so that $\varphi(\alpha) = \varepsilon = \varphi(\beta)$.

We make a case distinction:

1. $a\alpha = a\beta$: Both sides start with the same terminal symbol a.
   We rewrite the equation to $\alpha = \beta$ with φ_{DEL} and repeat this case distinction.

2. $x\alpha = \beta$: One side starts with the variable x and $[\![x]\!] = \varepsilon$. Without loss of generality, we choose the left side.
   Note, that $[\![x]\!] = \varepsilon$ implies $\nu(c(x))$. We rewrite with $\varphi_{x \to \varepsilon}$ and repeat this case distinction.

3. $x\alpha = a\beta$: One side starts with the variable x, the other side starts with the terminal symbol a, the assignment $[\![x]\!]$ starts with a. Without loss of generality we chose the left side.
   We rewrite with $\varphi_{x \to ax'} \circ' \varphi_{DEL}$. Note, that here we have two rewrites because $\varphi_{x \to ax'}$ results in both sides of the equation starting with a. Therefore, we immediately follow up with φ_{DEL}. Afterwards, we repeat this case distinction.

4. $x\alpha = y\beta$: Both sides start with a variable (x and y may be the same), $[\![x]\!]$ starts with a, $[\![y]\!]$ starts with a. Note, that this implies $\sigma(D_a(c(x)))$ and $\sigma(D_a(c(y))$. We rewrite with $\varphi_{x\to ax'} \circ' \varphi_{y\to ay'} \circ' \varphi_{DEL}$ and repeat this case distinction.

5. $\varepsilon = \varepsilon$: The word equation has been fully reduced to $\varepsilon = \varepsilon$. We are done.

This covers all possible cases. Cases like $a\alpha = b\beta$ for $a \neq b$ or $x\alpha = a\beta$ where $[\![x]\!]$ does not start with a need not be considered, because they are unsatisfiable and we assumed our word equation to be satisfiable. Furthermore, a rewrite done by the cases 1 - 4 always results in another equation that is still satisfiable.

To prove that we ultimately arrive at $\varepsilon = \varepsilon$, we must now only prove that each case moves us closer to $\varepsilon = \varepsilon$. We do this by showing that each case reduces the *complexity* of the word equation. We define the measure of complexity $||\cdot||$ to be $||\varepsilon|| = 0, ||a|| = 1, ||x|| = |[\![x]\!]|+1, ||\alpha_1\alpha_2|| = ||\alpha_1||+||\alpha_2||$. We chose this particular definition of complexity solely so that every rewrite reduces the equation's complexity strictly monotonically. Also, this definition ensures that $||\alpha|| = 0 \Leftrightarrow \alpha = \varepsilon$. Let us use the short notation $||\varphi||$ for the change in complexity φ achieves: $||\varphi|| = ||\alpha\beta|| - ||\varphi(\alpha)\varphi(\beta)||$. Note, that $||\varphi_1 \circ' \varphi_2|| = ||\varphi_1|| + ||\varphi_2||$.

Case 1 reduces the complexity of the word equation, because $||\varphi_{DEL}|| = -2$. Case 2 also reduces the complexity of the word equation, because $||\varphi_{x\to\varepsilon}|| \in \{-1, -2\}$ depending on the number of occurrences of x. Let us now consider case 3. $||\varphi_{x\to ax'}|| = 0$, because $|[\![x']\!]| = |[\![x]\!]| - 1$. Therefore, $||\varphi_{x\to ax'} \circ' \varphi_{DEL}|| = -2$. Let us now consider the final case 4. Again, $||\varphi_{x\to ax'} \circ' \varphi_{y\to ay'} \circ' \varphi_{DEL}|| = -2$. The only terminating case we can possibly choose is case 5. In other words, we only terminate for equations with complexity 0. This entails that for a solution φ with $\varphi(\alpha) = \varepsilon = \varphi(\beta)$ it always holds that $||\alpha\beta|| + ||\varphi|| = 0$. Keep in mind that the cases 1 - 4 reduce the complexity by 1 or 2. Therefore, every word equation $\alpha = \beta$ terminates after k iterations where $\frac{||\alpha\beta||}{2} \leq k \leq ||\alpha\beta||$.

□

3.3 Complexity

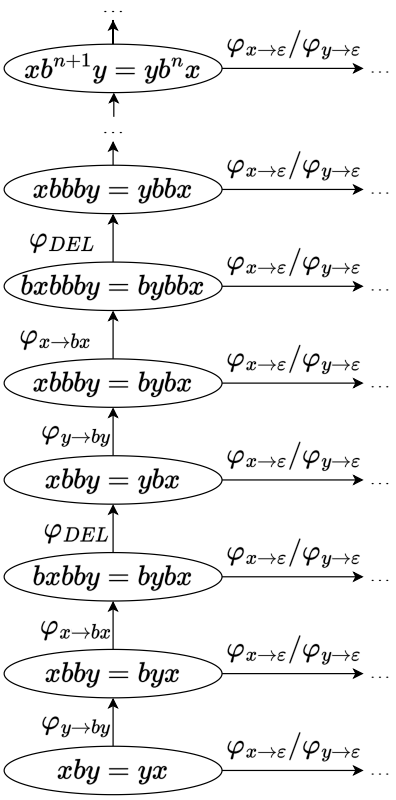

Figure 3.1: Graph for the non-terminating word equation $xby = yx, c(x) = (a|b)*, c(y) = b*$

The algorithm does not terminate on all unsatisfiable word equations: Consider the equation $xby = yx, c(x) = (a|b)*, c(y) = (b*)a$. We quickly see that rewriting it with $\varphi_{x\to\varepsilon}$ or $\varphi_{y\to\varepsilon}$ does not bring us towards $\varepsilon = \varepsilon$, because the resulting equations $xb = x$ and $by = y$ are dead ends. Figure 3.3 shows the graph of rewrites. We start in the bottom left with $xby = yx$. We can either delete x or y, by rewriting with $\varphi_{x\to\varepsilon}$ or $\varphi_{y\to\varepsilon}$, or we fix the shared prefix b. If we fix the shared prefix b, we trigger the chain of rewrites $\varphi_{y\to by} \circ' \varphi_{x\to bx} \circ' \varphi_{DEL}$. This leaves us with the new word equation $xbby = ybx$. The resulting equation again starts with x on the left and y on the right side. We can therefore again fix the prefix b. We see that repeating the rewrite chain $\varphi_{y\to by} \circ' \varphi_{x\to bx} \circ' \varphi_{DEL}$ for n times gives us the equation $xb^{n+1}y = yb^nx$. We do not reach $\varepsilon = \varepsilon$ and are caught exploring an infinite graph. The algorithm does not terminate.

As we will see in the benchmarks later on, this is an extreme case and a very rare occurrence. Still, we are unable to guarantee that our algorithm terminates on unsatisfiable word equations.

For satisfiable word equations with a solution $[\![\cdot]\!]$, the BFS can be bounded to terminate after $O(||\alpha\beta||)$ iterations. Each node can have $O(|\Sigma|)$ directed neighbors ($O(1)$ for the cases 1 - 5 and $O(|\Sigma|)$ for case 6). This accumulates to $O(|\Sigma|^{||\alpha\beta||})$ many nodes visited before reaching $\varepsilon = \varepsilon$.

Note, that this does not mean that the satisfiability problem for regex-constrained word equations is undecidable, only that this particular algorithm cannot decide it. We could work around this by making use of regular expression *quotients*. Let $p, q \in R(\Sigma)$. We define the quotient p/q to be the regular expression describing all words that match p and have a prefix that matches q, without that prefix. In other words $L(p/q) = \{w_2 \mid \exists\, w_1, w_2 \in \Sigma^*, w_1 \in L(q), w_1w_2 \in L(p)\}$. We observe that $D_a p = p/a$. If we can calculate the quotient p/q, we can solve the case 6 without having to fix

a prefix a. Instead, when assuming y to be a prefix of x, we rewrite with $\varphi_{x \to yx'}$ and introduce the new constraint $c(x') = c(x)/c(y)$.

3.4 Extensibility

The algorithm described in section 3.1 is agnostic to the regex theory at hand. It does not matter whether we use Brzozowski, Antimirov or transition regexes. The regex theory must only be *derivative-based*, meaning it defines D, ν, σ, π.

Simple length constraints of the form "x must be of length k" can also be expressed as derivative-based constraints. Let $c(x) = k \in \mathbb{Z}$ stand for the constraint that $|[\![x]\!]| = k$. Then the definitions

$$\begin{array}{rcl} D_a k & = & k - 1 \\ \nu(k) & = & k = 0 \\ \sigma(k) & = & k \geq 0 \\ \pi(k) & = & \Sigma \end{array}$$

allow us to integrate length constraints into the algorithm. In this work, although, we will only work with Brzozowski regex constraints.

Chapter 4

Implementation

We implement a string solver that uses the algorithm specified in chapter 3.1 in the programming language Haskell[1]. This chapter will introduce and document this solver. Readers who are not interested in technical-practical details may skip this chapter entirely.

4.1 Haskell

Haskell [11] is a purely functional, lazily-evaluated programming language with very expressive, declarative semantics. Purely functional means that Haskell data structures are immutable and that the language models side effects and IO as expressions. This makes optimization, verification and parallelization of Haskell programs fairly easy. Declarative means that instead of describing a process, we describe just a desired result and leave it to the runtime to evaluate our program. This makes it especially easy to write mathematical programs, as formulae can often be translated to Haskell in a very straightforward manner. Furthermore, Haskell also supports algebraic data types. The data type `Bool` can be defined in the following way:

```
data Bool = False | True
```

We introduce a new data type `Bool` whose values can either have the form `False` or `True`. For a more complex data type let us consider the data type `Either a b`. Its values encapsulate either a value of type `a` or a value of type `b`.

```
data Either a b = Left a | Right b
```

Here `a` and `b` are type variables. Values of the data type `Either String Int` for example can be constructed by writing `Left "hello"` or `Right 5`. For the above reasons, we chose Haskell as the language to implement our solver in.

[1] https://github.com/meipp/nielsen-transformation

4.2 String Solvers

String solvers are programs that solve string-related problems. As an input, they are given a *test instance* consisting of a word equation or an instance for some other string problem. They then attempt to solve this problem in a given time interval. By convention they produce the outputs *sat* if the problem is satisfiable, *unsat* if unsatisfiable, *unknown* if the algorithm in use cannot determine the problem's satisfiability, or *timeout* if a predefined timeout has been reached.

4.3 SMT-LIB

In the string solving community, benchmarks and test instances usually come in a common file format: The SMT-LIB [12] format. SMT-LIB is a collaborative effort to maintain and publish a format that can be used by all solvers. SMT-LIB files all have a structure similar to this example:

```
1 (declare-const x String)
2 (declare-const y String)
3 (assert (= (str.++ x "b") (str.++ "a" y "b")))
4 (assert (str.in_re x (re.++ (str.to_re "a") (re.* (str.to_re "b")))))
5 (assert (= (str.len y) 5))
6 (check-sat)
```

In lines 1 and 2 we declare two variables `x` and `y`. As SMT-LIB also supports other data types, we must explicitly declare `x` and `y` to be string-valued variables. Lines 3 to 5 define all logical statements we `assert` to be true. Line 3 defines the word equation $\mathtt{x}b = a\mathtt{y}b$. Line 4 introduces the regex constraint $\mathtt{x} \in ab*$. Line 5 introduces the length constraint $|\mathtt{y}| = 5$. Line 6 instructs the solver to check for satisfiability. This word equation is satisfiable with exactly one solution: $\mathtt{x} = abbbbb, \mathtt{y} = bbbbb$.

4.4 Architecture

Our solver consists of four modules:

- Algorithmic Core
- User definable Regex Theory
- SMT-LIB Interface
- Solver Frontend

Figure 4.1 shows the architecture of our implementation. At its heart is the library with the generic implementation of our algorithm that can be extended by a user definable regex theory. The library can be used in Haskell programs or in most other languages, as Haskell's foreign function interface

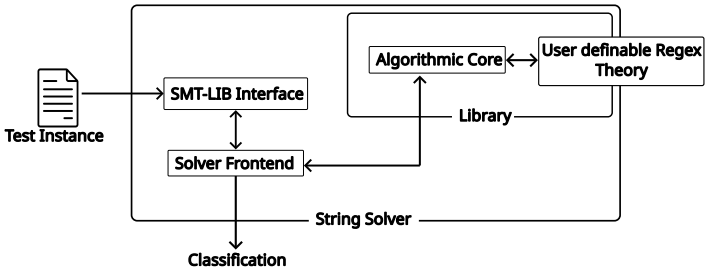

Figure 4.1: Architecture

conforms to C-style function calls. The user defined regex theory is visualized as halfway part of the library and halfway on the outside because users can either use our implementation of Brzozowski's regex theory, or they can provide their own implementation. The algorithmic core is written to be completely agnostic to the regex theory used.

If the user instead decides to use our implementation as a full string solver, they have to call it on an SMT-LIB file. The solver frontend calls the SMT-LIB interface on the test instance. The SMT-LIB interface parses the instance to a regex-constrained word equation. The frontend then makes a library call to determine the satisfiability of the equation and outputs one of the results *sat* or *unsat*. The following sections give a detailed description of the four modules.

4.4.1 User Definable Regex Theory

The core of our approach does not care about the particular regex theory. The regex theory needs only define the functions D, ν, σ, π. Therefore we keep it abstract and define a type class

```
class RegexTheory r where
    derive :: Char -> r -> r
    nullable :: r -> Bool
    satisfiable :: r -> Bool
    possiblePrefixes :: r -> [Char]
```

Every data type `r` that is an instance of the class `RegexTheory`, i.e. that implements the four functions `derive`, `nullable`, `satisfiable`, `possiblePrefixes`, can now be used to constrain variables. In chapters 2 and 3, we already defined D, ν, σ, π for Brzozowski derivatives. Our implementation of Brzozowski regexes instantiates the class `RegexTheory`.

Note that σ and π are just optimizations and are technically speaking not needed in a minimal implementation. If we want to disregard them, we can provide the default implementations

```
satisfiable r = True
possiblePrefixes r = sigma
```

This has the effect that satisfiability checks for any regex return `True`. Unsatisfiable derivatives now do not get detected and discarded early. Still, this does not affect the correctness because unsatisfiable variables can never be deleted from the equation as they are not nullable ($\neg\sigma(p) \Rightarrow \neg\nu(p)$). Therefore, it is impossible to reduce a word equation that contains an unsatisfiable variable to $\varepsilon = \varepsilon$. The default implementation of the prefix function just returns the entire alphabet Σ. This also does not affect correctness because adding more symbols a to calculate the derivative of, only results in more unsatisfiable regexes that again are ultimately not nullable.

4.4.2 Algorithmic Core

The algorithmic core is the backend of our solver and is responsible for handling the core algorithm as defined in chapter 3.1. We make the following definitions:

```
data Terminal = Terminal Char
    deriving Eq

data Variable r = Variable Char r
    deriving Eq
```

We introduce a new datatype `Terminal`. Each `Terminal` value is parameterized by only a `Char` value. E.g. to represent the terminal symbol a, we write `Terminal 'a'`. Similarly, we introduce `Variable r`, where `r` is a type parameter. Each variable is parameterized by a `Char`, i.e. its name, and by a regex value of type `r`. E.g. to represent the variable $x, c(x) = p$, we write `Variable 'x' p`. `deriving Eq` instructs the compiler to implement the canonical equality relation for our data types. I.e. `Terminal a == Terminal b` $\Leftrightarrow$ `a == b`.

For simplicity we introduce the following type aliases.

```
type Symbol r = Either Terminal (Variable r)

type Sequence r = [Symbol r]

type Equation r = (Sequence r, Sequence r)
```

A symbol is either a terminal or a variable. A sequence is a list of symbols. An equation is a tuple of two Sequences. We can now define the function

```
nielsen :: RegexTheory r => Equation r -> Bool
```

`nielsen` takes a word equation (constrained by some `RegexTheory r`) and returns `True` if it is satisfiable, `False` if it is unsatisfiable. As of now, we have implemented neither a timeout feature nor parallelization. Note, that we do not have to take care of *unknown*s, because our algorithm can classify any input as either *sat* or *unsat*.

4.4.3 SMT-LIB Interface

The SMT-LIB interface parses `.smt` files and extracts word equations from them. Currently, we only support the subset of the SMT-LIB standard necessary to handle regex-constrained word equations, namely those functions required to define regex constraints, word equations as well as the instructions `not` and `assert`. We only support variables of type `String`. Functions like `str.len` are not supported.

Also, the parser is not agnostic to the regex theory and currently parses every regex constraint as a Brzozowski regex. This has the consequence that only the library is truly independent of the regex theory. For now, when using the string solver, we are bound to Brzozowski derivatives. One could solve this problem by adding a generic regex type

```
data SmtLibRegex = Lambda
                 | Emptyset
                 | Symbol Char
                 | Concatenation SmtLibRegex SmtLibRegex
                 | Iteration SmtLibRegex
                 | Not SmtLibRegex
                 | And SmtLibRegex SmtLibRegex
                 | Or SmtLibRegex SmtLibRegex
```

and the accompanying type class

```
instance FromSmtLibRegex r where
    fromSmtLibRegex :: SmtLibRegex -> r
```

The parser then parses regex constraints as values of the type `SmtLibRegex`. Those values can then get converted to our custom regex theories, as long as they provide an implementation for `fromSmtLibRegex`.

4.4.4 Solver Frontend

The frontend handles the user interaction. It takes the file path of the `.smt` file as a command line argument, invokes the parser to parse a word equation and calls `nielsen` on it. It then outputs the result to `stdout`. The frontend can be expressed in the following pseudo code fragment:

```
filepath <- args[1]
equation <- SmtLibParser.parse(filepath)
if nielsen(equation)
    print "sat"
else
    print "unsat"
```

where `args[1]` is the first command line argument, `SmtLibParser.parse` is a function that parses word equations with Brzozwski regex constraints from a file, and `nielsen` is the function introduced in section 4.4.2. Our implementation does not return *unknown* on any instances and does not detect timeouts. For word equations that cannot be decided in finite time, the function `nielsen` does not terminate.

Chapter 5

Benchmarks

When comparing different approaches, the need arises to rank one against the other. For that, one usually employs benchmarks over benchmark sets. Benchmark sets are a collection of test instances used to evaluate the performance of algorithms. In the case of string solving, benchmark sets are usually in the SMT-LIB format and may consist of word equations, length constraints and regular expression constraints. These instances are usually either designed or curated to cover different aspects of the field. For example, one benchmark set may focus solely on small word equations with regex-constrained variables, while another may contain only very complex equations with length constraints.

This chapter focuses on the setup of our benchmarks. First, we explain the creation and makeup of the benchmark sets for our experiments. Then, we introduce the other string solvers we will compete against. Finally, we introduce the benchmarking framework ZaligVinder [13] that we will use to run the benchmarks.

5.1 Building the Benchmark Sets

First, let us have a look at how the benchmark sets are created.

5.1.1 SMTQuery

To automatically construct benchmark sets, we used the tool SMTQuery [14]. SMTQuery allows running SQL-style queries on a predefined corpus of test instances. The query

```
select Name from kaluza where isQuadratic
```

lists the name of all test instances from the Kaluza [15] benchmark set in which each variable occurs at most twice.

As a base corpus we use the ZaligVinder corpus [13], which is a compilation of the following benchmark sets:

Benchmark set	Instances
appscan [16]	8
automatark25 [17]	19979
banditfuzz [18]	357
cashewsuite [19]	394
joacosuite [20]	94
kaluza [15]	47284
kauslersuite [21]	120
Leetcode [22]	2666
light [23]	100
nornbenchmarks [24]	1027
pisa [16]	12
PyEx_All [25]	25421
slothtests [26]	33
strangersuite [20]	4
stringfuzz [27]	1065
stringfuzzregexgenerated [17]	4170
stringfuzzregextransformed [17]	10682
woorpje [28]	809
z3_regression [29]	243
Σ	114468

This includes word equations – quadratic or more complex –, regex matching, and other string problems. We will need to restrict the corpus when building our benchmark sets.

5.1.2 benchmarkset-1

Our approach solves word equations with regex-constrained variables. Thus, we are mainly interested in assessing its performance on test instances that follow this exact pattern. As the Nielsen transformation is only guaranteed to terminate on quadratic word equations [8], we will also restrict the benchmark set accordingly. Thus, we are interested only in **quadratic word equations with regex variable constraints**. This leaves us with the SMT query

```
select Name from * where (hasWEQ and (isQuadratic and hasRegex))
```

This query selects all test instances over all benchmark sets for which the following restrictions hold:

1. can be expressed as a word equation
2. the word equation is quadratic
3. the test instance contains at least one regex constraint

This initially leaves us with 2859 out of 114468 instances. We still cannot process all of them. Therefore, we delete more files for the following reasons:

Reason	Instances
declares Int variable	25
declares Bool variable	3
calls str.len	422
calls str.to_int	85
contains multiple equations	176
contains inequality	1
Σ	712

This concludes the construction of the benchmark set, leaving us with 2147 files.

5.1.3 benchmarkset-2

`benchmarkset-1` deals only with word equations where at least one variable has a regex constraint. We model unconstrained variables with the catch-all constraint $\emptyset'$. To assess the performance of our approach in a more general context, we drop the restriction that our variables must have regex constraints. We are left with a set of **quadratic word equations**. We construct the `benchmarkset-2` with the SMT query

```
select Name from * where (hasWEQ and isQuadratic)
```

This leaves us with initially 26315 out of 114468 instances. Again, we clear the benchmark set of unprocessable instances:

Reason	Instances
declares Int variable	171
declares Bool variable	21814
calls str.len	516
calls str.to_int	85
calls to ite (if-then-else)	51
contains multiple equations	976
contains inequality	157
Σ	23770

This concludes the construction of the benchmark set with 2547 files.

We constructed this benchmark set with generality in mind, but we are limited by the fact that we only parse a subset of the SMT-LIB standard. In `benchmarkset-1` we reduced 2859 to 2147 files, a 24.9% decrease. In `benchmarkset-2` on the other hand we have a decrease by 90.3% going from 26315 files initially to just 2547 in the end. We interpret this by assuming that there are not many test instances across the literature that contain unconstrained word equations. Most instances contain regex or length constraints. Thus, it is hard to find a truly general setting because a majority of the instances seems to be special cases.

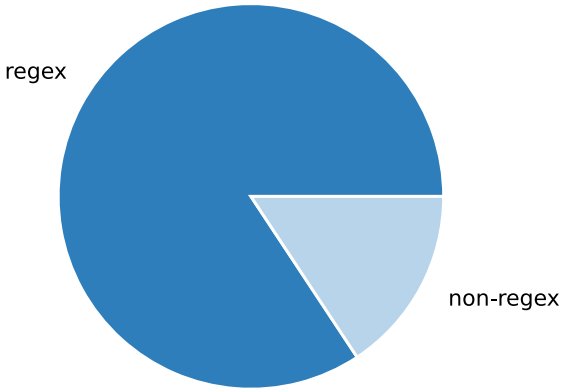

Figure 5.1: Makeup of `benchmarkset-2`, divided in regex and non-regex word equations

Figure 5.1 visualizes the makeup of `benchmarkset-2`. `benchmarkset-2` Is a superset of `benchmarkset-1`. It consists of the 2147 regex-constrained instances from `benchmarkset-1` and 400 more unconstrained instances.

5.1.4 benchmarkset-2 \ benchmarkset-1

We constructed `benchmarkset-2` to assess our approach on a more general setting. Seeing that `benchmarkset-2` is bigger than `benchmarkset-1` by only 400 files (18.6% increase), a quantitative analysis would be heavily biased towards instances with regex constraints, skewing our results. Therefore, we will also benchmark on the set difference `benchmarkset-2 \ benchmarkset-1`, consisting only of quadratic, unconstrained instances that contain one word equation. This set should show the performance of our approach in a general setting that is not our algorithmic niche.

5.2 Other String Solvers

This section discusses and introduces the string solvers we will benchmark against. We compare our approach to the state-of-the-art [13] solvers CVC4 [3], Z3str3 [30] and Z3seq [5]. We also include the successor CVC5 [4]. Additionally, we compare to two other, less general solvers – Woorpje [28] and Noodler [6] – because they share some interesting similarities with our approach.

5.2.1 CVC4

CVC4 [3], the Cooperating Validity Checker 4, is a string/SMT solver that reduces its queries to SAT-formulae and solves them with a SAT-solver backend. It supports different *Theories*, generates proofs and satisfying assignments. CVC4 supports highly parallelized execution. It is a collaborative effort between Stanford University and the University of Iowa.

CVC4 has been succeeded by CVC5 [4]. For the benchmarks we use version 1.8, which is the final release version of CVC4.

5.2.2 CVC5

CVC5 [4] is the successor to CVC4. CVC5 introduced a new proof engine, procedures for fixed-size bit vectors and non-linear arithmetic. Barbosa et al. [4] also claim that it outperforms CVC4 in terms of speed. As CVC4 has not been fully dethroned by its successor yet, we include both solvers for our benchmarks.

For our benchmarks we use version 1.0.3 of CVC5.

5.2.3 Z3

Z3 [5] is a general-purpose SMT solver published by Microsoft Research. It can handle quantifier-free as well as quantified formulae. Z3 cannot generate proofs, but it can generate models of satisfying assignments. Z3 contains multiple sub-solvers, called *tactics*.

For our benchmarks we use version 4.8.10 of Z3. We benchmark against the following two state-of-the-art tactics.

Z3seq

Z3seq [5] is a tactic that processes strings in a prefix-based manner. Similar to our approach, Z3seq uses regex derivatives to check regex constraints. It differs from our approach in that it checks constraints and equations separately.

Z3str3

Z3str3 [30] is a solver tactic for Z3 that introduces a technique labeled *theory-aware branching*, which extends Z3's branching heuristics allowing to pass information to the internal SAT-solver. Berzish et al. [30] claim, this technique improves the search step in Z3's solving engine, allowing for better performance.

5.2.4 Woorpje

The Woorpje string solver [28] specializes on word equations where every variable has an upper length bound, guessing unknown lengths. It reduces the bounded word equation to a SAT problem and solves it via the Glucose SAT solver [31]. Woorpje also supports regex constraints and offers different solving strategies. More interestingly, Woorpje has one strategy that uses the Nielsen transformation, although Day et al. [28] refer to it as *Levi's lemma*. As they use the traditional Nielsen transformation, this strategy only works for word equations without regex constraints. We will therefore use it only for benchmarking on `benchmarkset-2 \ benchmarkset-1`. In the benchmarks, we will refer to Woorpje with the general strategy as `woorpje`, and to Woorpje with Levi's lemma as a strategy as `woorpje-levi`. This strategy is interesting because we can benchmark against a string solver that uses the same approach as ours.

In our benchmarks we use the release `spin22` of Woorpje.

5.2.5 Noodler

Noodler [6] is the only string solver we are aware of that solves the word equation and the regex constraints together. Blahoudek et al. do this by representing each variable by an automaton and modeling the equation as concatenations of these automata.

As Noodler is not versionated, we used the commit `f752e79`[1] in our benchmarks.

This concludes the setup for our benchmarks. Figure 5.2 gives a compact overview of the different string solvers we will use. Our solver is listed at the bottom as `nielsen-transformation`. We use version 0.1.0 of our solver[2].

[1] https://github.com/vhavlena/Noodler/commit/f752e79eb9c0cded7e7e1bf73476c3b1ef1f25b1
[2] https://github.com/meipp/nielsen-transformation/releases/tag/0.1.0

Solver	Version	State of the art	Can solve regex	Combined solving
CVC4	1.8	✓	✓	✗
CVC5	1.0.3	✓	✓	✗
Z3Seq	4.8.10	✓	✓	✗
Z3str3	4.8.10	✓	✓	✗
woorpje	spin22	✗	✓	✗
woorpje-levi	spin22	✗	✗	✗
noodler	f752e79	✗	✓	✓
nielsen-transformation	0.1.0	✗	✓	✓

Figure 5.2: Comparison of previously introduced solvers. *Combined solving* means that the solver solves the constraints and the word equation in a combined manner

5.3 ZaligVinder

To run the benchmarks we use the benchmarking framework ZaligVinder [13]. ZaligVinder brings benchmark sets and string solvers together on a shared platform. We configure it with test instances from our benchmark sets. For each solver, a shell command has to be registered that gets invoked on each test instance. ZaligVinder then counts all classifications. In case a solver does not terminate, ZaligVinder will count it as a timeout. Afterwards ZaligVinder produces tables for the measurements, in the format of appendix A.

Chapter 6

Results

In this chapter, we present the findings of our benchmarks and a subsequent analysis. We ran the benchmarks on a system with an Intel i7-7500U CPU with four cores (2.70GHz base frequency, 3.50GHz Intel Turbo Boost) and 16GB of DDR4 memory. The complete results can be found in appendix A. We used this[1] particular setup for the benchmarks.

CVC, Z3 and our solver worked fine. Woorpje and Noodler had some problems with the regex-constrained equations. For Woorpje those errors are linked to the policy of guessing lengths. Noodler – we found – could not read the many Unicode escape sequences in the SMT-LIB files. This influences the correctness but not the time performance: A solver that immediately exits with *unknown* does not get a time penalty.

6.1 Benchmarks

In this section we will present the results of the benchmarks on our three benchmark sets. The next section will give an interpretation of the results.

6.1.1 benchmarkset-1

Figure 6.1 shows the results of the benchmark on `benchmarkset-1` in the output format of ZaligVinder. The column *Satis*, *NSatis*, *Unknown* refer to the number of instances the solver classified as satisfiable, unsatisfiable and unknown respectively. Unknowns can be either produced when the solver outputs an explicit "unknown" or when the solver does not produce an explicit "sat" or "unsat", crashes or does not give intelligible output. *Timeout* refers to the number of instances for which the solver did not terminate within 20 seconds. The column *Errors* refers to the number of instances where the program crashed or error output occurred. Note that errors still get counted as satisfiable, unsatisfiable or unknown. *Total Time* refers to the time (in milliseconds) each

[1] https://github.com/meipp/zaligvinder/releases/tag/0.1.0

Solver	Satis	NSatis	Unknown	Timeout	Errors	Total Time	Total Time w/o Timeout
CVC4	747	1400	0	0	0	240297	240297
CVC5	747	1400	0	0	0	247331	247331
Z3seq	748	1398	0	1	55	865382	845382
Z3str3	748	1398	0	1	55	950974	930974
woorpje	393	643	684	427	126	13319883	4779883
noodler	243	824	637	443	0	27816800	18956584
nielsen-transformation	747	1398	0	2	0	251808	211808

Figure 6.1: Benchmark over `benchmarkset-1`

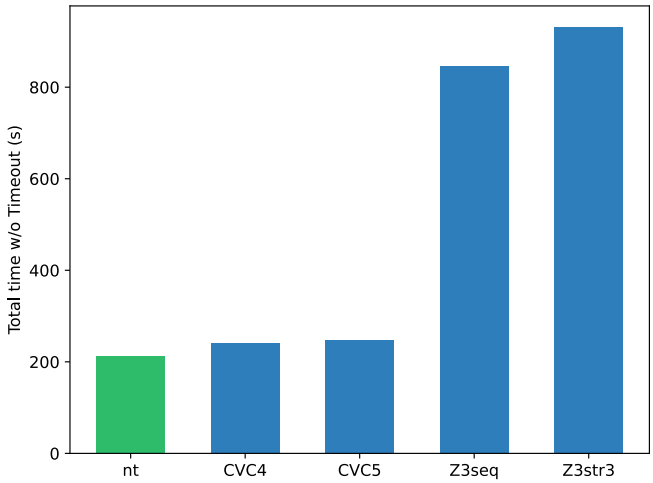

Figure 6.2: Comparison of CVC, Z3 and our solver on `benchmarkset-1` (lower is better). `nt` refers to our solver.

solver took to solve all instances combined – including a penalty of 20 seconds per timeout. *Total Time w/o Timeout* refers to the overall time without that penalty.

The state-of-the-art solvers CVC and Z3, solve all instances without *unknown*s. The Z3 solvers timeout on one instance while both CVC solvers can solve all instances within the time limit. Aside from the timeout, Z3 and CVC differ on only one other instance, with CVC classifying 747 instances and Z3 classifying 748 as satisfiable. The solvers Woorpje and Noodler perform poorly across all metrics. They do not reach similar classifications like CVC or Z3. They return many unknowns, timeout on relatively many instances and frequently produce errors. Ultimately, their running time is also significantly worse. While unknowns may be blamed on them not being able to read some instances, the timeouts cannot. Our solver, `nielsen-transformation`, is very close to the state-of-the-art solvers. We reach comparable results with the same number of satisfiable classifications as CVC and only one more timeout than Z3. Most importantly, looking at the *Total Time w/o Timeout*, we outperform Z3 by far and are also slightly faster than CVC.

Solver	Satis	NSatis	Unknown	Timeout	Errors	Total Time	Total Time w/o Timeout
CVC4	1055	1491	0	1	0	326543	306543
CVC5	1055	1491	0	1	0	354380	334380
Z3seq	1045	1409	0	93	55	2908422	1048422
Z3str3	1057	1408	5	77	55	2571637	1031637
woorpje	693	651	695	508	127	15410164	5250164
noodler	544	888	665	450	0	32926399	23926181
nielsen-transformation	1011	1453	0	83	0	2371652	711652

Figure 6.3: Benchmark over `benchmarkset-2`

Figure 6.2 highlights the performance of our solver against CVC and Z3. The margin by which we outperform CVC is slim, but we are an order of magnitude faster than Z3.

6.1.2 benchmarkset-2

Figure 6.3 shows the benchmark on `benchmarkset-2`. This includes regex-constrained and unconstrained word equations. We observe roughly the same patterns as in figure 6.1: CVC performs well with just one timeout. Woorpje and Noodler perform poorly again. Z3 looks very different than before: It has a significantly higher number of timeouts with 93 and 77 timeouts for Z3seq and Z3str3 respectively. One interesting observation is that Z3seq and Z3str3 behave differently. Z3seq has 16 more timeouts and one more classification as unsatisfiable, while Z3str3 has 12 more classifications as satisfiable and five more unknowns. On one hand, Z3str3 seems to be better at avoiding timeouts than Z3seq. This could be an example of the *Theory-aware branching* at work. On the other hand, Z3str3 returns unknown for instances it cannot solve, while this behavior is seemingly not implemented in Z3seq. Our solver performs worse than CVC and roughly similar to Z3. It has 83 timeouts, placing it between Z3str3 and Z3seq. It solves less satisfiable word equations than Z3 (1011 vs. 1045/1057) but more unsatisfiable word equations (1453 vs. 1408/1409). This is interesting because our approach can only reach the result unsatisfiable after an exhaustive graph search, running the risk of timeout. In this regard, we perform better than Z3 because we have a higher number of classifications as unsatisfiable, while not showing a higher number of timeouts. When comparing running times, our solver still outperforms Z3, Woorpje and Noodler but is notably slower than CVC.

Figure 6.4 shows the performance of our solver against CVC and Z3. It can be placed pretty much in the middle, outperforming Z3 by a notable margin but also getting clearly outperformed by CVC.

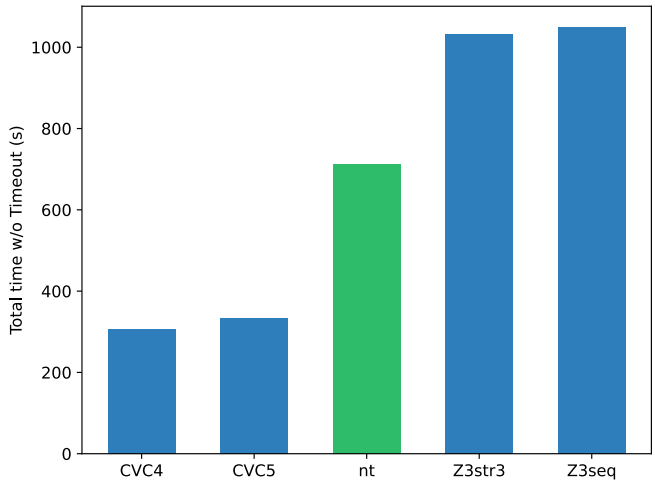

Figure 6.4: Comparison of CVC, Z3 and our solver on `benchmarkset-2` (lower is better). `nt` refers to our solver.

Solver	Satis	NSatis	Unknown	Timeout	Errors	Total Time	Total Time w/o Timeout
CVC4	308	91	0	1	0	90983	70983
CVC5	308	91	0	1	0	110981	90981
Z3seq	297	11	0	92	0	2085124	245124
Z3str3	309	10	5	76	0	1643121	123121
woorpje	299	4	8	89	0	2464010	684010
woorpje-levis	300	90	9	1	1	206817	186817
noodler	281	20	19	80	0	5256067	3656067
nielsen-transformation	264	54	0	82	0	2111959	471959

Figure 6.5: Benchmark over `benchmarkset-2 \ benchmarkset-1`

6.1.3 benchmarkset-2 \ benchmarkset-1

Figure 6.5 shows the benchmark on `benchmarkset-2 \ benchmarkset-1`. CVC and Z3 behave roughly the same as they did in 6.3. `woorpje` reached a satisfactory number of classifications on satisfiable instances but classified almost none as unsatisfiable. Noodler and our approach are completely unsatisfactory. Still, `woorpje`, Noodler and our solver do not perform worse than Z3 in terms of timeouts. `woorpje-levis`, on the other hand, performed really well, outperforming Z3 in terms of time and approaching CVC in terms of correctness. It also outperforms `woorpje` by far.

Figure 6.6 shows the performance of our solver against CVC, Z3 and `woorpje-levi`. It gets outperformed by all solvers. `woorpje-levi` is placed between Z3str3 and Z3seq, showing that the Nielsen transformation as a solving strategy is competitive with Z3.

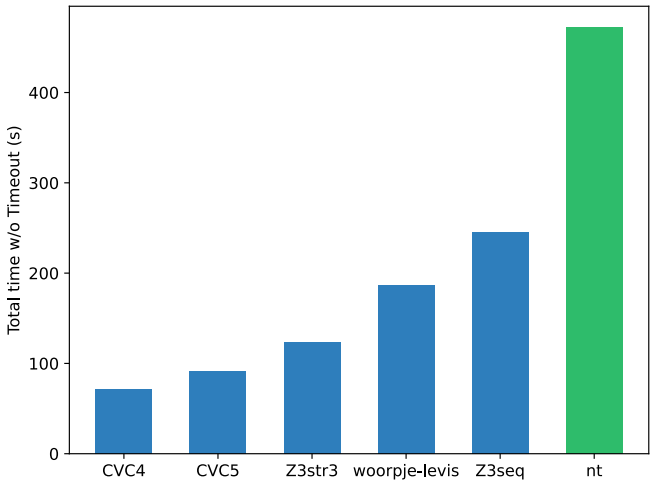

Figure 6.6: Comparison of CVC, Z3, woorpje-levi and our solver on `benchmarkset-2 \ benchmarkset-1` (lower is better). `nt` refers to our solver.

6.2 Interpretation

The interpretation of our findings is very straightforward: We perform well on `benchmarkset-1`, averagely on `benchmarkset-2` and badly on `benchmarkset-2 \ benchmarkset-1`. This can be directly linked to the composition of the benchmark sets.

`benchmarkset-1` fits perfectly to our problem description: Word equations with regex constraints. We solve these well and outperform every other string solver, even the highly optimized CVC solvers. This leads us to the conclusion that choosing a combined approach that solves both word equation and regex constraints at the same time gives us an edge over approaches that solve equations and constraints separately.

`benchmarkset-2` consists of mainly regex-constrained word equations and some unconstrained word equations. We immediately lose our edge over CVC. We still perform well enough to beat Z3, although this can be explained by the majority of the instances being regex-constrained. We outperform the other solvers. Still, it is notable that our approach loses its clear advantage over the other solvers as soon as we move out of our niche.

`benchmarkset-2 \ benchmarkset-1` consists solely of unconstrained word equations. Our solver gets outperformed by the state-of-the-art solvers. This makes sense because our only advantage was the combination of regex constraints with the word equation. As soon as we lose the regex constraints, we lose all foundation we had for an advantage. Not only that but now our solver can only make use of the Nielsen transformation and has to compete with other solvers that can choose from a multitude of tactics which may include the Nielsen transformation.

Subsequently, our solver performs significantly worse.

`woorpje-levis` – being competitive with Z3 – shows that the Nielsen transformation is viable for unconstrained word equations. It also shows that our implementation is not optimal in terms of running time. This is most likely due to Woorpje being more optimized on an instruction level. Also, we model unconstrained variables as $\emptyset'$ and unnecessarily calculate their derivative.

This concludes our findings. `woorpje-levis` shows that the Nielsen transformation is a valid tactic for solving unconstrained word equations. When combined with regex constraints in a combined approach, as shown by our reference implementation, the Nielsen transformation becomes a powerful tool outperforming even state-of-the-art solvers. Our approach was competitive inside of the specific niche of regex-constrained word equations. While it is not suitable as a general string solver, it could be used as a specialized tactic inside a portfolio solver.

Chapter 7

Conclusion

We presented an algorithm to solve regex-constrained as well as unconstrained word equations and provided a reference implementation. We compiled and published three benchmark sets specialized for Nielsen-based approaches. Our implementation outperforms current state-of-the-art solvers on quadratic, regex-constrained word equations. We conclude that our approach is best applied as a tactic.

While this thesis provided an initial proof of concept showing the viability of our approach, much future work can still be done.

Firstly, our solver supports neither length constraints nor equation systems containing more than one equation. This means we are not compatible with the entire SMT-LIB standard and thus cannot compete in string solving competitions.

Secondly, this solver is currently implemented as a standalone solver. As explained earlier, it would make most sense to implement it as a solver tactic for either one of the state-of-the-art solvers or in a portfolio solver.

Lastly, one could extend the algorithm in multiple ways. For once, we only implemented Brzozowski derivatives. One could instead implement transition regexes to see whether their claimed advantage [10] also holds within our approach. Furthermore, we noted that Dijkstra's algorithm can be used when given a heuristic. One could try different heuristics ranking word equations in terms of prospected solvability. Finally, we note that simple length constraints can also be expressed as derivative-based constraints. A future version of our solver could therefore possibly also handle length constraints. One last drawback of our algorithm is that it is not guaranteed to terminate on some unsatisfiable equations. This problem can be solved by making use of regex quotients.

Bibliography

[1] J. Nielsen, "Die isomorphismen der allgemeinen, unendlichen gruppe mit zwei erzeugenden," *Mathematische Annalen*, vol. 78, no. 1-4, pp. 385–397, 1917.

[2] J. A. Brzozowski, "Derivatives of regular expressions," *Journal of the ACM (JACM)*, vol. 11, no. 4, pp. 481–494, 1964.

[3] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 171–177. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_14

[4] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 415–442.

[5] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[6] F. Blahoudek, Y.-F. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Síč, "Word equations in synergy with regular constraints (technical report)," 2022. [Online]. Available: https://arxiv.org/abs/2212.02317

[7] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[8] J. D. Day and F. Manea, "On the structure of solution-sets to regular word equations," *Theory of Computing Systems*, pp. 1–78, 2021.

[9] V. Antimirov, "Partial derivatives of regular expressions and finite automaton constructions," *Theoretical Computer Science*, vol. 155, no. 2, pp. 291–319, 1996.

[10] C. Stanford, M. Veanes, and N. Bjørner, "Symbolic boolean derivatives for efficiently solving extended regular expression constraints," Microsoft, Tech. Rep. MSR-TR-2020-25, August 2020, updated November 2020. Extended version of paper in PLDI'2021. [Online]. Available: https://www.microsoft.com/en-us/research/publication/symbolic-boolean-derivatives-for-efficiently-solving-extended-regular-expression-constraints/

[11] S. Marlow *et al.*, "Haskell 2010 language report," *Available online http://www. haskell. org/(May 2011)*, 2010.

[12] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[13] M. Kulczynski, F. Manea, D. Nowotka, and D. B. Poulsen, "Zaligvinder: A generic test framework for string solvers," *Journal of Software: Evolution and Process*, p. e2400, 2021.

[14] J. D. Day, A. Kröger, M. Kulczynski, F. Manea, D. Nowotka, and D. B. Poulsen, "A generic information extraction system for string constraints," 2022. [Online]. Available: https://arxiv.org/abs/2208.08806

[15] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 513–528.

[16] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A z3-based string solver for web application analysis," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 114–124.

[17] M. Berzish, M. Kulczynski, F. Mora, F. Manea, J. D. Day, D. Nowotka, and V. Ganesh, "An smt solver for regular expressions and linear arithmetic over string length," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II*. Springer, 2021, pp. 289–312.

[18] J. Scott, F. Mora, and V. Ganesh, "Banditfuzz: A reinforcement-learning based performance fuzzer for smt solvers," in *Software Verification: 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20–21, 2020, Revised Selected Papers 13*. Springer, 2020, pp. 68–86.

[19] T. Brennan, N. Tsiskaridze, N. Rosner, A. Aydin, and T. Bultan, "Constraint normalization and parameterized caching for quantitative program analysis," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 535–546.

[20] J. Thome, L. K. Shar, D. Bianculli, and L. Briand, "An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving," *IEEE Transactions on Software Engineering*, vol. 46, no. 2, pp. 163–195, 2018.

[21] S. Kausler and E. Sherman, "Evaluation of string constraint solvers in the context of symbolic execution," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 259–270.

[22] T. Chen, X.-s. Zhang, R.-d. Chen, B. Yang, and Y. Bai, "Conpy: Concolic execution engine for python applications," in *Algorithms and Architectures for Parallel Processing: 14th International Conference, ICA3PP 2014, Dalian, China, August 24-27, 2014. Proceedings, Part II 14*. Springer, 2014, pp. 150–163.

[23] P. A. Abdulla, M. F. Atig, Y.-F. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer, "Trau: Smt solver for string constraints," in *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2018, pp. 1–5.

[24] P. A. Abdulla, M. F. Atig, Y.-F. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, "Norn: An smt solver for string constraints," in *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Springer, 2015, pp. 462–469.

[25] A. Reynolds, M. Woo, C. Barrett, D. Brumley, T. Liang, and C. Tinelli, "Scaling up dpll (t) string solvers using context-dependent simplification," in *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Springer, 2017, pp. 453–474.

[26] L. Holík, P. Janků, A. W. Lin, P. Rümmer, and T. Vojnar, "String constraints with concatenation and transducers solved efficiently," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–32, 2017.

[27] D. Blotsky, F. Mora, M. Berzish, Y. Zheng, I. Kabir, and V. Ganesh, "Stringfuzz: A fuzzer for string solvers," in *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*. Springer, 2018, pp. 45–51.

[28] J. D. Day, T. Ehlers, M. Kulczynski, F. Manea, D. Nowotka, and D. B. Poulsen, "On solving word equations using sat," 2019. [Online]. Available: https://arxiv.org/abs/1906.11718

[29] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, and X. Zhang, "Z3str2: an efficient solver for strings, regular expressions, and length constraints," *Formal Methods in System Design*, vol. 50, pp. 249–288, 2017.

[30] M. Berzish, V. Ganesh, and Y. Zheng, "Z3str3: A string solver with theory-aware heuristics," in *2017 Formal Methods in Computer Aided Design (FMCAD)*, 2017, pp. 55–59.

[31] G. Audemard and L. Simon, "On the glucose sat solver," *International Journal on Artificial Intelligence Tools*, vol. 27, p. 1840001, 02 2018.

Appendix A

Benchmarks

This chapter shows the benchmark results as produced by ZaligVinder. ZaligVinder partitions the benchmark sets into subsets, so-called tracks. `benchmarkset-1` has the track `kaluza` which corresponds with the instances of `benchmarkset-1` originally from the Kaluza benchmark set. Because of ZaligVinder's naming conventions, we refer to `benchmarkset-2 \ benchmarkset-1` as `benchmarkset-2-without-1`.

A.1 benchmarkset-1

Track: benchmarkset-1/joacosuite

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	2	0	0	0	0	2	459	459
CVC5	2	0	0	0	0	2	563	563
Z3seq	2	0	0	0	4	2	563	563
Z3str3	2	0	0	0	4	2	301	301
woorpje	0	0	2	0	0	2	194	194
noodler	0	0	2	0	0	2	2254	2254
nielsen-transformation	2	0	0	0	0	2	31	31

Track: benchmarkset-1/kaluza

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	323	0	0	0	0	323	40137	40137
CVC5	323	0	0	0	0	323	40894	40894
Z3seq	323	0	0	0	0	323	116686	116686
Z3str3	323	0	0	0	0	323	122030	122030
woorpje	323	0	0	0	0	323	118300	118300
noodler	0	0	323	0	0	323	3477941	3477941
nielsen-transformation	323	0	0	0	0	323	29044	29044

Track: benchmarkset-1/slothtests

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	1	4	0	0	0	5	810	810
CVC5	1	4	0	0	0	5	887	887
Z3seq	2	2	0	1	1	5	20889	889
Z3str3	2	2	0	1	1	5	21017	1017
woorpje	3	0	1	1	2	5	23362	3362
noodler	1	1	3	0	0	5	15861	15861
nielsen-transformation	1	2	0	2	0	5	40207	207

Track: benchmarkset-1/stringfuzzregextransformed

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	418	1392	0	0	0	1810	198491	198491
CVC5	418	1392	0	0	0	1810	204621	204621
Z3seq	418	1392	0	0	50	1810	725639	725639
Z3str3	418	1392	0	0	50	1810	806071	806071
woorpje	67	639	678	426	124	1810	13170095	4650095
noodler	239	820	308	443	0	1810	24299726	15439510
nielsen-transformation	418	1392	0	0	0	1810	182150	182150

Track: benchmarkset-1/z3_regression

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	3	4	0	0	0	7	400	400
CVC5	3	4	0	0	0	7	366	366
Z3seq	3	4	0	0	0	7	1605	1605
Z3str3	3	4	0	0	0	7	1555	1555
woorpje	0	4	3	0	0	7	7932	7932
noodler	3	3	1	0	0	7	21018	21018
nielsen-transformation	3	4	0	0	0	7	376	376

Total

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	747	1400	0	0	0	2147	240297	240297
CVC5	747	1400	0	0	0	2147	247331	247331
Z3seq	748	1398	0	1	55	2147	865382	845382
Z3str3	748	1398	0	1	55	2147	950974	930974
woorpje	393	643	684	427	126	2147	13319883	4779883
noodler	243	824	637	443	0	2147	27816800	18956584
nielsen-transformation	747	1398	0	2	0	2147	251808	211808

Benchmark took 20 minutes, 5 seconds

A.2 benchmarkset-2

Track: benchmarkset-2/cashewsuite

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	121	0	0	0	0	121	12706	12706
CVC5	121	0	0	0	0	121	13384	13384
Z3seq	121	0	0	0	0	121	41413	41413
Z3str3	121	0	0	0	0	121	42507	42507
woorpje	121	0	0	0	0	121	36453	36453
noodler	121	0	0	0	0	121	1219908	1219908
nielsen-transformation	121	0	0	0	0	121	11903	11903

Track: benchmarkset-2/kaluza

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	323	0	0	0	0	323	38734	38734
CVC5	323	0	0	0	0	323	41070	41070
Z3seq	323	0	0	0	0	323	115946	115946
Z3str3	323	0	0	0	0	323	123330	123330
woorpje	323	0	0	0	0	323	121773	121773
noodler	0	0	323	0	0	323	3406691	3406691
nielsen-transformation	323	0	0	0	0	323	30998	30998

Track: benchmarkset-2/slothtests

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	3	5	0	0	0	8	1234	1234
CVC5	3	5	0	0	0	8	1407	1407
Z3seq	4	3	0	1	1	8	22224	2224
Z3str3	4	3	0	1	1	8	22239	2239
woorpje	4	0	2	2	2	8	45100	5100
noodler	2	2	4	0	0	8	33160	33160
nielsen-transformation	3	3	0	2	0	8	40869	869

Track: benchmarkset-2/stringfuzzregextransformed

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	418	1392	0	0	0	1810	195276	195276
CVC5	418	1392	0	0	0	1810	201724	201724
Z3seq	418	1392	0	0	50	1810	695118	695118
Z3str3	418	1392	0	0	50	1810	778418	778418
woorpje	67	643	681	419	125	1810	12886536	4506536
noodler	257	862	317	374	0	1810	23735000	16254782
nielsen-transformation	418	1392	0	0	0	1810	171385	171385

Track: benchmarkset-2/woorpje

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	160	4	0	1	0	165	58997	38997
CVC5	160	4	0	1	0	165	70517	50517
Z3seq	149	4	0	12	0	165	418467	178467
Z3str3	161	4	0	0	0	165	68214	68214
woorpje	154	0	4	7	0	165	672842	532842
noodler	137	4	11	13	0	165	2692342	2432342
nielsen-transformation	116	3	0	46	0	165	1208948	288948

Track: benchmarkset-2/joacosuite

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	2	0	0	0	0	2	646	646
CVC5	2	0	0	0	0	2	721	721
Z3seq	2	0	0	0	4	2	454	454
Z3str3	2	0	0	0	4	2	435	435
woorpje	0	0	2	0	0	2	227	227
noodler	0	0	2	0	0	2	2300	2300
nielsen-transformation	2	0	0	0	0	2	72	72

Track: benchmarkset-2/stringfuzz

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	0	80	0	0	0	80	13707	13707
CVC5	0	80	0	0	0	80	17936	17936
Z3seq	0	0	0	80	0	80	1600000	0
Z3str3	0	0	4	76	0	80	1521530	1530
woorpje	0	0	1	79	0	80	1580695	695
noodler	0	11	6	63	0	80	1559094	299094
nielsen-transformation	0	45	0	35	0	80	866514	166514

Track: benchmarkset-2/z3_regression

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	28	10	0	0	0	38	5243	5243
CVC5	28	10	0	0	0	38	7621	7621
Z3seq	28	10	0	0	0	38	14800	14800
Z3str3	28	9	1	0	0	38	14964	14964
woorpje	24	8	5	1	0	38	66538	46538
noodler	27	9	2	0	0	38	277904	277904
nielsen-transformation	28	10	0	0	0	38	40963	40963

Total

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	1055	1491	0	1	0	2547	326543	306543
CVC5	1055	1491	0	1	0	2547	354380	334380
Z3seq	1045	1409	0	93	55	2547	2908422	1048422
Z3str3	1057	1408	5	77	55	2547	2571637	1031637
woorpje	693	651	695	508	127	2547	15410164	5250164
noodler	544	888	665	450	0	2547	32926399	23926181
nielsen-transformation	1011	1453	0	83	0	2547	2371652	711652

Benchmark took 26 minutes, 25 seconds

A.3 benchmarkset-2 \ benchmarkset-1

Track: benchmarkset-2-without-1/cashewsuite

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	121	0	0	0	0	121	10697	10697
CVC5	121	0	0	0	0	121	12488	12488
Z3seq	121	0	0	0	0	121	40995	40995
Z3str3	121	0	0	0	0	121	40944	40944
woorpje	121	0	0	0	0	121	36876	36876
woorpje-levis	121	0	0	0	0	121	43058	43058
noodler	121	0	0	0	0	121	988793	988793
nielsen-transformation	121	0	0	0	0	121	9789	9789

Track: benchmarkset-2-without-1/slothtests

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	2	1	0	0	0	3	60	60
CVC5	2	1	0	0	0	3	45	45
Z3seq	2	1	0	0	0	3	370	370
Z3str3	2	1	0	0	0	3	387	387
woorpje	1	0	1	1	0	3	20284	284
woorpje-levis	1	1	1	0	0	3	386	386
noodler	1	1	1	0	0	3	4261	4261
nielsen-transformation	2	1	0	0	0	3	268	268

Track: benchmarkset-2-without-1/woorpje

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	160	4	0	1	0	165	61777	41777
CVC5	160	4	0	1	0	165	72221	52221
Z3seq	149	4	0	12	0	165	430195	190195
Z3str3	161	4	0	0	0	165	67706	67706
woorpje	153	0	4	8	0	165	775557	615557
woorpje-levis	154	4	6	1	0	165	113009	93009
noodler	135	4	11	15	0	165	2487786	2187786
nielsen-transformation	116	3	0	46	0	165	1205558	285558

Track: benchmarkset-2-without-1/stringfuzz

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	0	80	0	0	0	80	13641	13641
CVC5	0	80	0	0	0	80	19572	19572
Z3seq	0	0	0	80	0	80	1600000	0
Z3str3	0	0	4	76	0	80	1521626	1626
woorpje	0	0	1	79	0	80	1580682	682
woorpje-levis	0	80	0	0	0	80	37558	37558
noodler	0	9	6	65	0	80	1573135	273135
nielsen-transformation	0	44	0	36	0	80	859710	139710

Track: benchmarkset-2-without-1/z3_regression

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	25	6	0	0	0	31	4808	4808
CVC5	25	6	0	0	0	31	6655	6655
Z3seq	25	6	0	0	0	31	13564	13564
Z3str3	25	5	1	0	0	31	12458	12458
woorpje	24	4	2	1	0	31	50611	30611
woorpje-levis	24	5	2	0	1	31	12806	12806
noodler	24	6	1	0	0	31	202092	202092
nielsen-transformation	25	6	0	0	0	31	36634	36634

Total

Solver	Satis	NSatis	Unknown	Timeout	Errors	SMT Solver Calls	Total Time	Total Time w/o Timeout
CVC4	308	91	0	1	0	400	90983	70983
CVC5	308	91	0	1	0	400	110981	90981
Z3seq	297	11	0	92	0	400	2085124	245124
Z3str3	309	10	5	76	0	400	1643121	123121
woorpje	299	4	8	89	0	400	2464010	684010
woorpje-levis	300	90	9	1	1	400	206817	186817
noodler	281	20	19	80	0	400	5256067	3656067
nielsen-transformation	264	54	0	82	0	400	2111959	471959

Benchmark took 6 minutes, 15 seconds