Hochschule für Technik, Wirtschaft und Kultur

Fakultät Informatik, Mathematik und Naturwissenschaften

Propositional Sieving of Single-Rule String Rewriting Systems Using Binary Decision Diagrams

Masterarbeit

Autor: Mario Wenzel
Betreuung durch Dr. rer. nat. Johannes Waldmann
und Prof. Dr. rer. nat. habil. Alfons Geser

October 4, 2016

Eidesstattliche Versicherung

Ich erkläre hiermit, dass ich diese Masterarbeit selbstständig ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen sind als solche einzeln kenntlich gemacht.

Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Mario Wenzel

Abstract:
This thesis describes a variant of Kurth's sieve, a method of generating single-rule string-rewriting systems that do not have straightforward termination or non-termination arguments and are therefore interesting. We approach this problem using binary decision diagrams (BDDs) to represent the set of rewriting systems and their properties.

During this work we will review the traditional approach of iteratively generating systems. Then we recapitulate BDDs and the useful algorithms within this data-structure.

We discuss methods for encoding rewriting systems as Boolean formulas and BDDs and present a Haskell framework to create BDDs that represent certain systems and their properties.

We explore predicates that are necessary for canonicity or sufficient for non-canonicity of the systems. We show properties that imply some (non)termination argument, like being a grid-rule or being bordered, and construct them as predicates within our framework.

We examine the impact on computation time and memory usage of various ways to construct the BDDs, like changing the variable ordering or changing the order of construction. The measurements show that constructing efficient formulas is non-obvious but it is possible faster than the traditional approach, given a reasonable amount of memory (16 Gigabytes of RAM) and specific search criteria.

We then count and generate all interesting systems with a right-hand side of up to length 14 and alphabet size 3 and hand them off to well-known termination tools (ttt2/AProVE) and filter all remaining systems where those tools can not give a termination or non-termination argument.

We present the 2 smallest systems that have no automated termination proof yet. Extending the approach to cycle termination, we also present the 3 smallest systems that have no automated cycle-termination proof yet.

Contents

1 Introduction

Termination is a crucial property of computations with immediate practical applications, in that the user wants a program to answer in finite time. [CPR06]

Methods for proving program termination are developed in various models of computation, and one fundamental model is (term) rewriting. In term rewriting, the rewriting rules can be thought of as the program while the term the rules rewrite, is the data. [Gie+06]

Methods are often developed from, and benchmarked on, small hard problems. For termination of rewriting, one such benchmark domain is termination of one-rule string rewriting. ([Kur90], [Ges02]) Analysis for small string rewriting systems have led to new approaches in proving (non-)termination for term-rewriting.

- The first automated termination proof for Zantema's problem [ZG00] $a^2b^2 \to b^3a^3$ obtained from (RFC) matchbounds [GHW04] was later generalized to term rewriting [KM09].
- The first termination proof (automated or not) for Zantema's other problem $a^2 \to bc, b^2 \to ac, c^2 \to ab$ by matrix interpretations [HW06] was also generalized later to term rewriting [EWZ08] and to complexity analysis [Wal10].

Even with one-rule string rewriting, there is a huge number of candidate problems, and most of them are not interesting, because their termination problem is trivial. To consider only the interesting systems, one could naively enumerate all, and then check each for "interestingness". This is a huge waste of effort. Instead one tries to write the generator program in such a way that it only generates interesting systems. [Ges02]

This approach quickly becomes impractical as well, because conceptually simple operations like Boolean combination of filter criteria are hard to realize, so an introduction of a new "non-interesting" criterion requires a complete rewrite of the generator, or a fall-back to the generate-and-test approach.

This thesis proposes representing sets of string rewriting systems as BDDs (reduced ordered binary decision diagrams). The advantage of this is:

- criteria can be expressed in a straightforward manner as a formula in propositional logic
- Boolean combination of criteria is realized by the BDD implementation
- from the resulting BDD (representing a set of "interesting" systems) one can easily compute the size of the set (without generating all its elements) [Knu09, Algorithm C]
- but finally one can generate elements (corresponding to paths in BDD) if reasonable (e.g., after intersection of several sets results in something that can be handled)

This approach is implemented in Haskell and using bindings to the well-known BDD C-library CUDD [Som15].

We recap the basic properties strings, and the theory of string rewriting and termination in Section 2 of this thesis. The traditional approach of generating and sieving rewriting systems, Kurth's Sieve, is analyzed and explained.

As a foundation for modelling rewriting systems and termination properties as Binary Decision Diagrams, BDDs, their structure, properties, construction, as well as algorithms concerning BDDs are discussed in Section 3.

The approach of modelling rewriting systems and termination properties as Boolean variables and formulas is explored in Section 4. This includes a discussion of implementation details, the Haskell API, and the constructed framework for BDD construction.

Section 5 contains implementations for basic predicates and helpers, that allow for an easier construction of complex predicates, which are discussed in Section 6. The latter includes a discussion of morphisms and embeddings between systems that maintain termination behaviour, allowing to show (non-)termination for just one system in an equivalence class, as well as predicates for termination criteria, like the inhibitor property, or grid property.

The resource usage of the BDD approach is discussed in Section 7. This entails a discussion of the efficiency of specific predicates and their combinations, different construction orders for the BDDs, different variable orderings, the growth of resource usage with the construction of larger systems, and a discussion on how to speed up and distribute the process by fixing and eliminating variables.

Actual hard and interesting systems that have been found during the course of this work, including a list of the smallest systems that have yet to be proven terminating or non-terminating by an automated termination prover for both string rewriting termination and cycle termination, are presented in Section 8.

Section 9 entails a discussion of the work and further areas of research.

A summary of this work, including preliminary results from sections 8.2 and 8.3, was submitted to and accepted at the 15th International Workshop on Termination (WST2016). Results from these chapters have been submitted to the annual Termination and Complexity Competition [Thi16]. That paper was written in collaboration with this thesis' supervisors Prof. Dr. rer. nat. habil. Alfons Geser and Dr. rer. nat. Johannes Waldmann.

2 Base Theories

In this section we introduce single rule rewriting on strings as a model of computation and Kurth's sieve as a method of finding specific instances of rewriting systems.

A string s is a sequence of n characters $s_0 s_1 \dots s_{n-1}$ over an alphabet Σ. The concatenation of strings s and t is st. $\tilde{s}$ is the reversal of s so that $\tilde{s} = s_{n-1} s_{n-2} \dots s_0$.

The length of a string s is $|s|$ and the number of occurrences of a letter $a \in \Sigma$ in s is $|s|_a$. The properties $|st| = |s| + |t|$ and $|st|_a = |s|_a + |t|_a$ follow naturally.

2.1 One Rule String Rewriting

This work examines single-rule string rewriting systems of the form $l \to r$ where l is called the left-hand side and r the right-hand side of the system or a group of systems. Both sides are strings over the alphabet Σ, therefore elements of Σ^*.

A rewriting rule $l \to r$ induces a one-step rewriting relationship $\to_{l \to r}$ over Σ^* that is defined as $\forall s, t \in \Sigma^* (\exists x, y \in \Sigma^* : s \to_{l \to r} t \iff s = xly \wedge t = xry)$. $\to^*_{l \to r}$ is the reflexive transitive closure of $\to_{l \to r}$. This is abbreviated to $\to$ and $\to^*$ respectively if it is unambiguous which rewriting rule is applied. The rewriting relationship is also called *reduction* relationship.

The rewriting reduction is compatible with concatenation meaning that $v \to^*_{l \to r} w \implies svt \to^*_{l \to r} swt$.

Single-rule string rewriting systems are a special case of string rewriting systems in that they only have a single rule and the system R is fully defined by $R = \{l \to r\}$. Since there are no further rewriting rules to a system, the words "rule" and "system" are used interchangeably in this work.

A string s is smaller than a string t if it is shorter than t or if they are the same length but smaller by lexicographic ordering induced by an order in Σ. So $s < t \iff (|s| < |t|) \vee (|s| = |t| \wedge s <_{\text{lex}} t)$.

Two interesting and generally undecidable properties of term- and string-rewriting systems are confluence and termination. For the class of one-rule string-rewriting systems, confluence, whether a system has the so called "diamond property", meaning that different sequences of rewriting steps applied to the same word can be reduced further to produce a common word, is known to be decidable [Wra90]. The question whether the termination problem for this class of rewriting systems is generally decidable, is still open and is therefore of importance.

We define $\mathsf{SN}(l, r) \iff l \to r$ is terminating, meaning that there is no infinite reduction chain $x_0 \to_{l \to r} x_1 \to x_2 \to \dots$.

Even though it is not known whether the termination problem is generally decidable for single rule string-rewriting systems, there are classes of systems within the class of single

rule string-rewriting systems for which the termination problem is indeed decidable. There are criteria C that imply termination of the systems in that class ($C(l,r) \implies \mathsf{SN}(l,r)$), non-termination of the systems in that class ($C(l,r) \implies \neg\mathsf{SN}(l,r)$), or that there is a decision procedure D that decides whether $l \to r$ terminates or not ($C(l,r) \implies (D(l,r) \iff \mathsf{SN}(l,r))$). For example the "length-decreasing" criterion ($C_{ld}(l,r) = |l| > |r|$) implies termination so all systems $l \to r$ that fit this criterion fall into a class of rewriting systems for which the termination problem is decidable in that the decision is always "yes".

2.2 Kurth's Sieve

Kurth's sieve is an iterative approach to generating, filtering and finding single-rule rewriting systems. Right-hand sides of systems are generated and checked for canonicity (see Section 6.1).

Given a right-hand side r, all left-hand sides l for that right hand-side are generated and the $l \to r$ system is checked for different criteria for which termination is (already) decidable. If the system fits any, it is "uninteresting". [Ges02]

The remaining systems are "interesting" and can be used to find new criteria for (non-)termination.

The most basic implementation of Kurth's Sieve for a right-hand side of size n and the filter criteria $C_1, C_2, \ldots$ would look like this:

Algorithm 1 naive Sieving over $\Sigma^{|l|} \to \Sigma^{|r|}$

Ensure: $|\Sigma| = n$
  for all $r \in \Sigma^n$ **do**
    if $r \leq \phi(r) \wedge r \leq \phi(\tilde{r})$ **then**
      for all $u \in [1 \ldots n-1]$ **do**
        for $l \in \Sigma^u$ **do**
          if $\neg C_1(l,r) \wedge \neg C_2(l,r) \wedge \ldots$ **then**
            $output \leftarrow l, r$
          end if
        end for
      end for
    end if
  end for

Where ϕ is the function that returns the lexicographically smallest renaming of the parameter (see Section 6.1.2) and $\tilde{w}$ is the reversal of w (see Section 6.1.3). This implementation ignores the special case where $r = \phi(\tilde{r})$ but extension to this case is trivial.

With larger n this algorithm's run time grows faster than exponential since the outer loop is executed n^n times. This can be significantly improved upon. Looking at the function ϕ we can see that every canonical renaming of w starts with the first letter of

Σ. Since only an nth of the generated right-hand sides starts with the first letter, if the first letter of the word is fixed, the run time can be improved by a factor of n.

This can be generalized to take the right-hand sides not from Σ^n but from the generator function $c_r(n, 1)$ with the following definition that generates only right-hand sides that are canonical after renaming.

Algorithm 2 Generating right-hand sides that are canonical after renaming

Ensure: $|\Sigma| \geq n$
  function $c_r(\mathit{length}, e)$
    if $\mathit{length} = 1$ **then return** $\Sigma_{1\ldots e}$
    else return $\Sigma_{1\ldots e} \times \mathrm{c_r}(\mathit{length} - 1, e) \cup \Sigma_{e+1} \times \mathrm{c_r}(\mathit{length} - 1, e + 1)$
    end if
  end function

This algorithm works with an arbitrarily large alphabet, that is ordered and can be sub-scripted, e.g. $\Sigma_1 = \{a\}$ and $\Sigma_{1\ldots3} = \{a, b, c\}$. The cardinality of the result of $\mathrm{c_r}(n, 1)$ can be bounded by $n!$ which is still at least exponential.

Now we can introduce the refined sieve including the function $\mathrm{alph}(w)$ that returns the alphabet of a word w. Using this function the left-hand side of a rule can be generated from the constricted alphabet of the right-hand side excluding the rules that delete a letter and therefore have a trivial termination problem (see Section 6.2.4).

Algorithm 3 Kurth's sieve, generating left-hand sides for every right-hand side that is canonical after renaming, and filtering the generated systems

Ensure: $|\Sigma| \geq n$
  for all $r \in c_r$(n,1) **do**
    if $r \leq \phi(\tilde{r})$ **then**
      for all $u \in [1 \ldots n-1]$ **do**
        for $l \in \mathrm{alph}(r)^u$ **do**
          if $\neg C_1(l, r) \wedge \neg C_2(l, r) \wedge \ldots$ **then**
            $\mathit{output} \leftarrow l, r$
          end if
        end for
      end for
    end if
  end for

This implementation, again, ignores the case where $r = \phi(\tilde{r})$. Note that Kurth's implementation iteratively generates the canonical right-hand sides of systems using a lexical-successor function. A programming language that supports lazy evaluation or generator expressions can efficiently implement the set-operations.

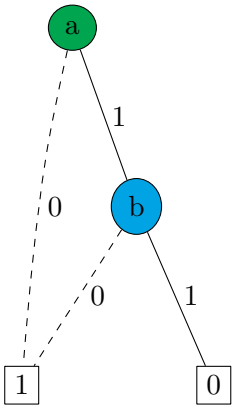
An implementation of $\mathrm{c_r}$ using Python's [Fou16] generator expressions that lazily generates and prints canonical (after renaming) right-hand sides looks like this:

```
from string import ascii_lowercase as alph

def cr(length, e):
    if length == 1:
        yield from alph[:e+1]
    else:
        for a in alph[:e]:
            for b in cr(length-1, e):
                yield a + b
        for b in cr(length-1, e+1):
            yield alph[e] + b

for r in cr(n, 0):
    print(r)
```

Listing 1: Generating canonical right-hand sides with Python

The implementation looks very similar to the presented algorithm with the set-operations unrolled into loops and concatenations, and changed indexing.

Another difference that is omitted here is, that Kurth and Geser, instead of just "throwing out" systems that do not fit the criteria $C_1, C_2, \ldots$ count how many systems were "thrown out" by which criterion. This is a trivial addition as well but doesn't serve the understanding of the traditional sieve.



ite-decomposition: $\text{ite}(a,\text{ite}(b,0,1),1) = (a \wedge ((b \wedge 0) \vee (\neg b \wedge 1))) \vee (\neg a \wedge 1)$

Figure 1: The function $f(a,b) = a \implies \neg b$ as a BDD as generated by PyEDA

3 Binary Decision Diagrams

In this section we introduce Binary Decision Diagrams (BDDs), their construction and properties. To illustrate certain features and procedures, we use code and generated diagrams from the BDD libraries PyEDA [Dra16] and CUDD [Som15].

3.1 Diagrams

A binary decision diagram is a rooted, directed, acyclic graph, where every node has either two or zero children, that represents a Boolean function $f(x_1, x_2, ..x_n)$ over the variables $x_1, x_2, ..x_n$. [Ake78] The nodes that are labeled $x_1, x_2, ..x_n$ have two children. The nodes labeled $\top$ and $\bot$ are called "sinks" and they don't have any children. The function is evaluated by traversing the graph from the root and visiting the left child (LO-branch), if the node's variable's assigned 0 or false, and the right child (HI-branch) if it is assigned 1 or true. This continues until either a $\bot$ node is reached and the function evaluates to false or a $\top$ node is reached and the function evaluates to true. [Knu09] [Bry86] Because the decision diagram is acyclic, this evaluation procedure eventually terminates.

Every node f with the left child g and the right child h represents the Boolean function $(f \wedge g) \vee (\neg f \wedge h)$. This is also called an *ite* (if-then-else) triple or *ite* function with the semantics $\text{ite}(f,g,h) = (f \wedge g) \vee (\neg f \wedge h)$ (see Figure 1). Every Boolean function can be decomposed into nested *ite* triples. These equivalences follow immediately from the definition:

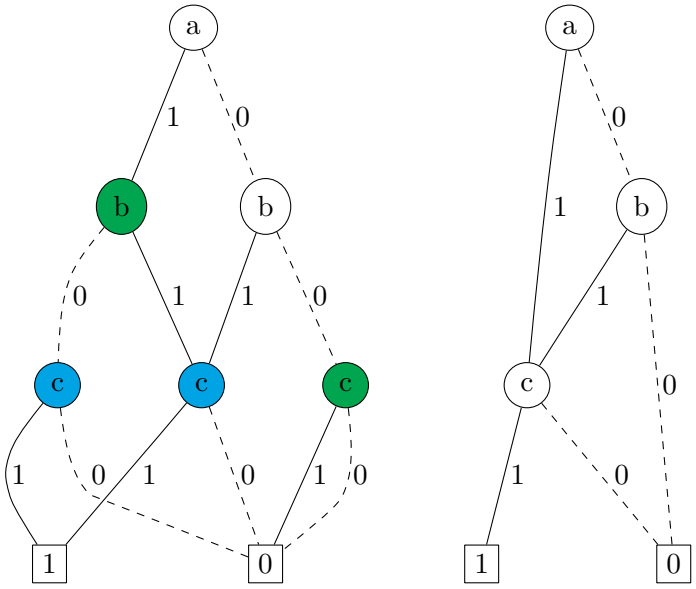


Figure 2: Unreduced and reduced BDD

$$a = \text{ite}(a, 1, 0) \tag{1}$$
$$\neg a = \text{ite}(a, 0, 1) \tag{2}$$
$$a \vee b = \text{ite}(a, 1, b) \tag{3}$$
$$a \wedge b = \text{ite}(a, b, 0) \tag{4}$$
$$a \implies b = \text{ite}(a, b, 1) \tag{5}$$

We call a Binary Decision Diagram *ordered* if, traversing from the root, the nodes are visited in the given order over the variables they are labeled with. Every node x_i can only have children that are labeled x_j for $i < j$ or are sinks. The left and right child of a node do not need to have the same label. If a Decision Diagram has this property (in that it is ordered), the aforementioned evaluation procedure terminates after at most n steps.

We call a Binary Decision Diagram *reduced* if no node has two identical children and there are no identical subgraphs in the BDD. Any node that has two identical children is removed from the diagram and any identical subgraphs are merged (see Figure 2). If a node is removed from the Decision Diagram because it has two identical children, it is replaced by either child node. If two subgraphs are merged, one subgraph is removed from the Decision Diagram and all its predecessors are changed to have the other subgraph as a child instead of the removed one.

We commonly refer to Reduced Ordered Binary Decision Diagrams (ROBDDs) only as BDDs. [Knu09]

A BDD represents all assignments where the Boolean function is true in a condensed form. The BDD representation of a Boolean function is canonical in that every Boolean function has one and only one representation as a ROBDD. [Knu09]

3.2 Variable Ordering

For the binary function $f(x_1, x_2, \ldots, x_n)$ we have to give an ordering for the n variables. A BDD for a function is canonical with regards to that specific variable ordering.

The ordering is quite important for the size of the BDD. Consider the function $f(a, b, c, d) = (a \iff d) \wedge b \wedge c$ which is equivalent to the function $g(a, b, c, d) = (a \iff b) \wedge c \wedge d$ up to renaming. Looking at the decision diagrams for both functions (see Figure 3), we see that for f the "conjunction chain" is duplicated until the question whether a equals d is resolved while for the function g the question whether a equals b is resolved immediately because both variables are close together with regards to the variable ordering.

The function $f(a, b, c, d, e, f) = (a \wedge b) \vee (c \wedge d) \vee (e \wedge f)$ is proposed as an example by Bryant [Bry86] for favorable and unfavorable variable orderings (see Figure 4). Bryant also constructed an example binary function that is exponential in size (to the number of input variables) for every variable ordering [Bry91].

3.3 BDD Construction and Combination

In order to construct large BDDs, existing BDDs can be merged using any logical function by applying the function to the two BDDs recursively from top to bottom. We can calculate the disjunction of the BDDs for the functions f and g over the same variables by evaluating $f \diamond_\vee g$. Knuth calls this operation "meld" ($\diamond$). Melding $f = (v_f, h_f, l_f)$ and $g = (v_g, h_g, l_g)$ is done using the following equivalence[Knu09]:

$$f \diamond g = \begin{cases} (v_f, l_f \diamond l_g, h_f \diamond h_g) & v_f = v_g \\ (v_f, l_f \diamond g, h_f \diamond g) & v_f < v_g \\ (v_g, f \diamond l_g, f \diamond h_g) & v_f > v_g \end{cases} \tag{6}$$

In this case the operators $=$, $<$, $>$ refer to the variable ordering and no particular value.

An example implementation in the Python language [Fou16] from the PyEDA library can be seen in Listing 2. This implementation generalizes the meld operation to a recursive application of the *ite* function, first calculating the new root of the resulting BDD node and then recursively applying *ite* for the children of the original nodes. We can also see that the step of removing nodes that have two identical children is done during the merge operation (`if g is h: return g`).

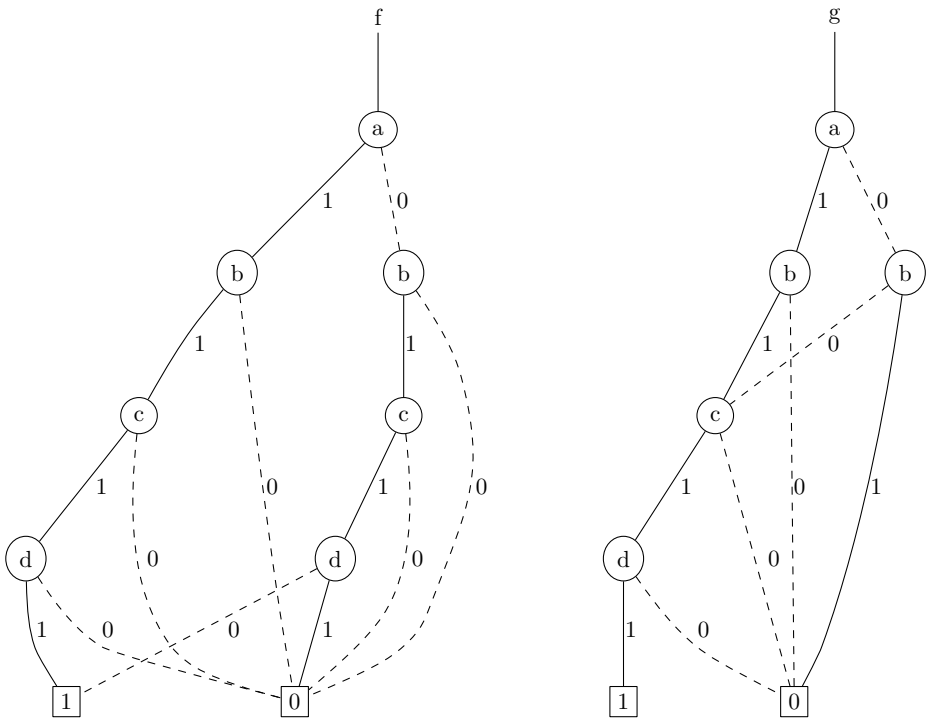
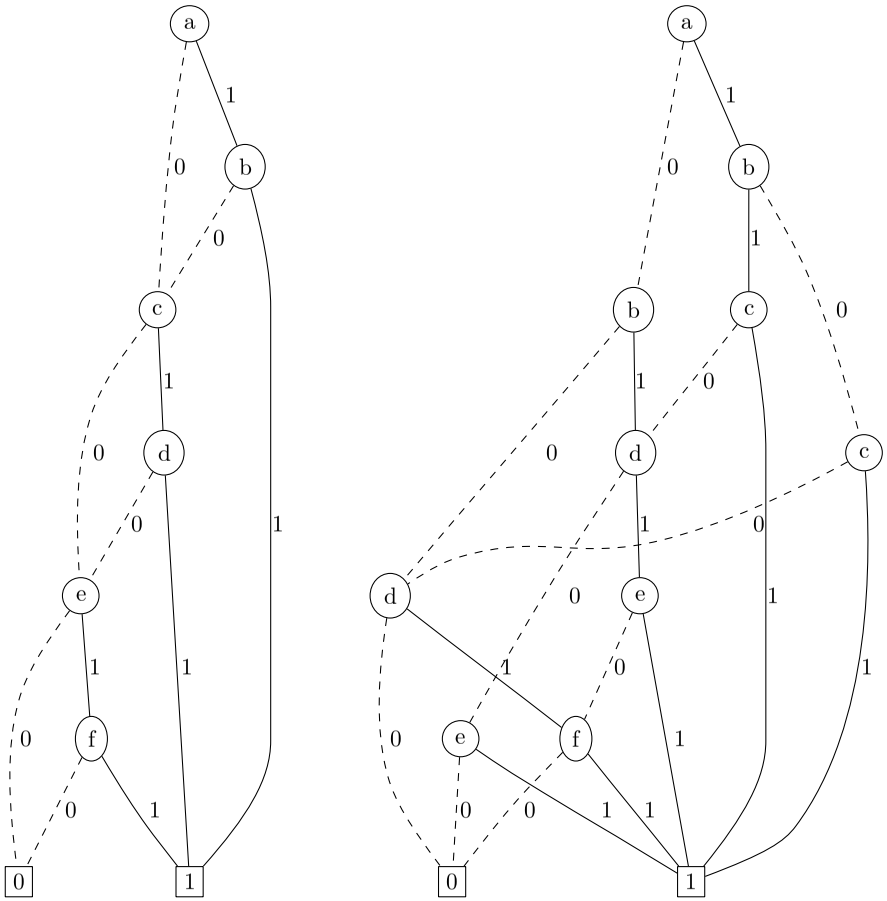


Figure 3: The functions $f(a,b,c,d) = (a \iff d) \wedge b \wedge c$ and $g(a,b,c,d) = (a \iff b) \wedge c \wedge d$ as BDDs



Variable ordering $a < b < c < d < e < f$ (left) and $a < d < b < e < c < f$ (right)

Figure 4: Favorable and unfavorable variable ordering for $f(a, b, c, d, e, f) = (a \wedge b) \vee (c \wedge d) \vee (e \wedge f)$

```
def _ite(f, g, h):
    """Return node that results from recursively applying ITE(f, g, h). """
    # ITE(f, 1, 0) = f
    if g is BDDNODEONE and h is BDDNODEZERO:
        return f
    # ITE(f, 0, 1) = f'
    elif g is BDDNODEZERO and h is BDDNODEONE:
        return _neg(f)
    # ITE(1, g, h) = g
    elif f is BDDNODEONE:
        return g
    # ITE(0, g, h) = h
    elif f is BDDNODEZERO:
        return h
    # ITE(f, g, g) = g
    elif g is h:
        return g
    else:
        # ITE(f, g, h) = ITE(x, ITE(fx', gx', hx'), ITE(fx, gx, hx))
        root = min(node.root for node in (f, g, h) if node.root > 0)
        npoint0 = {root: BDDNODEZERO}
        npoint1 = {root: BDDNODEONE}
        fv0, gv0, hv0 = [_restrict(node, npoint0) for node in (f, g, h)]
        fv1, gv1, hv1 = [_restrict(node, npoint1) for node in (f, g, h)]
        return _bddnode(root, _ite(fv0, gv0, hv0), _ite(fv1, gv1, hv1))
```

Listing 2: ite-implementation in PyEDA, from [Dra15]

3.4 Algorithms on BDDs

Knuth mentions, amongst others, following BDD-virtues we quickly want to recap:

- $f(x_1, x_2, ..x_n)$ can be evaluated in at most n steps.
- We can count the number of solutions to the equation $f(x_1, x_2, ..x_n) = 1$. This algorithm runs in linear time to the number of nodes in the BDD.
- We can speedily generate random solutions to the equation $f(x_1, x_2, ..x_n) = 1$.
- We can also list all solutions $x_1, x_2, ..x_n$ to the equation $f(x_1, x_2, ..x_n) = 1$. This algorithm runs in linear time to the number of solutions of the function.

The evaluation method for function f with the assignment $(x_1, x_2, \ldots, x_n)$ has already been discussed.

The number of solutions of the BDD encoded function $f(x_1, x_2, \ldots, x_n)$ can be calculated without generating a solution or the set of all solutions. All nodes are marked with the value 0 and the root node is marked with the value 1 if it is x_1 and 2^{n-1} for x_n as the root node in general. Now the value from the root is added to the value of its children, doubling the value for every variable in the ordering that has been skipped. This is done top to bottom for every node and all their children. For every node, its marked value is added to the value of its children, doubling the added value for every skipped variable. After the algorithm has finished, every node is marked with the number of paths it is reachable by, where every skipped variable represents two paths where the skipped variable is either assigned true or false. Now we can read the number the $\top$ node has been marked with. This is the number of models of the BDD.

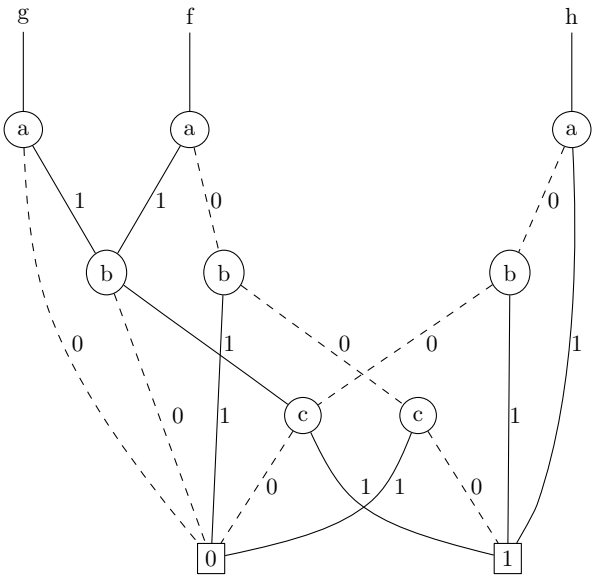
Using the former result we can now generate a random solution of the function where all solutions are equally likely to be generated. Again we start from the root and for every node we choose a child that does not lead to $\bot$ with a probability proportional to the number of paths that lead to this child compared to the paths that lead to the other child (if any). If the left child of a node is marked with n and the right child is marked with m, the variable is assigned true with the probability of $\frac{n}{n+m}$. If any variables are skipped, they are assigned true or false with equal probability. From any non-sink node in the BDD there is still at least one path that leads to $\top$ and also at least one path that leads to $\bot$ because if all children would lead to either $\top$ or $\bot$ the BDD would not be reduced.

We can generate all solutions of the function represented by the BDD by iterating over all paths of the BDD that do not lead to $\bot$. If this uses some generator or iterator pattern, the models can be generated one after another using constant memory independent of the actual number of solutions that are represented by the BDD.

3.5 BDD Base

A BDD Base is a graph representing multiple binary functions over the same variables $x_1, x_2, \ldots x_n$. The ordered and reduced properties hold for the whole graph. Nodes can additionally be annotated with the name of the function it represents. Using a BDD base facilitates sharing of subgraphs between all functions over the variables of the base (see Figure 5).

The BDD base is no longer rooted.



$$f(a,b,c) = (a \iff b) \wedge (b \iff c)$$
$$g(a,b,c) = a \wedge b \wedge c$$
$$h(a,b,c) = a \vee b \vee c$$

Figure 5: BDD base with three named functions

3.6 Complemented Edges

One optimization that is described by Akers [Ake78] are the complemented edges, also called *inverters*. Inverters allow not only sharing of some function f of the BDD base but also allows sharing occurrences of f with $\overline{f}$, in the best case halving the size of the BDD and in the worst case having no effect at all.

Figure 6 shows a BDD representing the binary function $f(a,b) = (a = b)$ or $\text{ite}(a, b, \overline{b})$. Without inverters, the occurrences of b and $\overline{b}$ can not be shared. With complement edges, the occurrence of b and $\overline{b}$ are shared and so are the occurrence of 1 and $\overline{1}$ (e.g. 0), as illustrated in Figure 7. Note that CUDD denotes a complement edge by a dotted line while a regular else-edge is represented by a dashed line.

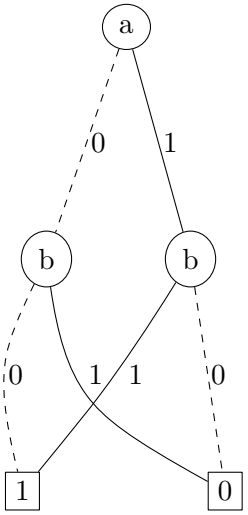


Figure 6: The function $f(a, b) = (a = b)$ as a BDD as generated by PyEDA

Figure 7: The function $f(a, b) = (a = b)$ as a BDD with complement edges as generated by CUDD

If complement edges are allowed to be used on every edge, the BDD is possibly no longer a canonical representation of a function because multiple BDDs can represent the same function. For example the BDDs ite$(\overline{a}, b, \overline{b})$ and $ite(a, \overline{b}, b)$ are different while their solutions (both representing $a \neq b$) are not. Allowing complement edges only at the root of a BDD and the else-edges of an *ite*-operation leads to canonicity once again. Under these restrictions, $ite(\overline{a}, b, \overline{b})$ is the correct BDD for the function $a \neq b$ while ite$(a, \overline{b}, b)$ is not, because it has an inverter on a then-edge.

Given the following equivalences, a BDD with complement edges can be transformed to be of the canonical form (forms without the then-edge being complemented on the right):

1. ite$(\overline{a}, \overline{b}, \overline{c})$ = ite(a, b, c)
2. ite$(a, \overline{b}, \overline{c})$ = ite$(\overline{a}, b, c)$
3. ite$(\overline{a}, \overline{b}, c)$ = ite$(a, b, \overline{c})$
4. ite$(a, \overline{b}, c)$ = ite$(\overline{a}, b, \overline{c})$

For optical reasons ite$(\overline{a}, b, c)$ is preferred over $\overline{\text{ite}(a, b, c)}$ which both denotes the root edge leading to a being complemented.

During evaluation of an assignment one has to keep track of how many complement edges were visited and the result negated, if the number of inverters was odd.

The other presented algorithms in principle still work the same way with BDDs with complement edges as well, given that the algorithms also keep track of negated edges. For humans the situation is more difficult. Given a graphical representation of a decision diagram with complement edges, it is much harder for humans to parse and reason about because it is difficult to keep track of all the negations and sharing possibilities.

The CUDD-library uses a BDD implementation with complement edges.

4 Rewriting Systems as a Boolean Formula

In this section we present a method for encoding rewriting systems as Boolean variables and properties of rewriting systems as Boolean formulas as well as implementation details.

4.1 Representing Rewriting Systems as a Boolean Formula

The basic idea is, to represent all rules $l \to r$ of a certain shape (fixed length of l and r with $|l| = u, |r| = v$, and a fixed alphabet) as all models of a Boolean formula of n variables. This formula is stored as a Binary Decision Diagram. For now we take the existence of an injective mapping between rules $l \to r$ and Boolean assignments of length n as a given (for possible mappings see Section 4.2).

$$\mu : \Sigma^{|l|} \times \Sigma^{|r|} \mapsto \{0,1\}^n \tag{7}$$

The operation μ is often referred to as `encode` while the inverse μ^{-1} is called `decode`.

When other formulas (predicates) over the same variables are constructed, that have the semantics of certain criteria that imply termination properties of rewriting systems, the predicates can be logically linked by an "and" or "and not" function, to get a Boolean function whose models can be interpreted as rewriting systems $l \to r$ that are or aren't members of certain classes of rewriting systems with certain termination properties (the systems fit into certain criteria).

To relate two predicates with the same semantics over different encodings of a rewriting system, we introduce the term of μ-compatibility. Two predicates P_i and P_i^μ are μ-compatible if and only if all models of P_i are, after mapping to the domain of Boolean variables, also models of P_i^μ (for implementations of μ-compatible predicates see Section 5.4ff).

$$\forall l \in \Sigma^u, r \in \Sigma^v : P_i(l,r) \iff P_i^\mu(\mu(l,r)) \tag{8}$$

Given a set of n Boolean variables (x_1 to x_n) that are used to encode all systems $l \to r$, a set of m predicates P that encode properties for systems $l \to r$, and the set of predicates P^μ that encode the same properties as the predicates in P but for systems encoded by the Boolean variables, a system $l \to r$ has the same properties independent of its encoding, given μ-compatible predicates from P and P^μ:

$$\begin{aligned}\forall l \in \Sigma^u, r \in \Sigma^v : P_1(l,r) \wedge P_2(l,r) \wedge .. \wedge P_m(l,r) \iff \\ P_1^\mu(\mu(l,r)) \wedge P_2^\mu(\mu(l,r)) \wedge .. \wedge P_m^\mu(\mu(l,r))\end{aligned} \tag{9}$$

Since Boolean formulas stored as BDDs can easily be combined on a formula level (see Section 3.3), the intersection of the set of all models of the μ-compatible predicates is equivalent to the set of models of the and-combined predicates:

$$P_1^\mu(x_1, x_2, .., x_n) \cap P_2^\mu(x_1, x_2, .., x_n) \cap .. \cap P_m^\mu(x_1, x_2, .., x_n) = (P_1^\mu \wedge P_2^\mu \wedge .. \wedge P_m^\mu)(x_1, x_2, .., x_n) \quad (10)$$

So instead of checking all $l \to r$ for all criteria one after another (see Kurth's Sieve, Section 2.2), the filter criteria are implemented as Boolean functions to work on a set of Boolean variables. Being stored as Binary Decision Diagrams, they then can be logically linked to get one function P^* with $P^* = (P_1^\mu \wedge P_2^\mu \wedge .. \wedge P_m^\mu)$ where all models of P^* can be interpreted as rewriting systems $l \to r$ with properties originally encoded by the predicates in P.

An implementation printing all systems with certain properties would at its simplest look like this:

$$\text{print}(\text{interpretSat}(\text{allSat}(\mathit{Predicate}_1 \wedge \mathit{Predicate}_2 \wedge .. \wedge \mathit{Predicate}_m))) \quad (11)$$

Where allSat returns a set of all models and interpretSat is an inverse of μ.

Note that if μ is not bijective, the predicates must not allow for any model that has no corresponding rule $l \to r$ in the argument domain. This can be achieved by adding a consistency predicate (see Section 5.1).

Comparing this approach to the traditional sieve we lose the ability to count which system was "thrown out" by which predicate. We only get the models of the main conjunction, which can be crafted to calculate specific numbers that the traditional sieve just generated along the way.

We call a triple $(|\Sigma|, |l|, |r|)$ the *shape* of the system $l \to r$ and we call a list of predicates that are included in the main conjunction (either in the positive or the negative, see Figure 13) the *configuration*.

4.2 Encoding of Letters and Systems

There are several possible binary encodings for a single letter[1], with a list of letters forming a word and a pair of words forming the rewriting rule.

standard binary The standard binary encoding is the encoding that is used in the binary number system with 00 meaning the first letter a of the alphabet $\{a, b, c, d\}$, 01

[1]The presented encoding schemes encode letters and encode words and the rule by extension. This is not strictly necessary.

meaning b, 10 meaning c and 11 meaning d. This encoding uses an amount of bits or BDD variables logarithmic to the alphabet size.
On the other hand, functions that semantically operate on a single letter of a word or rewriting rule are dependent on all variables instead of a subset.
This encoding scheme is also referred to as "full logarithmic encoding" [AM04].

one-hot The one-hot encoding uses as many bits or BDD variables for a letter as the size of the alphabet with all positions being false but the one that corresponds to a single letter, with 1000 meaning the letter a, 0100 meaning b, 0010 meaning c, and 0001 meaning d.
Functions that semantically operate on a single letter of a word can use a subset of variables to be defined. This is useful since many of the predicates defined in this chapter include something along the lines of $\exists a \in \Sigma$ or $\forall a \in \Sigma$ which make these functions being defined by small BDDs, in the sense that they include only few variables, useful.
One-hot encoding makes necessary that all paths within the BDD that lead to 1 include all the variables of the BDD, meaning that no two models of the BDD differ in only one but at least two bit.
This encoding scheme is sometimes referred to as "direct encoding" [SS11].

order The order encoding [PJ11] is similar to the one-hot encoding only that it doesn't use a single bit to signify a letter but a switch from 0 to 1 with 111 meaning the first letter a, 011 meaning b, 001 meaning c, and 000 meaning d.
The order encoding uses about as many variables/bits as the one-hot encoding while not having (aforementioned) single-letter-functions that do depend on all BDD variables.

In this work, the one-hot encoding was chosen. Whether the smaller variable footprint of the standard binary encoding outweighs the cost of the more complicated predicates warrants further research (see Section 9).

4.3 BDD Implementation and Haskell API

For implementation of this system we use the Haskell language [Has16]. The reason for using Haskell is that the prototype for this application has been implemented in Haskell using the functionally pure BDD library obdd [Wal16] that was lacking in speed and memory efficiency for this specific use case. The step to a different BDD implementation was small enough.

The CUDD library [Som15] was chosen because it is well established and has functional bindings [Wal15] to the Haskell language.

Having decided on the one-hot encoding, the following type declarations are derived to identify the components of a rewriting system.

```
data Side = L | R deriving ( Eq, Ord, Show )
newtype Pos = Pos Int deriving ( Eq, Ord, Show, Read, Enum, Num )
newtype Idx = Idx Int deriving ( Eq, Ord, Show, Read, Enum, Num )

type Var = (Side, Pos, Idx)
type Letter = [ Var ]
type Word = [ Letter ]
type Rule = (Word, Word)
```

Listing 3: type declarations to define a rewriting system

The functional CUDD bindings work using a `DDManager` that is the first argument of any operation and implicitly holds the state of the BDD base.

```
cuddInit :: DDManager
ithVar :: DDManager -> Int -> DDNode
```

Listing 4: CUDD types for initializing and variable selection

This is why the binary operations are redefined with the manager bound to them, to allow other operations in the same scope to be defined more easily.

```
readOne :: DDManager -> DDNode
readLogicZero :: DDManager -> DDNode
bAnd :: DDManager -> DDNode -> DDNode -> DDNode
bOr :: DDManager -> DDNode -> DDNode -> DDNode
bNand :: DDManager -> DDNode -> DDNode -> DDNode
bNor :: DDManager -> DDNode -> DDNode -> DDNode
bXor :: DDManager -> DDNode -> DDNode -> DDNode
bXnor :: DDManager -> DDNode -> DDNode -> DDNode
bNot :: DDManager -> DDNode -> DDNode
xEqY :: DDManager -> [DDNode] -> [DDNode] -> DDNode
true = readOne manager
false = readLogicZero manager
boolconst b = if b then true else false
not = bNot manager
(&&) = bAnd manager
(||) = bOr manager
(!&&) = bNand manager
(!||) = bNor manager
xor = bXor manager
(==) = bXnor manager
(-->) a b = (not a) || b
ite = bIte manager
or = foldr (||) false
```

```
and = foldr (&&) true
none = not . or
xEqY' = xEqY manager
```

Listing 5: Redefining binary operations

Any conflicting names from `Prelude` have been hidden and imported as `qualified P`.

The mapping `Var -> DDNode` (`get_node`) is done by generating all variables reading the system shape from the settings object `c`, ordering them using the order derived by Haskell and enumerating them which results in the mapping `Var -> Int`. This mapping has to be continuous since any "holes" in the mapping will implicitly introduce unused variables.

```
sigma = Idx ( C.sigma_size c )
sl = Pos $ C.lhs_length c
sr = Pos $ C.rhs_length c
vars :: S.Set Var
vars = S.fromList $ do
                      (side,top) <- [(L,sl) ,(R,sr)]
                      pos <- [0 :: Pos .. top-1]
                      idx <- [ 0 ..sigma-1]
                      return (side,pos,idx)

var_to_int vars elem = fromJust $ elemIndex elem (S.toAscList vars)
var_to_int' = var_to_int vars
get_node :: Var -> DDNode
ithVar' = ithVar manager
get_node = ithVar' . var_to_int'

-- Reverse operation
int_to_var = (!!) . S.toAscList
int_to_var' = int_to_var vars
```

Listing 6: Generating variables and mapping them to DDNodes

This mapping is done once to retrieve the variable tables (`[[DDNode]]`) for the left-hand and right-hand side that are passed to the predicates (see Section 5).

```
make_letter side pos sigma = for [0 .. sigma-1]$ \ k -> (side,pos,k)
l = for [ 0 .. sl - 1 ] $ \ i -> map get_node $ make_letter L i sigma
r = for [ 0 .. sr - 1 ] $ \ i -> map get_node $ make_letter R i sigma
```

Listing 7: generating variable tables

The main conjunction `g` is created by conjoining the predicate functions that have been passed the variables table. If a predicate is not to be included it is replaced by `true` or negated if the negation is to be included.

```
condition name semantics =
  case name c of
    Nothing -> true
    Just polarity ->
      ( case polarity of True -> id ; False -> not )
      $ semantics (l,r)
g = and
  [ is_rule (l,r)
  , condition C.bordered bordered
  , condition C.whole_alphabet (whole_alphabet sigma)
  , condition C.inhibitor (inhibitor sigma)
  , condition C.one_loop one_loop
  , ...
  ]
```

Listing 8: generating variable tables

Once the main conjunction `g` is constructed, we need to read its models. The number of models can easily be read with `countPathsToNonZero :: DDNode -> Double`. The models themselves can be read via the function `allSat :: DDManager -> DDNode -> [[SatBit]]` but `allSat` reserves memory for all models at once instead of lazily generating them, which is one of the features of BDDs (see Section 3.4). To do this, the imperative interface of the bindings has to be used with the functions `firstCube` and `nextCube` for the first solution and every next solution. This has to be done within the ST monad.

```
firstCube :: DDManager s u -> DDNode s u
          -> ST s (Maybe ([SatBit], DDGen s u Cube)) Source
nextCube :: DDManager s u -> DDGen s u Cube -> ST s (Maybe [SatBit])
```

Listing 9: Functions for iterating BDD solutions

The number of systems we want to generate is stored in `cnt` and if it is negative then all solutions are generated.

```
number_of_models = truncate . countPathsToNonZero
unless (number_of_models g > 0 P.&& C.results c /= 0) exitSuccess
let imngr = toImperativeManager manager
(s,c') <- stToIO $ do
    g' <- toImperativeNode g
    first <- I.firstCube imngr g'
    return $ fromJust first
let go cnt = unless (cnt P.== 0) $ do -- negative numbers create infinite
                                      -- loop leading to full consumption
    s <- stToIO $ I.nextCube imngr c'
```

```
    case s of
      Just s' -> do { print_solution s'; go $ cnt - 1}
      Nothing -> return ()
do { print_solution s; go $ C.results c - 1 }
```

Listing 10: Iterating BDD solutions

Note that this listing excludes calling of external solvers.

The last part is how to interpret and print a model of the binary function `g`. First the solution (`[SatBit]`) needs to be converted to `[[Bool]]` because we're working with cubes and one cube might represent multiple solutions if a variable of the BDD isn't included in the fulfilling assignment. For the one-hot encoding every cube represents one and only one model so every `[SatBit]` can be converted to a `[Bool]` and it is then enumerated and mapped to its respective variable. Finally this mapping is interpreted using the alphabet `[ 'a' .. ]` and printed to `stdout` from where it can be processed further.

```
expand :: SatBit -> [Bool]
get_solution :: [SatBit] -> [(Var, Bool)]
get_solution = zipWith (\idx b -> (int_to_var' idx, (head . expand) b)) [0..]
decode sigma sl sr m =
  let decode_letter side pos =
        let [a] = do
              (a,idx) <- zip [ 'a' .. ] [ 0 .. sigma-1 ]
              guard $ mM.! (side,pos,idx)
              return a
        in  a
      decode_word side len =
        map (decode_letter side) [ 0 .. len-1 ]
  in  ( decode_word L sl, decode_word R sr )
decode_solution = (decode sigma sl sr) . M.fromList . get_solution
print_solution = print . decode_solution
```

Listing 11: Interpreting BDD solutions

Additionally we introduce some type aliases for the encoded variants of the letters, words, and rules, to have some more readable type information.

```
  type EncLetter = [ DDNode ]
  type EncWord = [ EncLetter ]
  type EncRule = (EncWord, EncWord)
  type EncBool = DDNode
```

Listing 12: Type aliases for encoded variants

	l_0	l_1	$\rightarrow$	r_0	r_1	r_2
a	1	0		1	0	0
b	0	0		0	1	0
c	0	1		0	0	1

Figure 8: Example Encoding $ac \rightarrow abc$ with $\Sigma = \{a, b, c\}$

4.4 Selecting Bits, Letters and Words

A useful interpretation for building formulas with the one-hot encoding is the letter position being the columns of a table and the letters being the rows. Figure 8 shows an example encoding for the system $ac \rightarrow abc$ with $\Sigma = \{a, b, c\}$.

This idea fits nicely with using matrix indexing notation to select for certain subsets of bits or nodes in the BDD.

The following is defined, extending the well-known matrix indexing notation and the indexing definition from section 2.1 naturally:

- $w_{1,b}$ to mean the bit for the second letter being a b. Implementation see Listing 13.
- $w_{1,a..c}$ to mean the bits for second letter being a to c.
- $w_{1,b..}$ to mean the bits for second letter being b up to the last letter of Σ.
- $w_{1,..c}$ to mean conversely the bits for the second letter being up to c.
- $w_{1,\{b,c..\}}$ to mean the bits for second letter being b, then c up to the last letter of Σ.
- $w_{1,\Sigma}$ to mean all the bits for the second letter. Implementation is `(!!)`.
- w_1 to mean all the bits for the second letter as well. This notation, being shorter, is preferred.
- $w_{1..3}$ to mean all the bits for the second to fourth letter in order. Implementation see Listing 14.
- $w_{0..|w|,a}$ to mean all the bits for the letter a in order
- $w_{*,a}$ to mean all the bits for the letter a in order as well. This is essentially a bit-mask of a single letter being the letter in the actual word. Implementation see Listing 15.
- $w_{0..|w|,\Sigma}$, $w_{0..|w|}$, $w_{*,\Sigma}$ and w_* would then be the same as w. The latter is preferred to the longer alternatives.

All other implementations follow from the combination of the base implementations and the Listings 13, 14, and 15.

```
single_bit :: EncWord -> Int -> Idx -> EncBool
single_bit w a (Idx idx) = (w !! a) !! idx
```

Listing 13: single_bit defining $w_{a,idx}$

```
partial_word :: EncWord -> Int -> Int -> EncWord
partial_word w a b = take (b - a + 1) $ drop a w
```

Listing 14: partial_word defining $w_{a..b}$

For the selection of letters, distributivity with regards to concatenation is assumed so that

$$xy_{*,a} = x_{*,a}y_{*,a} \tag{12}$$

We say that the operator for selection of positions binds less strongly than concatenation:

$$xy_{m..n} = (xy)_{m..n} \tag{13}$$

In the following sections, μ-compatible functions and predicates for the given μ (one-hot) are defined. When the predicates are defined they are annotated with μ while, for readability reasons, this is omitted when the functions and predicates are used within the same or other definitions. When a μ-compatible function or predicate is defined, all auxiliary predicates and functions must also be of a μ-compatible nature.

```
extract_letter_bits :: EncWord -> Idx -> [EncBool]
extract_letter_bits w (Idx idx) = map (!! idx) w
```

Listing 15: extract_letter_bits defining $w_{*,idx}$

5 Encoding and Basic Predicates

In this section we present basic predicates that encode properties of encoded letters, words, and systems.

5.1 Correct Encoding (Consistency)

The one-hot encoding or direct encoding is the chosen encoding scheme. It encodes one letter of an alphabet of size $|\Sigma| = s$ with s bits, where the nth bit being true means that this letter is the nth letter of the alphabet Σ, while all other bits are false. Any two (or more) of the n bits being true, or none being true, is not considered a valid encoding of a letter of the alphabet.

The function of a single letter being one_hot encoded is defined as

$$\text{one\_hot}(l) \iff \exists a \in \Sigma : (\forall b \in \Sigma \setminus \{a\} : a \wedge \overline{b}) \tag{14}$$

Within the framework of binary functions, this can be recursively defined without quantifying over the alphabet as follows:

$$\text{one\_hot}^{\mu}(l_{\{a,b..\}}) = \begin{cases} 0 & |l| = 0 \\ (l_a \wedge (\bigvee l_{b..})) \vee (\overline{l_a} \wedge \text{one\_hot}(l_{b..})) & \text{otherwise} \end{cases} \tag{15}$$

Using the ite-operator, the following implementation is derived:

```
one_hot :: EncLetter -> EncBool
one_hot a = case a of
              [] -> false
              l:ls -> ite l (none ls) (one_hot ls)
```

Listing 16: one_hot predicate implementation

This looks quadratic since there's a fold (`none`) that is always applied to the tail of the list, but for all but the first position every expression in that fold has already been evaluated before and is part of the BDDs cache and the BDD base so that we can expect linear run time.

The whole system being correctly encoded is the application of the predicate over all letters of the right-hand side and left-hand side of the rule and its conjunction. This predicate is called is_rule because it being true means, that the bits of the function actually encode a single-rule string-rewriting system. This is the **consistency** criterion.

```
is_rule :: EncRule -> EncBool
is_rule (l, r) = and $ map one_hot (l ++ r)
```

Listing 17: is_rule predicate implementation

5.2 Preset Positions

It might be interesting to look for systems with certain properties with regards to some predicate as a function of the rule and the alphabet but also with regards to some patterns on a content-basis. The user might want to look for rules whose left-hand side starts with an a or confirm that any right-hand side that is canonical with regards to renaming does not start with a b.

As a user-interface to preset positions we use glob wildcard expansion [Glo].

Glob supports the following matching constructs:

- **?** matches any single character.
- ***** matches any string, including the empty string.
- **[abc]** matches any of the letters a, b or c.
- **[a-c]** matches any of the letters a to c.
- **[!abc]** matches any letter that is not a, b or c. This construct was not implemented.

Glob patterns are a strict subset of regular languages. Most glob libraries are specifically implemented to support file-name expansion, as it is commonly used in UNIX. Using parser-combinators it is no problem implementing a parser for glob-patterns.

```
import Text.ParserCombinators.ReadP

data Glob = GOne | GMany | GChar [Idx] deriving ( Eq, Show )

charToIdx c = Idx $ ord c - ord 'a'
gOne = do { char '?'; return GOne }
gMany = do { char '*'; return GMany }
gChar = choice (map char ['a'..'z'])
gAtom = do { c <- gChar; return $ GChar $ map charToIdx [c] }
gAtoms= do { char '['; cs <- many1 (gChar); char ']';
                    return $ GChar $ map charToIdx cs }
gRange= do { char '['; f <- gChar; char '-'; t <- gChar; char ']';
                              return $ GChar $ map charToIdx [f..t] }
gGlob = do { optional $ char '"';
          g <- many1 $ choice [gOne, gMany, gAtom, gAtoms, gRange];
                                        optional $ char '"'; return g }
parse_glob = head . map fst . filter (null.snd) . readP_to_S gGlob
```

Listing 18: parser for glob-patterns using Text.ParserCombinators

We can easily construct a BDD that has all the systems that fit a specific glob-pattern as models:

```
alphabet = [ 0 ..sigma-1 ]
match_glob :: [[DDNode]] -> [Glob] -> DDNode
match_glob w g = case (w, g) of
    ([],[]) -> true
    -- GOne | GMany | GChar [Int]
    (l:ls, GOne:gs) -> match_glob ls gs
    (_, [GMany]) -> true
    (l:ls, GMany:gs) -> match_glob ls g || match_glob w gs
    (l:ls, GChar idxs:gs) -> let unset_idxs = alphabet \\ idxs
                                 set_idxs = alphabet `intersect` idxs
                -- ignoring letters that are larger than the alphabet
                                 to_letters = map (\ (Idx i) -> l !! i )
                                 set_letters = to_letters set_idxs
                                 unset_letters = to_letters unset_idxs
                -- we also want to deduce from zeroes
                            in or set_letters && none unset_letters
                                              && match_glob ls gs
    _ -> false
```

Listing 19: constructing a BDD from glob-patters

By adding `match_glob r $ parse_glob $ C.globright c` and `match_glob l $ parse_glob $ C.globleft c` to the main conjunction, we can use glob-patterns to further refine the sieve.

Glob-patterns can also be used to reduce the number of variables in the BDD (see Section 7.5).

5.3 Whole Alphabet

Additionally, it needs to be encoded that the rule $l \to r$ actually uses the whole alphabet. If the user is specifically looking for systems with an alphabet of size n, the rule should indeed use all n different letters of the alphabet and not have some letter that neither occurs in the left-hand side nor in the right-hand side of the rule.

$$\text{whole\_alphabet}(l, r) \iff \forall a \in \Sigma : |l|_a + |r|_a > 0 \tag{16}$$

Within the framework of binary functions, this can be defined as follows:

$$\text{whole\_alphabet}^{\mu}(l, r) = \forall a \in \Sigma : \bigvee lr_{*,a} \tag{17}$$

From which the following implementation is derived:

```
whole_alphabet :: Idx -> EncRule -> EncBool
whole_alphabet (Idx sigma) (l, r) =
            and $ for [0..(sigma - 1)] $ \idx ->
            or (extract_letter_bits (l ++ r) (Idx idx))
```

Listing 20: whole_alphabet predicate implementation

If this predicate is not used, the preset alphabet size (by definition of the shape of the system, see Section 4.1) is the upper bound for the number of different letters in the rule $l \to r$. If it is used in the positive, this number is also the lower bound.

5.4 Helper Predicates

5.4.1 Letter Relationships

A construction of simple predicates that describe relationships on encoded letters is taken as the basis for building more complex predicates.

Equals for letters: *Equals* for letters on an encoded letter is defined as

$$u_x =_l^\mu v_y \iff \forall a \in \Sigma : u_{x,a} = v_{y,a} \tag{18}$$

Considering that two letters, due to the one-hot encoding, differ in none or exactly two bits, one (but the same) bit in each letter can be ignored in comparing those letters. From that, following implementation for equals for letters is derived.

```
l_eq :: EncLetter -> EncLetter -> EncBool
l_eq u v = xEqY' (drop 1 u) (drop 1 v)
```

Listing 21: l_eq (equals for letter) predicate implementation

Less-than for letters: *Lexicographically less than* for encoded letters is defined as

$$u_x <_l v_y \iff \exists a, b \in \Sigma : u_{x,a} \wedge v_{y,a} \wedge (a < b) \tag{19}$$

without existentially quantifying over the alphabet and referring to its intrinsic ordering, this recursive definition is actually implemented

$$u_{x,\{a,b..\}} <_l^\mu v_{y,\{a,b..\}} = \begin{cases} 0 & |u| = |v| = 0 \\ (u_{x,a} \wedge \bigvee v_{y,b..}) \vee (u_{x,b..} <_l v_{y,b..}) & \text{otherwise} \end{cases} \tag{20}$$

with the following implementation:

```
l_lt :: EncLetter -> EncLetter -> EncBool
l_lt u v = case (u, v) of
    ([], []) -> false
    (a:as, b:bs) -> (a && or bs) || l_lt as bs
```

Listing 22: l_lt (less-than for letter) predicate implementation

Less-or-equal for letters: *Lexicographically less-or-equal* can be implemented by the disjunction of less-than and equals for letters but the construction might take longer than a new implementation, that is similar to less-than for letters:

$$u_{x,\{a,b..\}} \leq_l^{\mu} v_{y,\{a,b..\}} = \begin{cases} 0 & |u| = |v| = 0 \\ (u_{x,a} \wedge \bigvee v_{y,\{a,b..\}}) \vee (u_{x,b..} \leq_l v_{y,b..}) & \text{otherwise} \end{cases} \tag{21}$$

with the following implementation:

```
l_le :: EncLetter -> EncLetter -> EncBool
l_le u v = case (u, v) of
    ([], []) -> false
    (a:as, b:bs) -> (a && or (b:bs)) || l_le as bs
```

Listing 23: l_le (less-or-equal for letter) predicate implementation

5.4.2 Word Relationships

From the letter relationships, word relationships lexicographically less-or-equals and lexicographically less-than for words of equal lengths are derived in the natural way.

$$s \leq_w^{\mu} t = \begin{cases} 1 & |s| = |t| = 0 \\ s_0 <_l t_0 \vee (a =_l b \wedge s_{1..} \leq_w t_{1..}) & \text{otherwise} \end{cases} \tag{22}$$

$$s <_w^{\mu} t = \begin{cases} 0 & |s| = |t| = 0 \\ s_0 <_l t_0 \vee (a =_l b \wedge s_{1..} <_w t_{1..}) & \text{otherwise} \end{cases} \tag{23}$$

With the corresponding implementations

```
w_le :: EncWord -> EncWord -> EncBool
w_le s t = case (s, t) of
          ([],[]) -> true
          (a:as,b:bs) -> (l_eq a b && w_le as bs) || l_lt a b
```

Listing 24: w_le (less-or-equal for words) predicate implementation

```
w_lt :: EncWord -> EncWord -> EncBool
w_lt s t = case (s, t) of
             ([],[]) -> false
             (a:as,b:bs) -> (l_eq a b && w_lt as bs) || l_lt a b
```

Listing 25: w_lt (less-than for words) predicate implementation

Next the prefix-relation between words, where s is a prefix of t is defined as:

$$\text{pre}^{\mu}(s,t) = |s| \leq |t| \wedge \forall i \in \{0..|s|\} : s_i =_l t_i \tag{24}$$

with the implementation

```
isPrefixOf :: EncWord -> EncWord -> EncBool
isPrefixOf s t = if length s > length t
                 then false
                 else and $ zipWith l_eq s t
```

Listing 26: isPrefixOf predicate implementation

Using the prefix-relation, the suffix-relation and the equals relation between words is easily defined as

$$\text{suf}^{\mu}(s,t) = \text{pre}(\widetilde{s},\widetilde{t}) \tag{25}$$

$$(s =_w^{\mu} t) = (|s| = |t| \wedge \text{pre}(s,t)) \tag{26}$$

where $\widetilde{s}$ is the reverse of s. The implementations are as follows

```
isSuffixOf :: EncWord -> EncWord -> EncBool
isSuffixOf s t = (reverse s) `isPrefixOf` (reverse t)
```

Listing 27: isSuffixOf predicate implementation

```
w_eq :: EncWord -> EncWord -> EncBool
w_eq s t = if length s P.== length t
           then s `isPrefixOf` t
           else false
```

Listing 28: w_eq (equals for words) predicate implementation

Using these helper predicates, more complicated predicates, which correspond to attributes of rewriting systems and applicable decision procedures, are derived in the following sections.

6 Predicates of Criteria

In this section we present predicates and their implementations, that encode certain criteria for termination, non-termination or the existence of a decision procedure for termination for rewriting systems that fit that criterion.

6.1 Morphisms and Embeddings

There are morphisms between rewriting systems with regards to termination. Recall from Section 2.1 that we write $\mathsf{SN}(l, r)$ iff $l \to r$ terminates. A mapping or embedding h from rewriting system to rewriting system respects termination if $\mathsf{SN}(l, r) \iff \mathsf{SN}(h(l, r))$. For example, reversing both the right and left-hand side of a rule does not affect its termination behaviour.

Two systems that are equivalent with regards to some morphism have the same termination behaviour and have, modulo the morphism, the same termination or non-termination argument. Having a proof for termination or non-termination for either system automatically results in a proof for all morphic or embedding/embedded systems.

For a set H of morphisms or mappings (to be discussed in the following sections), we are only interested in one canonical representative of a system with regards to that set of mappings. The canonical representative is called the H-canonical representative and we define the H-canonical representative of some system $l \to r$ to be $\min(\{h(l, r) | h \in H\})$, the smallest system with regards to some order.

A system $l \to r$ is considered smaller than a morphic or embedding system $l' \to r'$ if r is shorter than r' in length or if they have the same length but rl is lexicographically smaller than $r'l'$.

In order to only generate systems "once" and check termination arguments "once", we only want to generate H-canonical representatives. The actual mapping functions h are not necessary to check for canonicity. In the following sections we discuss predicates that are necessary or sufficient (or both) for (non-)canonicity with regards to some morphism or embedding h for a single $l \to r$ instead of actually checking $h(l, r) = l \to r$, which would also mean that $l \to r$ is canonical with regards to h. A canonicity predicate being false for a system has the semantics of that system not being the canonical representative and since we consider all systems of some shape, that system can be omitted from the enumeration while the canonical representative of the omitted system will not be or has not been omitted on canonicity grounds. Since we enumerate systems of ever greater length, all systems with a shorter length have already been considered or have a trivial termination problem.

6.1.1 Reversal

Two systems $l \to r$ and $l' \to r'$ share the same termination behaviour if both sides of the rule are reversed. [Ges02, Chapter 2.1]

$$\mathsf{SN}(l, r) \iff \mathsf{SN}(\tilde{l}, \tilde{r}) \tag{27}$$

The system $ba \to aab$ terminates iff the system $ab \to baa$ terminates. The system $ba \to aab$ is canonical with regards to reversal because it has the lexicographically smaller rl ($aabba \leq baaab$).

This is easily defined within the framework of allowed binary functions and predicates.

$$C^{\mu}_{\text{rev}}(l, r) = rl \leq_w \tilde{r}\tilde{l} \tag{28}$$

With the following implementation:

```
canonical_reverse :: EncRule -> EncBool
canonical_reverse (l,r) = w_le (r++l) (r'++l')
                          where l' = reverse l
                                r' = reverse r
```

Listing 29: canonicity after reversal implementation

6.1.2 Renaming

The systems $ab \to baa$ and $ba \to abb$ are only distinguished by the choice of letters from the alphabet. With regards to termination, the systems behave the same.

Let $l \to r$ be a rule over alphabet Σ, and let $\phi : \Sigma \to \Omega$ where Ω is some alphabet.

Since we only have a finite alphabet (by definition of the shape of systems) Σ we only need to consider all permutations π of Σ ($\pi : \Sigma \to \Sigma$).

This mapping can be extended via pointwise application to a mapping from Σ^* to Σ^*.

$$\pi : \Sigma^* \to \Sigma^* \tag{29}$$

$$\pi(st) = \pi(s)\pi(t) \tag{30}$$

The two rules $l \to r$ and $\pi(l) \to \pi(r)$ over the alphabet Σ have the same termination behaviour [Ges02, Chapter 2.1].

$$\mathsf{SN}(l, r) \iff \mathsf{SN}(\pi(l), \pi(r)) \tag{31}$$

We take a right hand side to be canonical (C_π) when there is no renaming of r that is lexicographically smaller than r.

$$C_\pi(l, r) = \forall \pi'(r \leq_w \pi'(r)) \tag{32}$$

Since we expect r to be larger than the alphabet Σ and the whole alphabet to be used in r, we only check the predicate for r instead of rl.

Within the area of μ-compatible functions, we define a predicate that does not quantify over all possible mappings. Instead we define a predicate with equivalent semantics.

The first letter that occurs in r has to be the first letter of Σ, otherwise there would be some π with $(\Sigma_0 \to r_0) \in \pi$. The second letter now is either a or b. The third letter is one of a or b if the second letter was a or one of a, b, or c if the second letter was b. In essence, we track the largest letter yet (progress p) and any next letter is only allowed to be smaller or equal than the largest letter yet or the single next-largest letter.

$$\mathrm{crn}^\mu(p, w) = \begin{cases} 1 & |w| = 0 \\ (\vee w_{0,..p} \wedge \mathrm{crn}(p, w_{1..})) \vee (w_{0,p+1} \wedge \mathrm{crn}(p+1, w_{1..})) & \text{otherwise} \end{cases} \tag{33}$$

For readability reasons, the range-check that the argument p is never larger than $|\Sigma|$ omitted.

From this we derive the following implementation with the included optimization, that for alphabets of size $|\Sigma| = 2$ the first letter has to be an a and after that, no additional conditions apply.

```
crn :: Idx -> EncWord -> EncBool
crn (Idx prog) w = case w of
    [] -> true
    l:ls -> (or (take prog l) && crn (Idx prog) ls) ||
            ((l !! min prog (length l - 1)) && crn (Idx (prog+1)) ls)
canonical_renaming :: Idx -> EncRule -> EncBool
canonical_renaming (Idx sigma) (_,r) = if sigma > 2
                                       then crn (Idx 0) r
                                       else head (head r)
```

Listing 30: canonical after renaming implementation

6.1.3 Reversal and Renaming

By combining reversal and renaming we obtain a third morphism between single-rule rewriting-systems. The system $ba \to abb$ is canonical after renaming ($abb \leq_{\text{lex}} baa$) and it is also canonical after reversal ($abbba \leq_{\text{lex}} bbaab$) but it is not canonical after renaming

and reversal in that after the system is reversed, the smallest renaming of the reversed system is smaller than the original system. The renaming of $ab \to bba$ to $ba \to aab$ is smaller than the original system $ba \to abb$. All those systems have the same termination behaviour.

$$\mathsf{SN}(l, r) \iff \mathsf{SN}(\pi(\tilde{l}), \pi(\tilde{r})) \tag{34}$$

We define canonicity after reversal and renaming $C_{\pi rev}$ to be the least system within this set of homomorphisms.

$$C^{\mu}_{\pi\mathrm{rev}}(l, r) = \forall \pi : rl \leq_w \pi(\tilde{r})\pi(\tilde{l}) \tag{35}$$

From this we derive the following implementation that indeed quantifies over all permutations to represent all possible renamings.

```
canonical_reverse_renaming :: EncRule -> EncBool
canonical_reverse_renaming (l, r) = none $ map (`w_lt` (r ++ l)) variants'
  where variants' = map transpose $ permutations $
                    transpose $ reverse r ++ reverse l
```

Listing 31: canonical after reversal and renaming implementation

The simple case of canonicity after reversal without renaming can be ignored if this predicate is used in the negative since there is a renaming π that presents as no renaming (the identity permutation).

6.1.4 Coding

The systems $bba \to aabb$ and $eecd \to cdcdee$ (and $bbac \to acacbb$ by renaming) have the same termination behaviour because any reduction step in one system can be mapped to a reduction step in the other one by applying the mapping $\{a \to cd, b \to e\}$. The set $\{cd, e\}$ (or $\{ac, b\}$ for the renamed system) is a *code*.

A set $C \subseteq \Sigma^+$ is a code if each $w \in C^*$ has a unique decomposition as a product of words from C.

For $C = \{c_1, \ldots, c_n\}$, consider the rewriting system $\overline{C} = \{c_i \to i\}$ over $\overline{\Sigma} = \Sigma \cup \{1, \ldots, n\}$. We call C *non-overlapping* if $\overline{C}$ has no critical pairs i.e. no overlaps between code words, and no self-overlap.

$\{a, bc\}$ is a non-overlapping code. $\{ab, bc\}$ is a code but it is overlapping (abc is a critical pair). Note: $\{ab, bc\}$ is comma-free.

For a non-overlapping code C, the reduction to $\overline{C}$-normal form defines a mapping $h_C : \Sigma^* \to \overline{\Sigma}^*$. E.g., for $C = \{a, bc\}$ we have $h_C(babcc) = b[a][bc]c$. Such a mapping is extended to rewriting systems by $h_C(R) = \{(h_C(l), h_C(r) \mid (l, r) \in R\}$.

For a non-overlapping code C, and SRS $l \to r$ with $l \subseteq C^*$ and $r \subseteq C^*$, we have that forall $u, v \in \Sigma^* : u \to v \iff h_C(u) \to h_C(v)$. The systems R and $h_C(R)$ have the same termination behaviour $\mathsf{SN}(R) \iff \mathsf{SN}(h_C(R))$ and if $|h_C(r)| < |r|$, then we have reduced the termination problem of $l \to r$ and R is not canonical after coding.

While this is in general hard to formulate within the realm of μ-compatible functions, the following simple instance of the general scheme is already useful: Let Σ consist of distinct letters $a_1, \ldots, a_n$. Then take the code $\{a_1a_2, a_3, \ldots, a_n\}$. Only one code word has length two, all others have length one. Such a code is admissible for a given rewrite system (l, r) if each occurrence of a_1 in l and in r has an occurrence of a_2 immediately to the right of it, and vice versa.

$bca \to aabc$ is reduced to $[bc]a \to aa[bc]$ via the code $\{a, bc\}$, where $[bc]$ is treated as a single letter. We call a code n-code if any member of the code is of length n. The system $bca \to aabc$ is not canonical after 2-coding because there is a two-letter word in the code.

We define the predicate 2-coding canonicity within the framework of μ-compatible functions as follows:

$$C^{\mu}_{c2}(l, r) = \exists a, b \in \Sigma : ((\forall p \in \{0, \ldots, |l| - 1\} : l_p = a \iff l_{p+1} = b) \wedge (\forall p \in \{0, \ldots, |r| - 1\} : r_p = a \iff r_{p+1} = b)) \quad (36)$$

The range check is omitted for readability. It is implicitly included in the following implementation that takes the bitmaps representing the occurrence of two letters and compares them:

```
hom' :: [EncBool] -> [EncBool] -> EncBool
hom' l1 l2 = and [
               not (head l1), not (last l2),
               and $ zipWith (==) (tail l1) (init l2)
               ]
hom :: Idx -> EncRule -> EncBool
hom (Idx sigma) (l, r) =
  or $
  for [(x,y)|x <- [0 .. sigma - 1], y <- [0 .. sigma - 1], x /= y] $
    \ (a, b) -> (
      hom' (extract_letter_bits l (Idx a)) (extract_letter_bits l (Idx b))
      &&
      hom' (extract_letter_bits r (Idx a)) (extract_letter_bits r (Idx b))
  )
can_coding2 :: Idx -> EncRule -> EncBool
can_coding2 s = not . (hom s)
```

Listing 32: 2-coding implementation

As stated above, this predicate only encodes the specific case of all elements of the code being made up of distinct letters and is only a sufficient condition for 2-coding but not necessary. Consider the 2-code $\{a, bc, dc\}$ that is a non-overlapping code but is not considered a 2-coding by the given implementation.

Even though we are only considering specific instances of 2-codings, larger coding schemes might as well include occurrences of 2-coding and fit this example. Consider the system $abcd \to ddabc$ with the 3-code $\{abc, d\}$. The 2-code $\{ab, c, d\}$ is also a code (as is $\{a, bc, d\}$).

If a system $l \to r$ has no overlaps in that $\mathrm{OVL}(l, r) \cup \mathrm{OVL}(r, l) = \emptyset$, the set $\{l, r\}$ is a code can be used to map the system $l \to r$ to the obviously terminating system $a \to b$ which fits nicely with the termination argument of Kurth's Criterion D (see Section 6.2.3).

We also note that every overlap of l and r with the coding C is also uniquely decomposable into words in C. Let o be an overlap between l and r and l' and r' the prefix and suffix respectively that completes the word so that $l = l'o$ and $r = or'$. Say o does not have a unique decomposition using the non-overlapping code C then l' and r' do not have one either. Let s be the shortest suffix of r and t the shortest prefix of o so that $st \in C$. Since t is the prefix of l and $st \in C$, C can not be non-overlapping. Therefore any overlap between l and r has a unique decomposition in C^+ if l and r have.

6.1.5 Bordered

A string w is *bordered* by z if there is a non-empty z that is both a prefix and a suffix of w. The rule $l \to r$ is bordered if l and r are bordered by the same z [Ges02]. The smallest such z is called the "border".

As an example, $ababa \to aaabbba$ is a bordered system with the border a.

Only hinting at the decision procedure outlined by Geser [Ges02, Theorem 6.27], a system that is bordered has, after some decomposition steps, an embedded smaller system that terminates if and only if the original system terminates. The length of a loop in the original system is bounded by the length of the loop in the embedded system. If the termination of the embedded system is considered, the termination of the original system need not be considered. Since the embedded system is smaller than the original system, it is assumed that is termination is either trivial or has already been considered.

Within the framework of Boolean functions this can be defined as:

$$\begin{aligned}\mathrm{bordered}^{\mu}(l, r) = \exists p \in \{1..\left\lfloor \frac{|l|}{2} \right\rfloor\} : \\ l_{0..p} =_w l_{(|l|-p)..|l|-1} \wedge l_{(|l|-p)..|l|-1} =_w r_{0..p} \wedge r_{0..p} =_w r_{(|r|-p)..|r|-1}\end{aligned} \quad (37)$$

We only need to consider the cases where the border is no longer than half the size of the left-hand side of the rule. If it were longer than half the size it would have a self-overlap.

The shortest border does not have a self-overlap, because the overlap would then be a border that is shorter.

The following code for the bordered predicate is derived:

```
bordered :: EncRule -> EncBool
bordered (l, r) = or $ for [ 1 .. (length l `quot` 2) ] $ \ w ->
                  let l' = take w l
                      l''= reverse $ take w $ reverse l
                      r' = take w r
                      r''= reverse $ take w $ reverse r
                  in and [ r'  `w_eq` r'' ,
                           r'' `w_eq` l'  ,
                           l'  `w_eq` l'' ]
```

Listing 33: bordered predicate implementation

6.2 Termination Arguments

6.2.1 One-Loop/Factor

A rewriting system $l \to r$ has a loop of length 1 or one-loop if there is a word w from which a word awb can be reached with a single application of the rewriting rule. The system $l \to r$ is not terminating.

$$\text{one\_loop}(l, r) \iff (\exists w, a, b \in \Sigma^* : w \to_{l \to r} awb) \tag{38}$$

An example system that fulfills this predicate is $ab \to abb$.

The word w that leads to the rewriting loop shall be the shortest word that does so. Any word shorter than w leads to terminating behaviour while any word u larger than w can be split into $u = vwx$ with the behaviour $vwx \to_{l \to r} vawbx$.

The smallest possible w that leads to non-terminating behaviour is l. So for the case that $w = l$

$$\text{one\_loop}(l, r) \iff (\exists a, b \in \Sigma^* : l \to_{l \to r} alb) \tag{39}$$

And by application of the rule it follows that

$$\text{one\_loop}(l, r) \iff (\exists a, b \in \Sigma^* : r = alb) \tag{40}$$

The system $l \to r$ has a loop of length 1 if the left-hand side of the rule is a factor of the right-hand side.

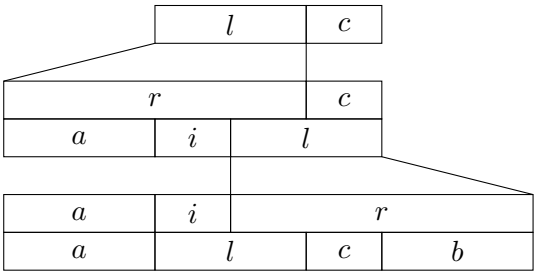


Figure 9: Visualization of two-looping reduction

Within the framework of Boolean function this can be defined as:

$$\text{one\_loop}^{\mu}(l, r) = \exists d \in \{0 \ldots (|r| - |l|)\} : l =_w r_{d..d+|l|-1} \tag{41}$$

From this the following code for the one_loop predicate is derived:

```
one_loop :: EncRule -> EncBool
one_loop (l, r) = or $ for [0..(length r - length l)] $
                    \d -> w_eq l $ take (length l) (drop d r)
```

Listing 34: one_loop predicate implementation

6.2.2 Two-Loop

A rewriting system $l \to r$ has a loop of length 2 or two-loop if there is a word w from which a word awb can be reached with two applications of the rewriting rule. The system $l \to r$ is not terminating. We call the words a and b *reproduced*.

$$\text{two\_loop}(l, r) \iff (\exists w, a, b \in \Sigma^* : w \to^2_{l \to r} awb) \tag{42}$$

One system that has this property is the system $ba \to aabb$ that has a two-loop starting from the word baa ($\underline{ba}a \to aab\underline{ba} \to aa\mathbf{ba}bb$).

The word w that leads to the rewriting loop shall be the shortest word that does so. Any word shorter than w leads to terminating behaviour while any word u larger than w can be split into $u = vwx$ with the behaviour $vwx \to^2_{l \to r} vawbx$.

In this section we develop a predicate that encodes the property of rewriting systems $l \to r$ having a loop of length two.

To even have one rewriting step for $l \to r$ the word w must be either of the form lc or cl, to say, the left-hand side of the rule with some smallest outer context c that leads to the infinite reduction.

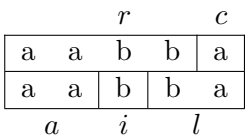


Figure 10: $rc = ail$ for $ba \rightarrow aab$

The lc-case: The starting word lc leads, with one application, to rc that matches the word ail where a is the left reproduced word, i is some inner context, and l is the left-hand side of the rule to continue application of the rule. $alcb$ must match air to be reached with one step from ail (see Figure 9).

$$lc \rightarrow rc = ail$$
$$ail \rightarrow air = a\mathbf{lc}b$$

For two_loop to be true one has to find some c and i that fulfill the following conditions (see Figures 10 and 11):

1. c is a suffix of l, because of $rc = ail$
2. i is a prefix of l, because of $air = alcb$ ($ir = lcb$)
3. il is the suffix of rc, because of $rc = ail$
4. lc is the prefix of ir because of $air = alcb$

The cl-case: For the converse, the naming of the inner context i and the outer context c is swapped, starting with the word il:

$$il \rightarrow ir = lcb$$
$$lcb \rightarrow rcb = a\mathbf{il}b$$

This leads to the same constraints.

1. i is a prefix of l, because of $ir = lcb$
2. c is a suffix of l, because of $rcb = ailb$ ($rc = ail$)
3. lc is the prefix of ir because of $ir = lcb$
4. il is the suffix of rc, because of $rcb = ailb$

It can be seen that both cases exhaust all possibilities and that the case ilc need not be examined further.

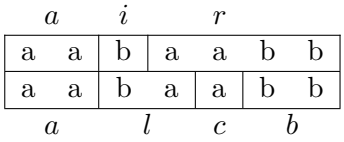


Figure 11: $air = alcb$ for $ba \rightarrow aab$

From Figure 10 one can see that for empty c, $l \rightarrow r$ has a loop of length one, because l then needs to be a suffix of r. The converse case follows from Figure 11 and empty i.

If $i = l$, then (see Figure 10) ll is to be a suffix of rc and $l \rightarrow r$ then has a loop of length 1. The same holds for $c = l$.

We ignore the cases that are actually loops of length one for this predicate and only encode loops of exactly length two.

Within the framework of Boolean functions this can be defined as:

$$\text{two\_loop}^{\mu}(l,r) = \exists p,q \in \{1...|l|-1\} : \\ \text{pre}(l(l_{p..}), l_{..q-1}r) \wedge \text{suf}(l_{..q-1}l, r(l_{p..})) \quad (43)$$

From this the following code for the *two_loop* predicate is derived:

```
two_loop :: EncRule -> EncBool
two_loop (l, r) =
    or $ for [ (a,b) | a <- [1..length l-1]
                     , b <- [1..length l-1] ] $
         \ (p,q) -> let c = drop p l
                        i = take q l
                    in ((l ++ c) `isPrefixOf` (i ++ r))
                    && ((i ++ l) `isSuffixOf` (r ++ c))
```

Listing 35: two_loop predicate implementation

i and c can overlap so that $|ic| > |l|$. For example, $baa \rightarrow aaabab$ has $i = ba$ and $c = aa$ leading to a two-loop from $w = \underline{baa}aa$ in two steps to $aaa\underline{\mathbf{baa}}\mathbf{aa}bab$.

i and c can encompass only a part of l so that $|ic| < |l|$. For example $bab \rightarrow abbba$ has $i = b$ and $c = b$ leading to a two-loop from $w = \underline{bab}b$ in two steps to $ab\underline{\mathbf{bab}}\mathbf{b}ba$.

i and c can fully encompass l so that $ic = l$. This is the case with the given example $ba \rightarrow aab$.

6.2.3 Criterion-D

Kurth's Criterion-D is an argument of termination, if certain overlaps between the left-hand side and the right-hand side of a rule do not exist. The system terminates if one or both of the overlaps l, r or r, l is empty. [Ges02, Theorem 2.7]

$$\text{critd}(l, r) = (\text{OVL}(l, r) = \emptyset) \vee (\text{OVL}(r, l) = \emptyset) \tag{44}$$

Where the Overlap between a and b is the set of all $w \in \Sigma^+$ where w is both a suffix of a and a prefix of b.

$$\text{OVL}(a, b) = \{w | w \in \Sigma^+ \wedge \text{suf}(w, a) \wedge \text{pre}(w, b)\} \tag{45}$$

Within the framework of allowed binary functions, we can define OVL^μ as $\text{OVL}(a, b) \neq \emptyset$ the following way:

$$\text{OVL}^\mu(a, b) = \exists d \in \{1..|a| - 1\} : \text{pre}(a_{d..}, b) \tag{46}$$

The implementations of OVL^μ and critd^μ follow naturally from these definitions:

```
overlap :: EncWord -> EncWord -> EncBool
overlap a b = or $ for [1 .. length a - 1] $
                                    \d -> isPrefixOf (drop d a) b
criterion_d :: EncRule -> EncBool
criterion_d (l, r) = (not $ overlap l r) || (not $ overlap r l)
```

Listing 36: Overlap and Criterion-D implementation

6.2.4 Subset

The subset-property is a basic property of rules generated by Kurth's Sieve in that the left-hand side of a rule is composed of letters of the right-hand side of the rule. [Ges02] It follows that every letter of the left-hand side also exists in the right-hand side.

$$\text{subset}(l, r) \iff \forall a \in \Sigma : (|l|_a > 0 \implies |r|_a > 0) \tag{47}$$

The rule $ab \to bbaa$ has the subset-property while $abc \to bbaa$ has not.

Rules that do not have the subset-property trivially terminate by counting letters.

Within the framework of Boolean functions subset can be defined as follows:

$$\text{subset}^\mu(l, r) = \forall a \in \Sigma : (\bigvee l_{*,a} \implies \bigvee r_{*,a}) \tag{48}$$

From this the following code for the *subset* predicate is derived:

```
subset :: Idx -> EncRule -> EncBool
subset (Idx sigma) (l, r) = and $ for [0..(sigma - 1)] $ \idx ->
              let inl = or (extract_letter_bits l (Idx idx))
                  inr = or (extract_letter_bits r (Idx idx))
              in (inl ---> inr)
```

Listing 37: subset predicate implementation

This predicate implies the grid-property (see Section 6.2.5) and need not be considered if rules with the grid-property aren't allowed.

6.2.5 Grid Rule

A grid rule is a rewriting rule $l \to r$ that has at least one letter occur equally as often or more often in l than in r.

$$\text{grid}(l, r) \iff \exists a \in \Sigma : (0 < |l|_a \geq |r|_a) \tag{49}$$

A system that has the grid property either terminates or has a loop of length no longer than two [Ges02]. Therefore every grid system that does not have a loop of length one or two (see Sections 6.2.1 and 6.2.2) terminates.

The rule $ab \to baa$ has the grid-property while $ab \to bbaa$ has not.

To implement this predicate we use the following equality:

$$|l|_a \geq |r|_a = \begin{cases} |l_{1\ldots}|_a \geq |r_{1\ldots}|_a & l_{0,a} = r_{0,a} \\ |l_{1\ldots}|_a \geq |r_{1\ldots}|_a - 1 & l_{0,a} \wedge \neg r_{0,a} \\ |l_{1\ldots}|_a \geq |r_{1\ldots}|_a + 1 & \text{otherwise} \end{cases} \tag{50}$$

Together with the fact that $|\epsilon|_a = 0$, we define a function more_or_equal(d, l, r) with the semantics that for the number of bits in l set to true $(\#l)$, the number of bits in r set to true $(\#r)$, and some integer d the following holds: $\#l - \#r >= d$.

```
more_or_equal :: Int -> [EncBool] -> [EncBool] -> EncBool
more_or_equal d l r = case (l, r) of
  ([], [])    -> boolconst (d >= 0)
  ([], r:rs) -> ite r (more_or_equal (d - 1) [] rs)
                      (more_or_equal d [] rs)
  (l:ls, rs) -> ite l (more_or_equal (d + 1) ls rs)
                      (more_or_equal d ls rs)
```

Listing 38: more_or_equal helper predicate implementation

The implementation of `grid` follows naturally from that

```
grid :: Idx -> EncRule -> EncBool
grid (Idx sigma) (l, r) = or $ for [0..(sigma-1)] $ \ idx ->
            let ls = extract_letter_bits l (Idx idx)
                rs = extract_letter_bits r (Idx idx)
            in or ls && more_or_equal 0 ls rs
```

Listing 39: grid predicate implementation

Starting not from `more_or_equal 0` but from `more_or_equal (-1)` allows us to have a strict version of the grid property that implies termination by letter counting.

6.2.6 Sénizergues

If the left-hand side of a system is of the form a^*b^*, there is decision procedure for the termination problem [Sén96]. We ignore the case where the left-hand side is empty and restrict our implementation to the form a^+b^*.

The rule $ab \to bbaa$ has the Sénizergues-property while $aba \to bbaa$ does not.

The formulation of sen is easily defined as:

$$\text{sen}(l, r) \iff \exists a, b \in \Sigma : a \neq b \wedge l \in a^+b^* \tag{51}$$

In the framework of μ-compatible functions we define the helper function apbs with the following semantic:

$$\text{apbs}(w, s) \iff \begin{cases} w \in b^+ & s \\ w \in a^+b^* & \text{otherwise} \end{cases} \tag{52}$$

With the first parameter w as the word that is checked for the property and the parameter s for whether the word has switched from the first to the second letter, the definition of apbs^μ is as follows:

$$\text{apbs}^\mu(w, s) = \begin{cases} 1 & |w| = 1 \\ \text{apbs}(w_{1..}, \overline{w_0 =_l w_1}) & |w| > 1 \wedge \overline{s} \\ (w_0 =_l w_1) \wedge \text{apbs}(w_{1..}, s) & |w| > 1 \wedge s \end{cases} \tag{53}$$

From here sen^μ is easily defined as

$$\text{sen}^\mu(l, r) = \text{apbs}(l, 0) \tag{54}$$

This implementation fits the definition:

```
a_plus_b_star :: EncWord -> Bool -> EncBool
a_plus_b_star w sw = case (w, sw) of
            ([l], b) -> true
            (l:ls, False) -> ite (l `l_eq` head ls)
                                 (a_plus_b_star ls False)
                                 (a_plus_b_star ls True)
            (l:ls, True) -> (l `l_eq` head ls)
                              && a_plus_b_star ls True
senizergues :: EncRule -> EncBool
senizergues (l, r) = a_plus_b_star l False
```

Listing 40: a^+b^* and Sénizergues predicate implementation

This predicate need not be considered if the alphabet size $|\Sigma|$ is greater than two and inhibitors (see Section 6.2.7) aren't allowed.

6.2.7 Inhibitor

An inhibitor system is a system $l \to r$ that has a letter occurring in r that does not occur in l. This letter is called the "inhibitor".

$$\text{inhibitor}(l, r) \iff \exists a \in \Sigma : (|l|_a = 0 \wedge |r|_a > 0) \tag{55}$$

The rule $ab \to bbcaa$ is an inhibitor-system while $ab \to bbaa$ is not.

The inhibitor occurs in r but not in l and can not be used to apply the rule $l \to r$. Therefore any infinite reduction that occurs from some word $x_0 i x_1 i .. i x_n$ has to occur from one of the $x_?$. It has been shown that the termination of rewriting systems, where all right-hand sides have an inhibitor, is decidable [McN01].

Within the framework of Boolean functions inhibitor can be defined as follows:

$$\text{inhibitor}^\mu(l, r) = \exists a \in \Sigma : (\overline{\bigvee r_{*,a}} \wedge \bigvee l_{*,a}) \tag{56}$$

From this the following code for the inhibitor predicate is derived:

```
inhibitor :: Idx -> EncRule -> EncBool
inhibitor (Idx sigma) (l, r) = or $ for [0..(sigma - 1)] $ \idx ->
          let inl = or (extract_letter_bits l (Idx idx))
              inr = or (extract_letter_bits r (Idx idx))
          in (not inl && inr)
```

Listing 41: inhibitor predicate implementation

7 Resource Considerations and Measurements

In this chapter, we try to give an experimentally based overview of the efficiency of implemented predicates and the impact different orderings of variables or the construction of the main conjunction have.

In Section 5 we presented 12 predicates that imply (non)canonicity, some (non)termination argument, or a decision procedure. We can not present resource usage for all 4096 combinations of predicates for a few system shapes on paper. While this would still be doable, checking all 5×10^9 possible orderings of the predicates is certainly not. Therefore the necessarily incomplete analysis in this section will be as follows:

Subsection 7.1: For three different system shapes, predicates will be checked for efficiency (definition follows) on their own. The most and least efficient predicates are then combined and their combined efficiency is calculated. This is also done for all predicates combined.

Subsection 7.2: The predicates are reordered randomly within the main conjunction.

Subsection 7.3: We present 12 variable orderings and measure their resource usage for a few example shapes.

Subsection 7.4: We take some specific configurations and measure the resource usage for different system shapes. This gives a rough estimate about resource usage for larger systems.

Subsection 7.5: We eliminate variables of the BDD by fixing letters and analyze the impact on running time for different cases of elimination.

Subsection 7.6: We plot all memory measurements of this chapter against the measured time.

Subsection 7.7: Instead of just counting systems, systems are generated and the time is measured to write the systems to `stdout`.

Since it is possible that different implementations of predicates are more or less efficient for specific encodings, variable orderings, and orderings of the main conjunction, while the predicates have been implemented with the one-hot encoding and the specific "optimal" variable ordering in mind, the results in the section should be taken with a grain of salt and are possibly subject to circular logic. On the other hand, considering the sheer number of possible implementations and orderings, it is out of the scope of this thesis to question every decision and every implementation.

We also can only experimentally observe some effects the different changes and settings. The actual BDDs are much too complicated to reason about. Even for small configurations ($|\Sigma| = 2, |l| = 4, |r| = 6$, pnR^+got^-, see Figure 13 for predicate short names) with few models (42) the resulting BDD is beyond what can reasonably be understood (Figure 12) by looking at it.

If not otherwise stated, all times are given in seconds, all memory usages are given in kilobytes. The measurements have been done on a hexacore *AMD Phenom II X6 1090T Processor* processor with 16GB of RAM.

Short	Description	Definition
r	canonical after reversal	6.1.1
n	canonical after renaming	6.1.2
R	canonical after reversal and renaming	6.1.3
p	2-coding	6.1.4
b	bordered	6.1.5
o	one-loop/factor	6.2.1
t	two-loop	6.2.2
d	criterion-D	6.2.3
s	subset	6.2.4
g	grid rule	6.2.5
z	Sénizergues	6.2.6
i	inhibitor	6.2.7
a	all letters used	5.3
c	consistency predicate (positive only)	5.1

Predicate p^+ is included as positive, so that p holds for a system and predicate p^- is included as the negative, that it doesn't hold for a system.

Figure 13: Short names of predicates

7.1 Efficiency of Predicates

We define the **efficiency** of a predicate as the cardinality of the set of systems of a shape minus the cardinality of the set of systems for which the predicate holds divided by the resource used. In the case of time efficiency, this number depends on the processor speed.

For example, for the shape $3, 5, 8$ ($|\Sigma| = 3, |l| = 5, |r| = 8$) there are 1 594 323 systems. If grid-systems aren't allowed, there are 107 907 systems remaining and filtering took 26ms. The time efficiency therefore is $(1\,594\,323 - 107\,907)/1000/26 = 57.2$. The unit is thousand systems per ms but is omitted. For the analogue case with measured memory, the unit is thousand systems per KB, but is omitted as well.

In comparison, for this shape and loops of length one disallowed, 1 568 220 systems remain after 19ms of calculation, leading to a time efficiency of just 1.4. By definition, the efficiency of no predicate and no filtering is 0.

If both predicates are disallowed, for this shape after $0.468s$ 99 900 systems remain, leading to a combined time efficiency of about 3.2.

In case of time efficiency, if there is a method of post-filtering for a predicate (see Section 8.2) that can analyze on average more systems per millisecond than the time

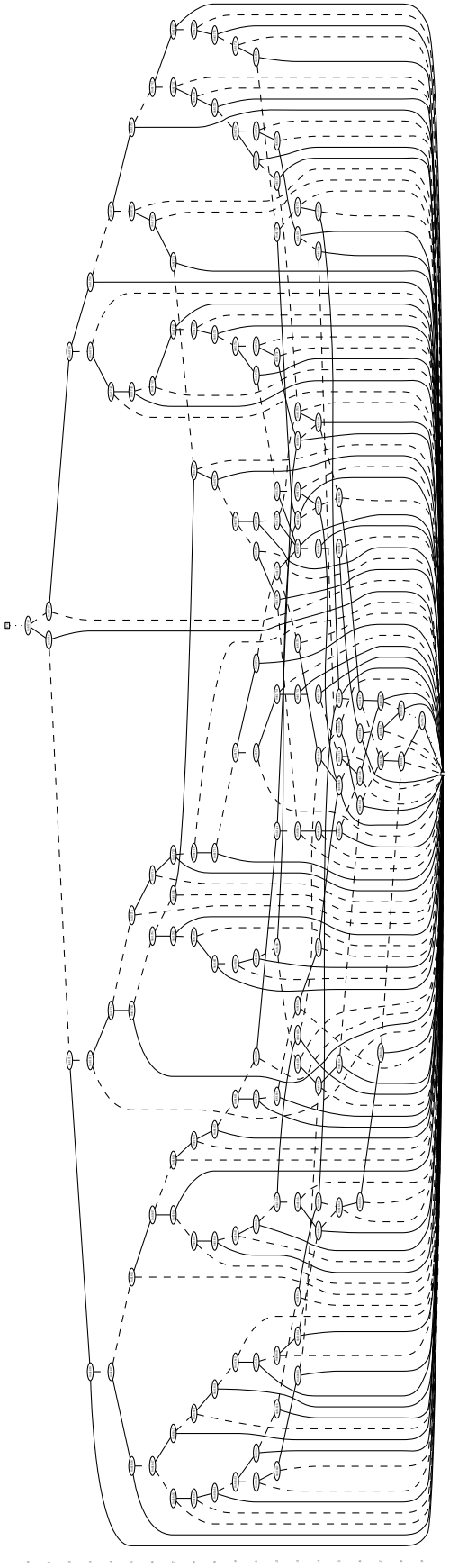


Figure 12: $(3, 4, 7, pmR^{+}got^{-})BDD$

efficiency of that predicate, it is more efficient to not use that predicate at all and defer the systems to the post-filter.

We use the following three configurations for experimentally testing time (E_t) and memory (E_m) efficiency of the predicates: $C_1 = (2, 7, 10)$, $C_2 = (3, 6, 9)$, and $C_3 = (4, 6, 10)$. The short names of predicates can be seen in seen in Figure 13. The raw measured resource usage is to be found in Table 1 and the resulting efficiencies in Table 2 (p59/60).

We make the following observations:

- There are indeed predicates that are strictly better than others. We can group the predicates into good predicates that have minimal resource footprint and exclude many systems ($r^+n^+g^-i^-$) and bad predicates that have large resource footprint and exclude few systems (o^-t^-), while the rest has either "medium" run time behaviour and minimal impact on the result.
- The predicates that have a worse impact on run time also have a worse impact on memory usage and vice versa
- If the conjunction includes one of the worse predicates it has worse run time behaviour. If it only includes "good" predicates, the run time behaviour is also good.
- For predicate combinations, the maximum and minimum efficiencies seem to be a bound for the combined efficiency.
- For the "good" predicates the sum of used resources seems to be an upper bound for the used resources of the conjunction.
- The "good" predicates generally increased their time efficiency with larger shapes, while the "bad" ones did not.

The following seems unclear or has no obvious reason

- Why do o^- and t^- have such a bad resource footprint while b^- does not? Both use word equality over a comparable amount of words.
- while the resource footprint for combining "good" predicates seems to be bounded by the sum of used resources, this is not true when o^- is included in the main conjunction.

It is useful to test predicates beforehand and it might be useful not to include the "bad" predicates if some traditional filtering step can do it while using less resources (testing for short loops after the fact could certainly be implemented). It does not seem possible to make a reasonable prediction about resource usage of predicate combinations from the single predicates. The fact that (for the "good" predicates) the sum of the used resources seems to be an upper bound for the combined resource use just means that no measurements should have been done beforehand since the BDD with the combined predicates has already been calculated by the time the BDDs for the single predicates are done being calculated.

p	$\lvert C_1\rvert = 131072$			$\lvert C_2\rvert = 14348907$			$\lvert C_3\rvert = 4294967296$		
	$\frac{\lvert p\rvert}{\lvert C_1\rvert}$	$t(s)$	$m(MB)$	$\frac{\lvert p\rvert}{\lvert C_2\rvert}$	$t(s)$	$m(MB)$	$\frac{\lvert p\rvert}{\lvert C_3\rvert}$	$t(s)$	$m(MB)$
r^+	50.2%	0.011	9	50.02%	0.015	15	50.0%	0.138	14
n^+	50.0%	0.009	11	16.67%	0.01	10	4.19%	0.011	14
R^+	25.1%	0.016	14	12.51%	0.059	17	7.14%	2.794	71
p^+	100.0%	0.009	14	99.97%	0.011	9	99.74%	0.029	15
o^-	96.9%	0.01	12	99.45%	0.086	20	99.88%	9.222	560
t^-	99.85%	0.016	13	100.0%	0.166	18	100.0%	15.265	254
d^-	53.09%	0.013	10	19.56%	0.07	18	9.74%	7.296	446
s^+	99.81%	0.009	14	92.89%	0.01	14	81.87%	0.011	8
g^-	37.1%	0.151	15	5.6%	0.071	14	2.09%	0.191	15
z^-	90.62%	0.011	9	95.88%	0.011	9	98.54%	0.011	9
i^-	98.44%	0.009	14	74.74%	0.009	14	41.06%	0.012	14
b^-	86.62%	0.006	14	96.2%	0.011	15	98.42%	0.178	15
rn^+	37.6%	0.008	7	14.59%	0.013	13	3.89%	0.102	22
rp^+	50.2%	0.008	13	50.01%	0.013	15	49.87%	0.151	24
r^+o^-	48.65%	0.015	5	49.75%	0.238	34	49.94%	24.281	1208
r^+t^-	50.11%	0.015	14	50.02%	0.132	19	50.0%	15.889	258
r^+g^-	18.6%	0.152	15	2.8%	0.073	15	1.04%	0.504	28
r^+i^-	49.39%	0.011	13	37.38%	0.014	13	20.53%	0.119	22
nR^+	25.1%	0.016	12	8.34%	0.066	18	2.1%	3.245	67
np^+	50.0%	0.012	14	16.66%	0.013	14	4.18%	0.044	17
n^+o^-	48.45%	0.011	14	16.58%	0.128	26	4.19%	28.509	1505
n^+t^-	49.93%	0.012	8	16.67%	0.129	18	4.19%	16.904	271
n^+g^-	18.55%	0.145	15	0.93%	0.06	14	0.09%	0.221	22
n^+i^-	49.22%	0.009	11	12.46%	0.01	13	1.73%	0.008	14
p^+ot^-	96.8%	0.015	14	99.42%	0.476	47	99.62%	112.827	4391
rn^+gi^-	14.18%	0.147	15	0.75%	0.073	15	0.05%	0.351	33
all	4.17%	0.243	21	0.11%	15.898	368	aborted after 16GB/15min		

Table 1: Raw measurements of predicate running time and memory

p	$\lvert C_1\rvert = 131072$			$\lvert C_2\rvert = 14348907$			$\lvert C_3\rvert = 4294967296$		
	$\frac{\lvert p\rvert}{\lvert C_1\rvert}$	E_t	E_m	$\frac{\lvert p\rvert}{\lvert C_2\rvert}$	E_t	E_m	$\frac{\lvert p\rvert}{\lvert C_3\rvert}$	E_t	E_m
r^+	50.2%	$5.9 \cdot 10^6$	7.3	50.02%	$4.8 \cdot 10^8$	479.0	50.0%	$1.6 \cdot 10^{10}$	$1.5 \cdot 10^5$
n^+	50.0%	$7.3 \cdot 10^6$	5.7	16.67%	$1.2 \cdot 10^9$	1152.0	4.2%	$3.7 \cdot 10^{11}$	$2.9 \cdot 10^5$
R^+	25.1%	$6.1 \cdot 10^6$	6.7	12.51%	$2.1 \cdot 10^8$	702.0	7.14%	$1.4 \cdot 10^9$	$5.5 \cdot 10^4$
p^+	100.0%	0.0	0.0	99.97%	$3.9 \cdot 10^5$	0.46	99.74%	$3.9 \cdot 10^8$	751.0
o^-	96.9%	$4.1 \cdot 10^5$	0.32	99.45%	$9.1 \cdot 10^5$	3.79	99.88%	$5.7 \cdot 10^5$	9.14
t^-	99.85%	$1.2 \cdot 10^4$	0.01	100.0%	3560.0	0.03	100.0%	443.0	0.03
d^-	53.09%	$4.7 \cdot 10^6$	6.1	19.56%	$1.6 \cdot 10^8$	627.0	9.74%	$5.3 \cdot 10^8$	8490.0
s^+	99.81%	$2.8 \cdot 10^4$	0.02	92.89%	$1.0 \cdot 10^8$	72.0	81.87%	$7.1 \cdot 10^{10}$	$9.7 \cdot 10^4$
g^-	37.1%	$5.5 \cdot 10^5$	5.51	5.6%	$1.9 \cdot 10^8$	934.0	2.09%	$2.2 \cdot 10^{10}$	$2.6 \cdot 10^5$
z^-	90.62%	$1.1 \cdot 10^6$	1.28	95.88%	$5.4 \cdot 10^7$	61.7	98.54%	$5.7 \cdot 10^9$	6576.0
i^-	98.44%	$2.2 \cdot 10^5$	0.14	74.74%	$4.0 \cdot 10^8$	252.0	41.06%	$2.1 \cdot 10^{11}$	$1.8 \cdot 10^5$
b^-	86.62%	$2.9 \cdot 10^6$	1.22	96.2%	$5.0 \cdot 10^7$	35.6	98.42%	$3.8 \cdot 10^8$	4294.0
rn^+	37.6%	$1.0 \cdot 10^7$	11.0	14.59%	$9.4 \cdot 10^8$	904.0	3.89%	$4.0 \cdot 10^{10}$	$1.9 \cdot 10^5$
rp^+	50.2%	$8.2 \cdot 10^6$	5.1	50.01%	$5.5 \cdot 10^8$	481.0	49.87%	$1.4 \cdot 10^{10}$	$8.6 \cdot 10^4$
r^+o^-	48.65%	$4.5 \cdot 10^6$	14.0	49.75%	$3.0 \cdot 10^7$	210.0	49.94%	$8.9 \cdot 10^7$	$1.7 \cdot 10^3$
r^+t^-	50.11%	$4.4 \cdot 10^6$	4.4	50.02%	$5.4 \cdot 10^7$	371.0	50.0%	$1.4 \cdot 10^8$	$8.1 \cdot 10^3$
r^+g^-	18.6%	$7.0 \cdot 10^5$	7.2	2.8%	$1.9 \cdot 10^8$	907.0	1.04%	$8.4 \cdot 10^9$	$1.5 \cdot 10^5$
r^+i^-	49.39%	$6.0 \cdot 10^6$	4.9	37.38%	$6.4 \cdot 10^8$	686.0	20.53%	$2.9 \cdot 10^{10}$	$1.5 \cdot 10^5$
nR^+	25.1%	$6.1 \cdot 10^6$	8.0	8.34%	$2.0 \cdot 10^8$	694.0	2.1%	$1.3 \cdot 10^9$	$6.2 \cdot 10^4$
np^+	50.0%	$5.5 \cdot 10^6$	4.7	16.66%	$9.2 \cdot 10^8$	816.0	4.18%	$9.4 \cdot 10^{10}$	$2.4 \cdot 10^5$
n^+o^-	48.45%	$6.1 \cdot 10^6$	4.7	16.58%	$9.4 \cdot 10^7$	443.0	4.19%	$1.4 \cdot 10^8$	$2.7 \cdot 10^3$
n^+t^-	49.93%	$5.5 \cdot 10^6$	7.6	16.67%	$9.3 \cdot 10^7$	654.0	4.19%	$2.4 \cdot 10^8$	$1.5 \cdot 10^4$
n^+g^-	18.55%	$7.4 \cdot 10^5$	7.2	0.93%	$2.4 \cdot 10^8$	983.0	0.09%	$1.9 \cdot 10^{10}$	$1.9 \cdot 10^5$
n^+i^-	49.22%	$7.4 \cdot 10^6$	6.0	12.46%	$1.3 \cdot 10^9$	960.0	1.73%	$5.3 \cdot 10^{11}$	$2.9 \cdot 10^5$
p^+ot^-	96.8%	$2.8 \cdot 10^5$	0.3	99.42%	$1.7 \cdot 10^5$	1.7	99.62%	$1.4 \cdot 10^5$	3.6
rn^+gi^-	14.18%	$7.7 \cdot 10^5$	7.5	0.75%	$2.0 \cdot 10^8$	934.0	0.05%	$1.2 \cdot 10^{10}$	$1.3 \cdot 10^5$
all	4.17%	$5.2 \cdot 10^5$	5.9	0.11%	$9.0 \cdot 10^5$	38.0	aborted after 16GB/15min		

Table 2: Predicate running time and memory efficiency

7.2 Order of Main Conjunction

The order of construction of the main conjunction should in theory have impact on the resource usage of the construction procedure. At least if the result set is known to be empty earlier, the conjunction should short-circuit earlier. To achieve this we include the predicates "whole alphabet" and "grid" to allow for short-circuiting for certain configurations.

As configured shapes we take $C_1 = (2, 10, 11)$, $C_2 = (2, 9, 12)$, and $C_3 = (3, 7, 9)$. The number of models for the first and the last configuration is 0 while the middle has 83 918 models. We enable all predicates and test a few (10) permutations chosen at random (Table 3). We expect to see, that for those permutations with the predicates g^-, a^+, and c far to the left, for the short-circuiting configurations, use far less resources than the other permutations.

π	C_1 (t)	C_1 (m)	C_2 (t)	C_2 (m)	C_3 (t)	C_3 (m)
$n^+g^-o^-d^-z^-ci^-b^-t^-s^+a^+R^+p^+$	2.130	21 080	2.149	24 724	1.475	57 672
$n^+t^-s^+g^-d^-o^-ca^+p^+R^+b^-i^-z^-$	2.219	20 716	2.210	23 016	1.400	57 220
$d^-i^-o^-t^-z^-s^+b^-a^+p^+g^-n^+cR^+$	2.405	18 488	2.437	21 644	1.228	55 944
$n^+b^-o^-g^-a^+cs^+i^-R^+t^-p^+z^-d^-$	2.539	43 596	3.226	47 148	16.536	300 704
$g^-z^-R^+p^+n^+s^+t^-b^-i^-ca^+d^-o^-$	2.566	38 000	3.354	49 836	18.802	303 708
$z^-b^-R^+a^+g^-s^+t^-n^+d^-co^-p^+i^-$	2.613	26 412	2.912	45 356	17.234	328 216
$g^-cz^-s^+R^+p^+n^+t^-i^-d^-o^-a^+b^-$	2.630	44 332	2.827	52 116	21.517	379 188
$a^+cd^-z^-g^-p^+R^+i^-s^+b^-o^-t^-n^+$	3.476	54 224	5.654	93 392	30.112	424 376
$p^+z^-s^+b^-o^-t^-ca^+i^-g^-R^+n^+d^-$	4.090	51 912	5.742	73 432	23.415	302 880
$s^+i^-cp^+b^-o^-g^-R^+n^+a^+d^-z^-t^-$	4.179	62 144	6.943	101 864	27.444	366 292
Average	2.885	38 090	3.745	53 252	15.916	257 620
Standard Deviation	0.713	14 870	1.626	26 872	10.332	136 463

time (t) in seconds, memory (m) in megabyte

Table 3: Impact of order of main conjunction (`foldr`)

And indeed we see some short-circuiting behaviour for some configurations but it is not entirely clear what the cause for this behaviour is. The first three permutations show the best run time behaviour but for C_1 and C_2 this is only a slight improvement by up to factor 3 while for C_3 the improvement is considerably larger (up to factor 20). It is not clear what the three best permutations have in common that they lead to a significant difference in run time behaviour. Within the functional interface for CUDD we can not easily inspect the sub-results, and if we could, this possibly changes any behaviour caused by lazy evaluation.

The `all` predicate that is used to build the main conjunction is implemented as a `foldr`, so that the term $P_1 \vee P_2 \vee P_3 \vee P_4 \vee P_5$ is evaluated as $P_1 \vee (P_2 \vee (P_3 \vee (P_4 \vee P_5)))$. This is done in order to reuse previous evaluations of the tail of the conjunction (see one-hot

example, Section 5.1). Considering possible BDD-implementations, it might be that only the left-hand side of a conjunction is checked for being a constant node (0 or 1-nodes of the BDD). By folding the conjunction to the right, there may often be large BDDs on the left side of the operation while folding to the left ($(((P_1 \vee P_2) \vee P_3) \vee P_4) \vee P_5$) could lead to smaller BDDs on the left-side of the operation. This reduces the reusability of the tail of the fold. Replacing `all` with `foldl (&&) true` we again try 10 random permutations (Table 4).

π	C_1 (t)	C_1 (m)	C_2 (t)	C_2 (m)	C_3 (t)	C_3 (m)
$g^-cz^-s^+R^+p^+n^+t^-i^-d^-o^-a^+b^-$	2.066	18 680	2.159	20 788	0.911	55 652
$i^-g^-R^+ca^+p^+t^-n^+b^-s^+d^-o^-z^-$	2.143	19 424	2.179	23 540	1.086	65 248
$s^+o^-t^-g^-z^-cd^-R^+i^-a^+p^+b^-n^+$	2.152	21 964	2.359	39 888	40.649	270 112
$d^-co^-i^-a^+R^+g^-t^-s^+z^-b^-p^+n^+$	2.164	20 824	2.191	24 824	1.138	56 800
$cR^+b^-s^+z^-a^+t^-n^+i^-d^-o^-g^-p^+$	2.170	17 940	2.177	19 040	0.987	58 072
$i^-o^-s^+t^-n^+d^-g^-z^-b^-p^+ca^+R^+$	2.355	44 220	2.424	44 744	20.854	499 616
$g^-b^-n^+R^+d^-z^-p^+t^-s^+o^-ca^+i^-$	2.627	44 112	2.888	63 924	15.145	225 080
$z^-n^+g^-b^-a^+i^-p^+R^+d^-o^-s^+ct^-$	2.641	43 432	2.873	62 472	15.939	218 980
$b^-t^-i^-s^+z^-d^-g^-R^+a^+n^+o^-p^+c$	2.883	51 904	3.919	69 420	62.643	633 264
$z^-b^-p^+R^+a^+s^+t^-d^-g^-i^-n^+o^-c$	3.131	50 584	3.837	70 936	46.821	574 916
Average	2.433	33 308	2.701	43 958	20.617	265 774
Standard Deviation	0.349	13 820	0.643	20 188	21.073	214 555

time (t) in seconds, memory (m) in megabyte

Table 4: Impact of order of main conjunction (`foldl`)

Using `foldl` has indeed had impact on the resource usage. While for the configurations C_1 and C_2 the results for the taken time were closer together (differences of less than factor 0.5 for C_1 and less than factor 2 for C_2) and better overall, for C_3 there were extreme results that differed by a factor of over 65. The average results for C_3 were worse using `foldl` but had better top results. The memory usage for a given time with `foldl` is comparable to those in `foldr`.

Those permutations that were fastest had the consistency predicate c far to the left while those that fared worst had it far to the right. Such relation was not visible before. This might be a huge factor since the consistency predicate forces every path to 1 to include all variables. Having this predicate included enforces the maximum depth on the BDD. The position of this predicate could very well have more of an impact on the running time and memory usage than the position of other predicates.

Replacing `all = foldr (&&) true` with a `foldl`-based implementation has indeed a negative effect on the time it takes to create the main conjunction.

From this point on we take use the construction order $g^-cz^-s^+R^+p^+n^+t^-i^-d^-o^-a^+b^-$ with `foldl`.

7.3 Variable Ordering

The default variable ordering is derived implicitly (see Listing 42). Other orderings are defined by switching `L` and `R` or implementing `Ord Var` to overwrite the implicitly derived ordering. In the implicit ordering all variables belonging to the left-hand side come before the variables belonging to the right-hand side (or vice versa if `L` and `R` are switched). All variables (of the same side) belonging to the nth letter of a side come before the $n+1$th letter of a side. A letter (of the same word of the same side) being the ith letter of the alphabet comes before the letter being the $i+1$th word of the alphabet.

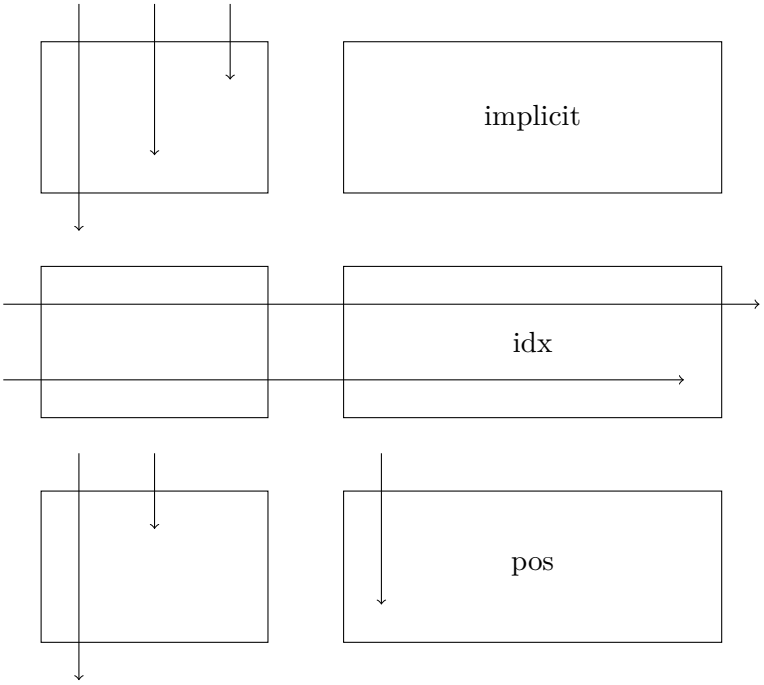
```
data Side = L | R deriving ( Eq, Ord, Show )
newtype Pos = Pos Int deriving ( Eq, Ord, Show, Read, Enum, Num )
newtype Idx = Idx Int deriving ( Eq, Ord, Show, Read, Enum, Num )
type Var = (Side, Pos, Idx)
```

Listing 42: Implicitly defined variable ordering

We check the following orderings:

- implr: implicit, left-hand side before right-hand side
- imprl: implicit, right-hand side before left-hand side
- idxlr: all variables designating the first letter of the alphabet come before all variables designating the second letter etc., left-hand side before right-hand side
- idxrl: all variables designating the first letter of the alphabet come before all variables designating the second letter etc., right-hand side before left-hand side
- poslr: all variables designating the first letter of a word come before all variables designating the second letter etc., left-hand side before right-hand side
- posrl: all variables designating the first letter of a word come before all variables designating the second letter etc., right-hand side before left-hand side
- all orderings reversed (~)

Again we take the predicates (with the average order $i^- o^- s^+ t^- n^+ d^- g^- z^- b^- p^+ c a^+ R^+$ using `foldl`) and shapes ($C_1 = (2, 10, 11)$, $C_2 = (2, 9, 12)$, and $C_3 = (3, 7, 9)$) from the previous section.



The boxes represent the left- and right-hand side of the rules respectively using the table representation for rules $l \to r$.

Figure 14: Different possible orderings

ordering	C_1 (t)	C_1 (m)	C_2 (t)	C_2 (m)	C_3 (t)	C_3 (m)
implr	2.064	40 540	2.235	44 832	21.928	499 416
imprl	2.981	59 028	3.533	82 480	87.082	1 513 884
~implr	2.869	57 828	3.407	80 184	69.513	1 293 196
~imprl	2.581	41 524	2.569	44 604	15.708	456 740
idxlr	14.354	496 816	18.020	501 868	aborted after 8gb/3min	
idxrl	24.242	664 828	28.227	628 760	aborted after 8GB/3min	
~idxlr	18.313	557 072	22.732	608 472	aborted after 8GB/3min	
~idxrl	21.841	560 348	23.685	606 116	aborted after 8GB/3min	
poslr	3.265	76 332	3.034	97 648	76.208	1 179 356
posrl	3.525	76 196	3.258	109 576	81.615	1 237 904
~poslr	3.634	105 304	3.298	104 292	103.377	1 759 464
~posrl	4.100	112 792	3.664	120 616	105.994	1 758 652

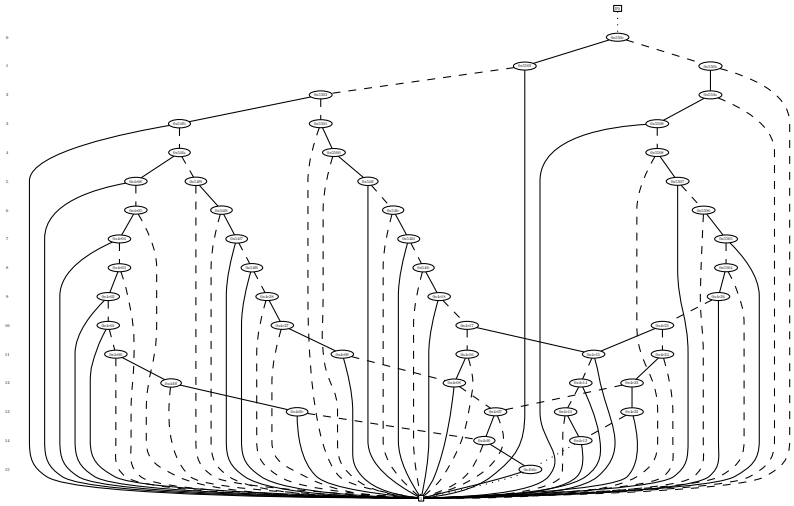
time (t) in seconds, memory (m) in megabyte

Table 5: Impact of variable ordering

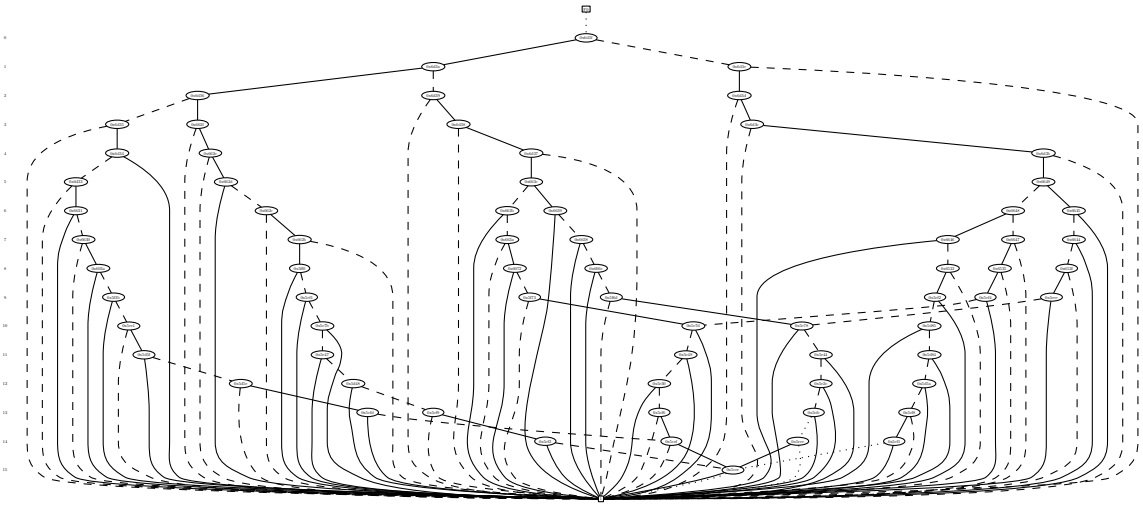
Looking at the measured resource usages in Table 5 we see that the variable ordering indeed has a huge impact. The wrong ordering can lead to more resources used by a factor of more than 10. This is not surprising, considering that the correct variable ordering is important for decreasing BDD size (see Section 7.3). The already chosen (implicit) ordering is the best ordering from the set of possible orderings. There are algorithms to dynamically order or reorder the variables of a BDD to minimize BDD size [Rud93] while it is constructed. CUDD supports multiple reordering algorithms which are also exposed by the Haskell bindings for CUDD but are not available for the functional interface provided by the bindings library but only the imperative interface.

The difference between the orderings is visible looking at the drawn BDDs (see Figure 15). The BDD with the implicit variable ordering has 51 nodes while the BDD with the index ordering has 70. For larger shapes ($(3, 6, 9)$ with the same configuration) we cannot draw the BDDs but we can count the number of nodes and the difference between the node counts only increases with larger configurations. The BDD with the implicit variable ordering has 11155 nodes while the BDD with the index ordering has 65473. For the smaller BDDs the index ordered BDD is larger by about 40% while for the larger BDDs the difference is about as large as factor 6.

It is not surprising that the implicit orders have the best result, since these were the orderings we had in mind when implementing the predicates in the first place. It is unclear why the implicit ordering rl is much worse than the implicit ordering lr. We go on using the ordering *implr*.



Implicit ordering



Index ordering

Figure 15: Variable order comparison for $(2, 3, 5, aR^{+}got^{-})$

7.4 Resource Usage with System Size

We take as configurations an^+gi^- ("easy", Figure 16), which is a configuration that is semantically similar to a combinatorial approach of traditional sieving. A combinatorial implementation for the canonical-after-renaming (n) predicate was already shown (see Section 2.2). It is imaginable that a letter counter that restricts generated systems to non-grid systems can be used to extend a system generator. As a second configuration we take $anRp^+dgotdib^-$ ("hard", Figure 17) which includes most of the presented predicates and is close to what has been used for finding interesting systems (see Section 8.1).

As shapes we take $|\Sigma| = 2$ (brown), $|\Sigma| = 3$ (blue), and $|\Sigma| = 4$ (red) with all useful combinations (see Section 8.1) for the lengths of l and r for the rule $l \to r$ up to a maximum system length of $|lr| \leq 26$ on the x-axis. For the entry $|lr| = 24$ with $|\Sigma| = 2$ we have the combinations $(2, 11, 13), (2, 10, 14)$, etc. up to $(2, 2, 22)$. The taken time is shown on the left y-axis with crosses for data points while the taken memory is shown on the right y-axis using circles. Both axis are scaled logarithmically.

As we can see in both Figure 16 and 17, the time to calculate the main conjunction for the given shapes for $|\Sigma| = 2$ can be described by an exponential function with the base two, meaning that the time taken doubles with every growth of $|lr|$ by one. This seems also true for $|\Sigma| = 3$ and $|\Sigma| = 4$ with the easy configuration. Whether the growth is also exponential or larger than exponential for the hard configuration and $|\Sigma| = 3$ and $|\Sigma| = 4$ is not obvious from the obtained data points.

For the alphabet size 2 or fewer predicates, the run time behaviour is favorable. For larger alphabets we run into resource limits fast.

7.5 Eliminating Variables and Fixing Letters

In order to use fewer variables or distribute workload for post-processing (see Section 8.2), some variable positions can be preset with constant values. Instead of the first letter of the right-hand side of a rule being composed of $(l_1, l_2, ..l_|\Sigma|)$ it is composed of BDD constants $(1, 0, .., 0)$ meaning the first letter a.

Using user-provided glob-patterns (see Section 5.2) we can replace variables within the BDD with constants (the true and false-node). Since both variables and true/false are nodes within the BDD, this replacement is transparent to the implemented predicates.

Any set letter of the suffix and prefix of a pattern up to a `*` can be replaced by constants. Given, for example, a pattern like `*aaa*` for a word of length 5, the middle letter is necessarily an a but for replacing variables with constants, these cases are ignored. Given the pattern `ab*c*` we can replace the variables encoding the first two letters with constants. With the pattern `a?b*c*d` we can replace the variables encoding the first, third and last letter with constants.

From the globbing patterns for the left-hand side and right-hand side of the rewriting system a set of predefined positions is derived and a mapping from the position to the

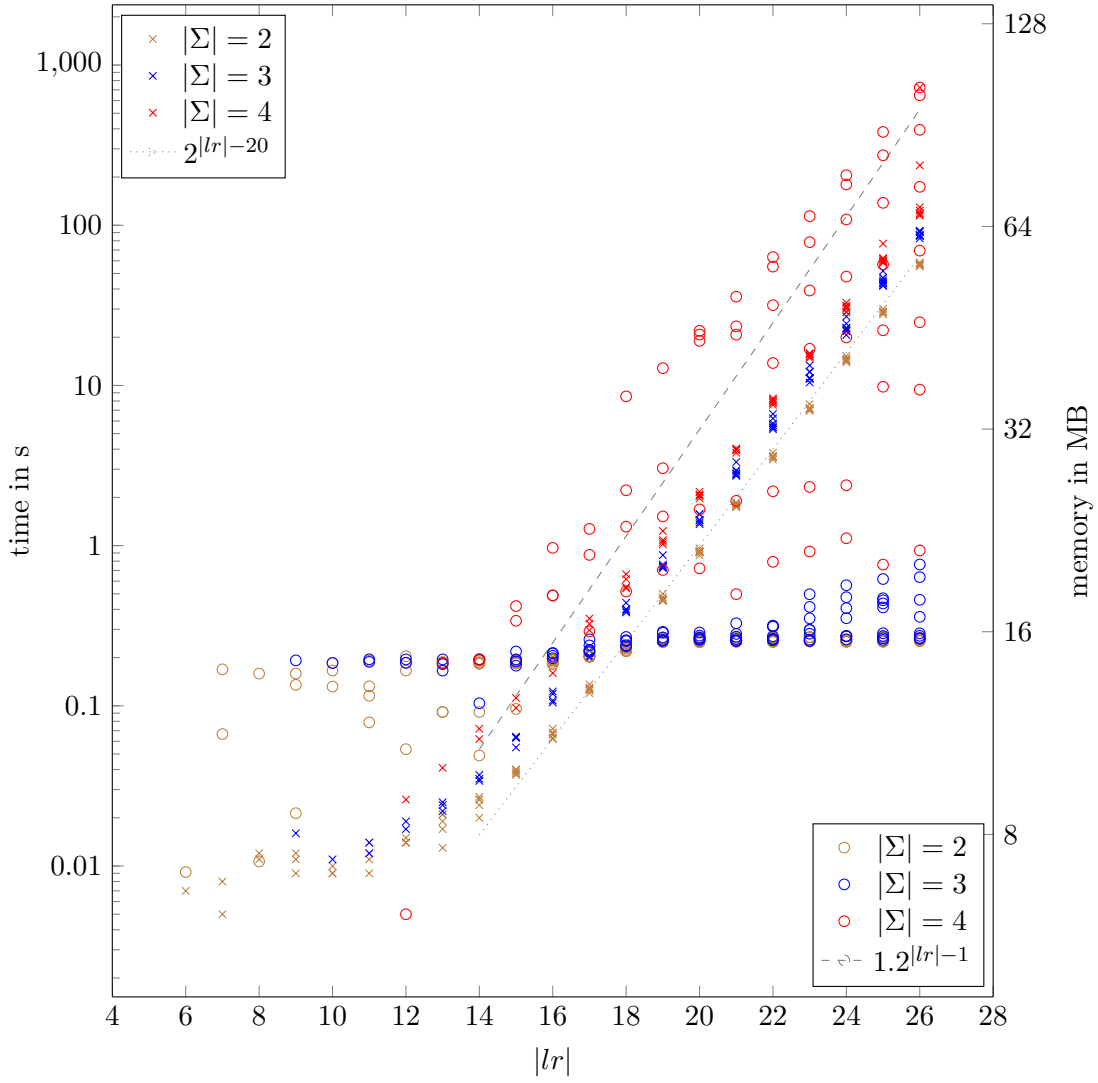


Figure 16: Growth of resource usage with growing shape for configuration an^+gi^-

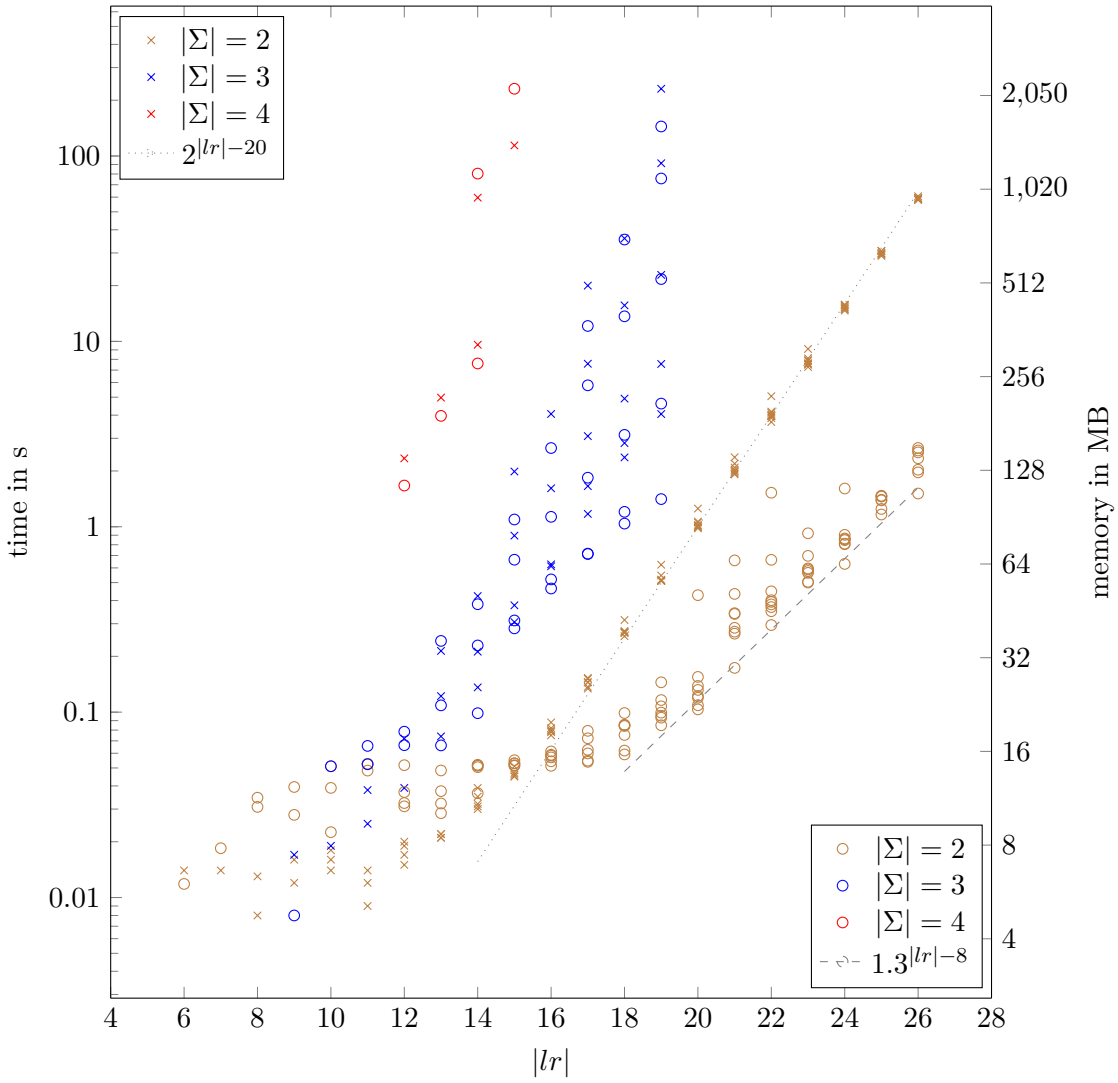


Figure 17: Growth of resource usage with growing shape for configuration $anRp^{+}dgotdib^{-}$

set constant created. When the variables are created, the mapping from the remaining positions to their corresponding variables is augmented with the mapping to constants. This leads to a mapping from `Var` to `DDNode` that is transparent to the predicates with some of the `Var` mapping to actual BDD variables while others map to BDD constants.

```
import qualified Data.Map as M
import qualified Data.Set as S

vars :: S.Set Var
globbed_const :: M.Map Var Bool
manifested_vars = vars S.\\ M.keysSet globbed_const
var_to_int vars el = fromJust $ elemIndex el (S.toAscList manifested_vars)
var_to_int' = var_to_int manifested_vars
ithVar' = ithVar manager
get_node :: Var -> DDNode
get_node var = case M.lookup var globbed_const of
                    Nothing -> (ithVar' . var_to_int') var
                    Just b -> boolconst b
```

Listing 43: Augmenting the variable lookup-table with constants

The preset variables need to also be injected into the models so that the operation is transparent to the interpreting function.

```
int_to_var = (!!) . S.toAscList
int_to_var' = int_to_var manifested_vars
get_solution :: [SatBit] -> [(Var, Bool)]
get_solution bits = M.toList globbed_const ++ zipWith
                (\ idx b -> ( int_to_var' idx , (head . expand) b)) [0..] bits
```

Listing 44: Injecting fixed variables into solutions

Figure 18 shows an example of what preset positions look like using the table-interpretation of a rewriting rule. A single side of a rule is, with regards to the implementation, still of type `[[DDNode]]`, even with some variables replaced by constants. This is transparent to the implementation of the predicates and functions.

	l_0	l_1	$\rightarrow$	r_0	r_1	r_2
a	$l_{0,a}$	$l_{1,a}$		1	0	$r_{2,a}$
b	$l_{0,b}$	$l_{1,a}$		0	1	$r_{2,b}$
c	$l_{0,c}$	$l_{1,a}$		0	0	$r_{2,c}$

Figure 18: Preset positions example for systems with $|l| = 2, |r| = 3, |\Sigma| = 3$ and $l = ab*$

If the preset positions do not match the encoding, the consistency predicate will be a contradiction.

There are two (not exclusive) ways to eliminating variables. One can replace a variable whose else-edge leads to 0 (or then-edge leads to 1) by 1/0 respectively (see Figure 19). This does not change the number of solutions.

If one replaces a BDD node with a constant where both edges do not lead into sinks, the number of solutions is decreased. To calculate all solutions, the BDD has to be constructed twice both with the node replaced with 1 and 0 (see Figure 20).

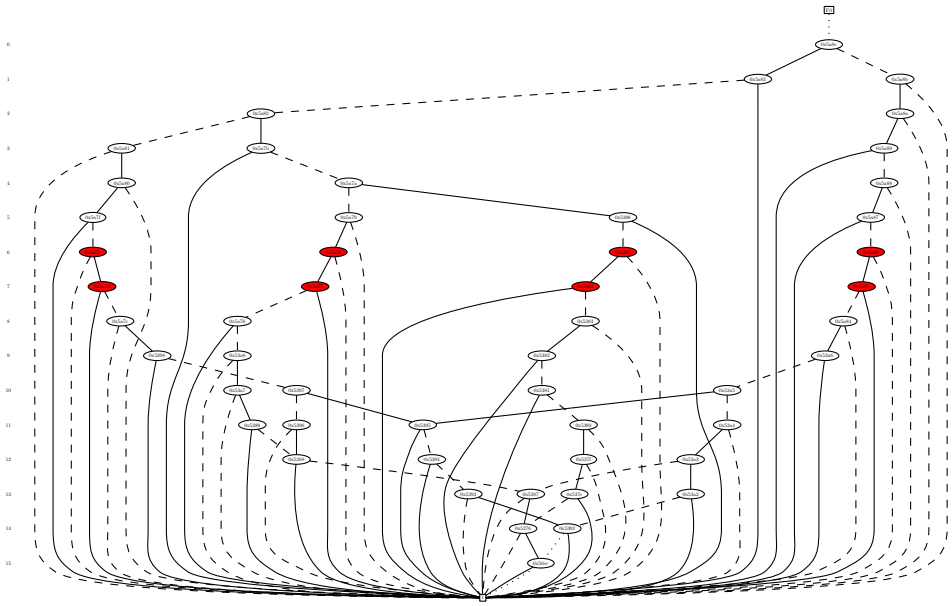
To establish what amount of resources, if any, is saved by eliminating variables without changing the set of models, we have to decide on configurations and shapes that lend themselves to variable elimination. Including "canonical after renaming" and "bordered" in the positive in the main conjunction for the system $l \to r$ allows us to eliminate all the variables for the first letter of both l and r as well as the third variable for the second letter of l and r if the alphabet is $|\Sigma| = 3$. Having $|\Sigma| = 3$ and the aforementioned predicates allows us to eliminate up to 7 variables since we know that r has the form `a*` and since the system is bordered, l has as well. Since r is canonical after renaming, we can further refine r to `a[ab]*`. We take the configuration as $Rnab^{+}got^{-}$ and the shapes of the systems as $S_1 = (3, 6, 9)$ (3247 models, 45 variables) and $S_2 = (3, 6, 11)$ (124 571 models, 51 variables).

$\|ev\|$	l	r	S_1 (t)	S_1 (m)	S_2 (t)	S_2 (m)
0	`*`	`*`	12.090	219 760	57.710	742 732
1	`*`	`[ab]*`	6.170	140 716	40.700	531 600
1	`*`	`?[ab]*`	6.447	146 448	41.437	554 180
2	`[ab]*`	`[ab]*`	2.570	91 564	18.941	298 296
2	`*`	`[ab][ab]*`	2.662	85 520	24.282	364 880
3	`*`	`a*`	2.722	87 800	25.788	371 040
3	`[ab]*`	`[ab][ab]*`	1.053	63 592	9.723	216 604
4	`*`	`a[ab]*`	1.024	61 204	15.060	260 728
4	`[ab]*`	`a*`	0.974	63 000	9.636	208 120
5	`[ab]*`	`a[ab]*`	0.580	46 216	7.334	153 368
6	`a*`	`a*`	0.586	49 096	3.179	129 500
7	`a*`	`a[ab]*`	0.357	37 348	2.050	99 340

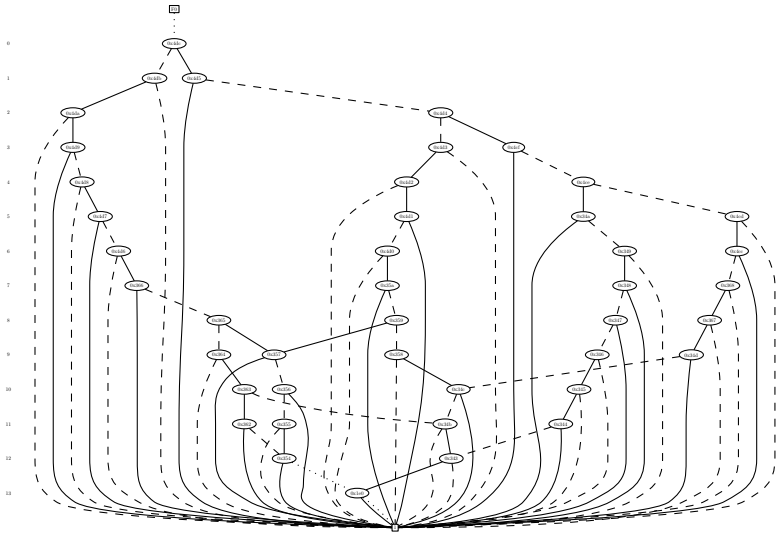
$|ev|$ - number of eliminated variables
time (t) in seconds, memory (m) in megabyte

Table 6: Impact of variable elimination without changing the set of models

As we can see from Table 6, eliminating variables has indeed a positive impact on the resource usage of the construction of the main conjunction. For the shape S_1 the memory usage was cut to up to a sixth and the needed time was cut to up to about a thirtieth. Even for the common case that only the right-hand side of a rule can be fixed to `a[ab]*`, the time was cut by a factor of 6. For the shape S_2 the improvement was smaller but for the common case it was still a time improvement by a factor of 4.

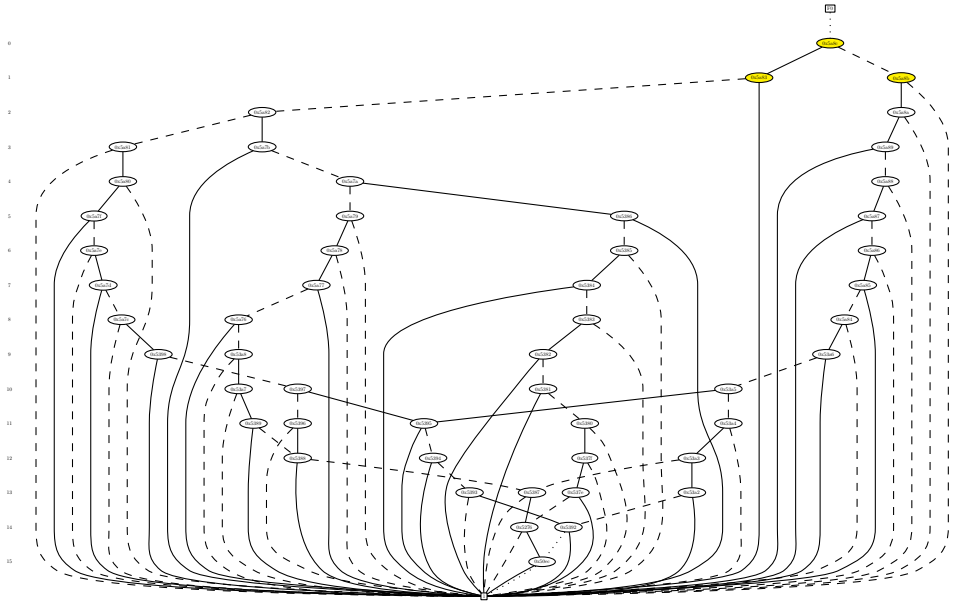


Without glob-patterns (7 models)

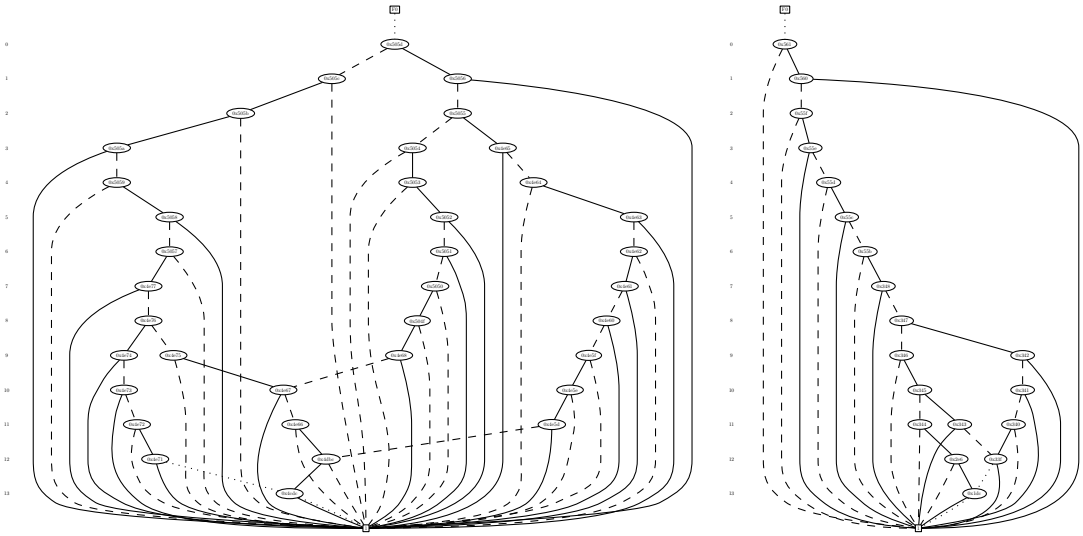


Fixed right-hand side to `a*` (7 models)

Figure 19: Fixing variables without changing the number of models



Without glob-patterns (7 models)



`a*` (4 models) `b*` (3 models)

Fixed left-hand side

Figure 20: Fixing variables, changing the number of models

For some configurations, variables were eliminated while no improvement was detectable. Going from the glob-patterns for (l, r) (`[ab]*`,`[ab][ab]*`), eliminating 3 variables, to (`*`, `a[ab]*`), eliminating 4 variables, led to no improvements at all. This implies that different BDD variables have different impact on the running time of the calculation of the main conjunction, even if both variables can be fixed without changing the set of models. In any case, variables should be fixed where possible. If rules are canonical after renaming (see Section 6.1.2), the right-hand side of the rules can and should be fixed to `a[ab][abc][abcd]...*`.

To check for any improvements gained by splitting the calculation of the main conjunction into several chunks, we again take a configuration with many predicates ($anRp^{+}dgotdib^{-}$) and the shapes $S_1 = (2, 9, 15)$ and $S_2 = (3, 6, 11)$. We fix the prefix for the left-hand sides to all possible prefixes of length 1, 2, and 3 (for S_1). The results can be found in Table 7.

The elimination of variables did not lead to any noticeable improvement for the shape S_1 while there was a huge improvement for the shape S_2. For S_2 the speed improvement of splitting the BDDs was large enough that calculating the split BDDs one after another was consistently faster than calculating the whole BDD for the pattern `*` all at once.

For the 1 286 277 systems for S_1 there were only 8242 unique right-hand sides, while for S_2 there were 11 555 unique right-hand sides for 496 811 systems. This result suggests that for the shape S_1 a large part of the BDDs is shared across prefixes leading to only little improvement, because the shared parts need to be calculated again for every process. While for the shape S_2 the BDDs for the different prefixes only share few nodes which improves parallelizability of the process.

Independent of whether the improvement for any special case is as stark as with the S_2 example, the generation of systems can be split across multiple processes or systems. This is useful when any post-filtering methods are applied (see Section 8.1).

7.6 Time and Memory Usage

While any practical computation device is constrained in its operation both by time and memory, it is not yet clear whether it is necessary for the presented use cases to track both or whether one is merely a function of the other. To take a deeper look into this questions, all measurements from the previous sections have been plotted into one scatter plot with the time taken to construct the main conjunction on the x-axis and the taken memory on the y-axis (Figure 21). Both axis are logarithmic.

In the graph we can see a horizontal line of points at about 16MB, which is the constant memory usage observed in Figure 16 for the growth with few predicates (see Section 7.4). We can also see that the ratio of the logarithm of the memory consumption and the logarithm of the time usage seems, at worst, to be constant (or growing slightly). If it is constant, the memory consumption is bounded by a function $m = e^{\frac{\log(t)}{c}}$ for time t, memory m, and some constant c.

$\|ev\|$	l	$\|S_1\|$	S_1 (t)	S_1 (m)	$\|S_2\|$	S_2 (t)	S_2 (m)
0	*	1 286 277	15.290	99 472	496 811	29.021	584 328
2/3	a*	441 902	14.620	68 336	84 738	4.157	160 020
2/3	b*	844 375	14.319	64 012	199 113	3.992	162 672
-/3	c*				212 960	3.930	163 140
sum		1 286 277	28.939	132 348	496 811	12.079	485 832
max		844 375	14.620	68 336	212 960	4.157	163 140
4/6	aa*	166 937	13.992	48 768	13 288	0.809	61 292
4/6	ab*	274 965	14.307	51 176	29 431	0.902	65 700
-/6	ac*				42 019	0.867	66 184
4/6	ba*	504 251	15.666	50 760	68 846	0.873	66 664
4/6	bb*	340 124	14.068	46 864	44 272	0.834	64 800
-/6	bc*				85 995	1.004	66 940
-/6	ca*				81 956	1.043	65 880
-/6	cb*				91 584	1.048	66 536
-/6	cc*				39 420	0.834	65 528
sum		1 286 277	58.033	197 568	496 811	8.214	589 524
max		504 251	15.666	51 176	91 584	1.048	66 940
6/-	aaa*	60 260	13.894	39 032			
6/-	aab*	106 677	14.382	42 120			
6/-	aba*	130 276	15.203	40 860			
6/-	abb*	144 689	13.898	43 660			
6/-	baa*	272 061	14.264	43 396			
6/-	bab*	232 190	14.223	41 152			
6/-	bba*	208 349	13.934	41 176			
6/-	bbb*	131 775	13.907	40 124			
sum		1 286 277	113.705	331 520			
max		272 061	15.203	43 660			

$|ev|$ - number of eliminated variables
time (t) in seconds, memory (m) in megabyte

Table 7: Impact of variable elimination, changing the set of models

Ignoring the cases for which the memory requirements seem constant, time and memory consumption during construction of the main conjunction seem to not be independent of each other. This is also the intuitive understanding of the process in that constructing many objects that take a lot of memory takes a lot of time.

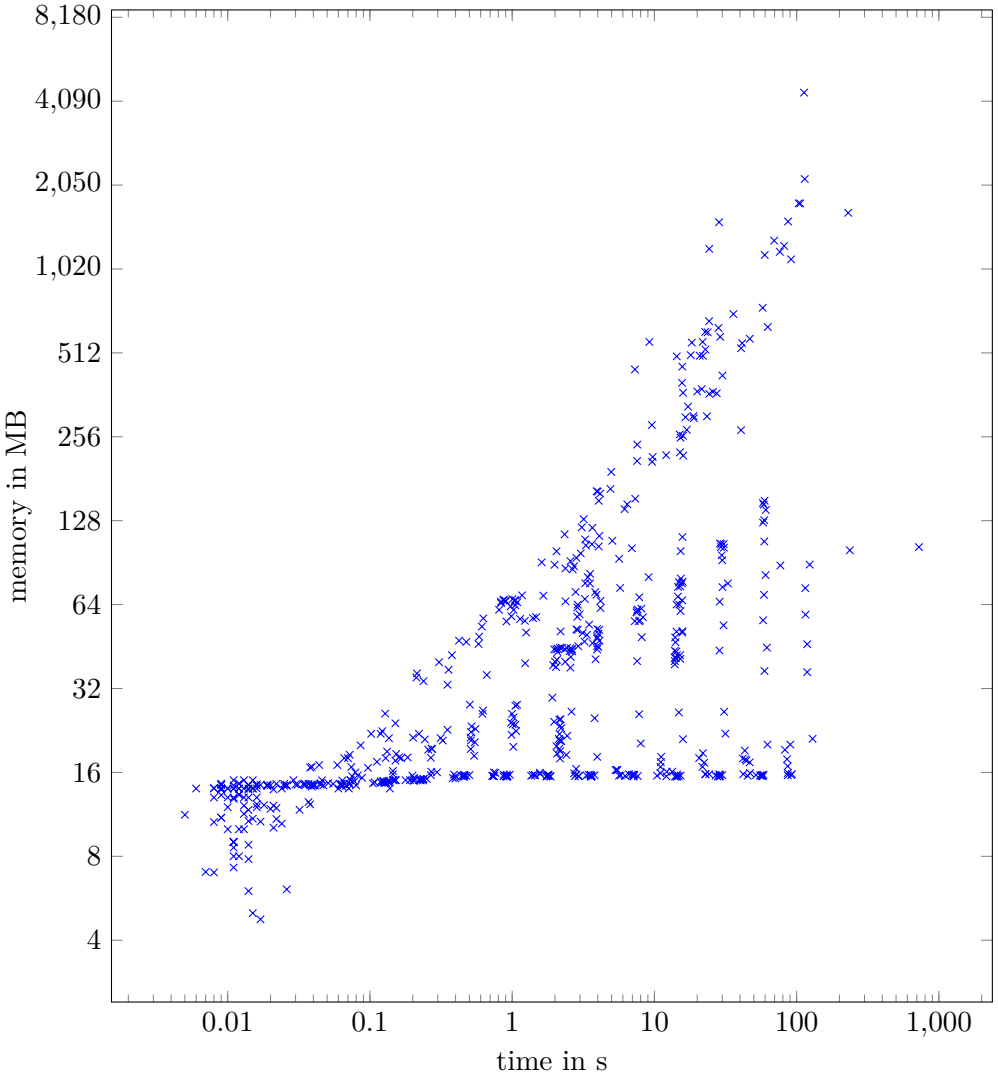


Figure 21: Growth of resource usage with growing shape for configuration $an^{+}gi^{-}$

7.7 Generating Systems

During the preceding measurements, only the construction of the main conjunction was measured and the number of models counted (see Section 3.4). No model was actually generated. Generating models can be done in linear time to the number of models in the BDD. The question that remains is, how much time is used actually generating systems, and applying the decode function μ^{-1} (see Section 4.1). To do this, several things are measured or calculated:

- $|S|$-The number of models for the shape with the given configuration

- t - The time it takes to construct the main conjunction without generating systems, using the parameter `-results 0`.
- $t+$- The time it takes to construct the main conjunction including generation of all systems. The actual output is piped to `/dev/null` minimize the impact of io operations on timing.
- $\Delta t = (t+) - t$
- $v = \frac{|S|}{\Delta t}$ in $\frac{1}{s}$ or Hz.
- io - The write speed in megabytes per second as the ratio of Δt and the amount of data written to `/dev/null` as given by `pv` [Woo15]. This is useful to see in order to check whether generating systems, without post-processing, is blocked by slow io.

We take the configuration $anRp^{+}dgotdib^{-}$ from previous measurements (with the glob-pattern `a*` for r) and use a few larger shapes to have a significant amount of systems to measure. As we can see from Table 8 that generating systems takes about a second per 30 000 to 40 000 models. Depending on the size of the systems, for our cases this works out to about 0.8 to 1.2 megabytes of io operations per second. In general every system $l \to r$ generates $|lr| + 8$ bytes of io. If the systems are written to a disk, the disk should allow writing as fast as this so that the writing operation does not block the generation of systems. The system generation is CPU bound. If the CPU is faster and the systems are written to disk, the disk should be faster as well.

S	$\lvert S\rvert$	t	$t+$	Δt	v	io
$(2, 9, 15)$	1286277	14.853	46.934	32.081	40095	1.225
$(2, 10, 15)$	2197888	28.717	86.062	57.345	38327	1.207
$(2, 9, 16)$	2792104	28.175	100.534	72.359	38587	1.215
$(2, 13, 16)$	17911528	449.038	1014.485	565.447	31677	1.118
$(3, 6, 11)$	496811	6.906	19.137	12.231	40619	0.965
$(3, 7, 11)$	859429	40.702	64.815	$24,113$	35642	0.883
$(3, 6, 12)$	2009307	14.477	71.855	57.378	35019	0.868
$(3, 7, 12)$	4281527	83.684	199.87	116.186	36851	0.947

time (t, $t+$, Δt) in seconds, speed (v) in systems per second,
write speed (io) in megabyte per second

Table 8: Generating systems, opposed to just counting

8 Finding Systems

In this section we recap the orchestrated sieving of systems that took place and present the termination problems we found that could not be solved by automated termination provers and are, by our definition, most interesting.

8.1 Counting Systems

As a configuration we take all predicates besides the subset property (see Section 6.2.4), which filters a subset of the systems that are filtered by the grid property. The whole alphabet property is included in the main conjunction as well (see Section 5.3). The configuration is $Rnpa^{+}igzotbd^{-}$. For alphabet sizes larger than two, the Sénizergeus criterion is not applied (see Section 6.2.6). All useful shapes (s, l, r) are the integer solutions to the following inequalities:

$$2 \leq s \tag{57}$$
$$s \leq l \tag{58}$$
$$3 \leq l \tag{59}$$
$$l + s \leq r \tag{60}$$

All other shapes are not "useful" for the rewriting system $l \to r$ with the given configuration.

- If $s < 2$ then there's only one letter in the alphabet and the termination problem is trivially solved by counting the lengths of the right-hand side and left-hand side of the system.
- If $l < s$ then not all letters of the alphabet Σ are in l. But since we assumed the whole alphabet property, there has to be a letter in r that is not in l. Since we also assumed that there is no inhibitor, for shapes $l < s$, there are no systems with that property.
- If $l < 3$ then the left-hand side is either of length 1 or 2 and fits Sénizergues' criterion (see Section 6.2.6).
- If $r < l + s$ then either $r \leq l$ then we have a simple termination via counting the lengths of the sides of the rule or if the lengths are equal, checking whether both sides are equal. No interesting system has the property $r \leq l$. Otherwise we have the case $r > l$ but $l + s \leq r$ which means that there is at least one letter in r that occurs equally as often or more often in l. Since for every letter in the alphabet the number of occurrences in r has to be larger than the number of occurrences in l, to not satisfy the grid property, no interesting system has the property $l + s > r$.

Since we're looking for systems that are canonical after renaming (see Section 6.1.2), we can also fix the first letter of the right-hand side to `a` and the second letter to `[ab]`, eliminating a few variables.

We want to find the shortest systems that do not fit the criteria that imply any particular decision procedure for the termination problem. We define an order over systems $l \to r$ as the order over the length of the right-hand side, then the length of the left-hand side and then the lexicographical order of rl.

$$\begin{aligned} |l_1 \to r_1| \leq |l_2 \to r_2| \iff & ((|r_1| < |r_2|) \\ & \vee (|r_1| = |r_2| \wedge |l_1| < |l_2|) \\ & \vee (|r_1| = |r_2| \wedge |l_1| = |l_2| \wedge r_1 l_1 \leq r_2 l_2)) \end{aligned} \quad (61)$$

Using this ordering we can start with the shapes $(2, 3, 5)$, $(2, 3, 6)$, $(3, 3, 6)$, and $(2, 4, 6)$ to look for small systems. With these shapes we find 23 systems that do not fit any criteria. They are:

$$\begin{gathered} aba \to abbaab,\ aba \to aabbab,\ cba \to abcacb, \\ cba \to abcabc,\ cba \to abaccb,\ cba \to aabccb, \\ cba \to aabcbc,\ cba \to aabbcc,\ cab \to abcbca, \\ bca \to abccab,\ bca \to abaccb,\ bca \to abacbc, \\ bca \to aabccb,\ abaa \to aaabab,\ baba \to aaabbb, \\ abba \to aababb,\ baab \to aabbab,\ baba \to aabbab, \\ baab \to aabbba,\ baba \to aabbba,\ baab \to ababba, \\ abba \to ababab,\ baba \to abbaab \end{gathered}$$

Trying to replicate the results from Kurth and Geser, we give a few numbers for specific sizes of the right-hand side for the given configuration. We include the subset property where "grid" is not included in the negative. Since we don't include the shapes that we do not consider useful, the actual numbers are different:

|r|	shapes	$Rnas^+$	$Rnas^+g^+$	Rna^+g^-	$Rna^+g^-o^+$
5	1	66	47	19	11
6	3	1625	1338	287	118
7	5	18 206	15 402	2804	792
8	8	395 908	363 364	32 544	5002
9	11	6 642 400	6 234 959	407 441	31 206
10	15	173 513 016	167 593 735	5 919 281	201 296

		$Rna^+go^-t^+$	$Rna^+got^-d^+$	$Rna^+gotd^-b^+$	$Rnpa^+igzotdb^-$
5	1	1	1	2	0
6	3	12	71	20	23
7	5	36	1121	167	350
8	8	152	18 500	1518	3943
9	11	478	279 799	15 278	42 191
10	15	1752	4 563 247	173 344	486 493

Table 9: Number of systems for certain configurations for a given size of the right-hand side

In the traditional sieve the alphabet size is set to the length of the right-hand side, which is too large considering the shapes we consider "useful". The canonical-after-renaming-and-reversal predicate (see Section 6.1.3) has exponential run time using the `permutations` function, so it isn't useful for larger alphabets. The sieve also generates all left-hand sides that are smaller than the right-hand side, even left-hand sides that are just smaller by one, leading to a large amount of systems that are grid systems. We also had to restart the application for every configuration, instead of counting and sieving the systems of different properties in one run. This is why it is not easy to replicate the actual results of Kurth and Geser, while we still can find the systems that we consider interesting.

Using the shapes $(7,1,7),(7,2,7),(7,3,7),(7,4,7),(7,5,7),(7,6,7)$ (without the whole alphabet predicate from Section 5.3), we can indeed reproduce some numbers from Geser [Ges02, Table 6.1] for $n = |r| = 7$. For $n = |r| = 8$ this is already expensive.

	configuration	Geser	Wenzel	restricted shapes
Total	Rns^+	3 151 054	3 151 054	18 206
grid	$Rnsg^+$	3 145 038	3 145 038	15 402
non-grid	Rn^+g^-	6016	6016	2804
factor	$Rn^+g^-o^+$	3004	3004	792
2-loop	$Rn^+go^-t^+$	57	57	36
Crit. D	$Rn^+got^-d^+$	1869	1869	1121
bordered	$Rn^+gotd^-b^+$	215	215	167

Table 10: Reproduced numbers from Geser [Ges02, Table 6.1] for $n = 7$

In a concerted enumeration effort, computers from the *HTWK Leipzig* were used to enumerate systems of up to right-hand side size 14 with a varying alphabet size. The University's computers had octacore *Intel Core i7-6700* processors and 16GB of RAM. All prefixes of length 6 for alphabet size 2, and size 5 for alphabet size 3 and more, were generated using the canonical right-hand side generator from Listing 1. These prefixes were used as a glob pattern for the right-hand side (see Section 7.5), eliminating variables and splitting the computation across multiple jobs. These prefixes were combined with all useful shapes up to a length of the right-hand side of 14, creating 6161 job files of the form `../dist/build/srs-count/srs-count -globright "abbbb*" -R True -n True -p True -a True -i False -g False -o False -t False -b False -matchbox maybe -results -1 3 10 13` for the file `r13-l10-s3-abbbb.cfg`, that can be run from an outside shell script.

The predicate for Criterion D (see Section 6.2.3) was not included, since it was a recent addition to the suite at that time. The Seńizergues criterion (see Section 6.2.6) is included for shapes that have an alphabet size of 2.

In order to not write all systems to a shared network storage, we included a filtering process with an external solver that was run after the main conjunction was generated and the number of solutions was counted and before the systems were written to `stdout`. The solution iteration from Listing 10 was augmented to call external termination tools, parsing the output and filtering the systems for specific results. Bindings for *ttt2* [Kor+09] and *matchbox* [Wal04] were written using this method. Matchbox was chosen because it is quite fast and ttt2 because it is well known. Bindings to the tool *AProVE* [Gie+04] were considered as well but not implemented because AProVE does not fail gracefully in that it might just terminate printing exceptions without giving any proper output on the termination properties of the input. The post-filtering can be applied using the parameters `-matchbox` and `-ttt2` with the expected result of the tool as the argument.

```
mkdir -p "processing"
mkdir -p "fin"
mkdir -p "results"
while :
do
  CFG=`ls *.cfg | sort -R | head -n 1`
  if [ -z "$CFG" ]; then
    exit 0
  fi
  mv $CFG processing || continue
  ulimit -v 16000000; bash processing/${CFG} &> /tmp/${CFG}.out
  ulimit -v unlimited
  mv /tmp/${CFG}.out results
  mv processing/${CFG} fin
done
```

Listing 45: Processing job files

Using the processing script from Listing 45, a random job was run on a computer, limiting the allowed virtual memory of the job to 16 000 000 Bytes. This was done to prevent swapping behaviour and to also not impact the normal operation of the computers too much, since they were used at the time for teaching purposes. Using this process, we were able to generate all systems with the given properties for all shapes up to a right-hand side of size 14 and an alphabet of size 3 or less (see Table 11). Increasing the alphabet size to 4, we were able to generate all systems with a length of the right-hand side of up to 13.

l\r	4	5	6	7	8	9	10	11	12	13	14
3	-	2	3	3	3	3	3	3	3	3	3
4	-	-	**2**	3	**4**	4	**4**	4	**4**	4	**4**
5	-	-	-	**2**	**3**	**4**	**5**	**5**	**4**	**4**	**4**
6	-	-	-	-	**2**	**3**	**4**	**5**	**4**	**4**	**3**
7	-	-	-	-	-	**2**	**3**	**4**	**4**	**4**	**3**
8	-	-	-	-	-	-	**2**	**3**	**4**	**4**	**3**
9	-	-	-	-	-	-	-	**2**	**3**	**4**	**3**
10	-	-	-	-	-	-	-	-	**2**	**3**	**3**
11	-	-	-	-	-	-	-	-	-	**2**	**3**
12	-	-	-	-	-	-	-	-	-	-	**2**

The cells that are filled green have been enumerated fully up to the stated alphabet size and there is no useful larger alphabet size. The cells that are filled red have been enumerated fully up to the stated alphabet size take one and enumeration for the stated alphabet size ran out of resources for some prefixes. The cells that aren't filled have been fully enumerated up to the stated alphabet size and there were larger alphabets that would have been useful but for all prefixes all larger alphabet enumerations ran out of resources. The cells that are bold contain systems that matchbox couldn't solve.

Table 11: Maximum size of alphabet for which enumeration was possible

All in all we generated 10 977 553 715 systems, with 681 systems remaining after the first filtering using matchbox. The calculation of the main conjunctions without the post-processing and actually generating the systems took 714 562 seconds (around 200 hours), not including the jobs that failed because they ran out of resources. The longest job took 5.5 hours to construct the main conjunction and it was to calculate it for the shape $(4, 13, 9)$ and the prefix `abbba` for the right-hand side of the rules. 43 jobs took more than an hour to construct the main conjunction, with an overall average of more than two minutes per main conjunction.

In Table 12 we see the number of systems generated for each alphabet size and length of right-hand side. Table 13 shows the same for the systems not filtered out by matchbox.

$\lvert\Sigma\rvert$ \ $\lvert r\rvert$	2	3	4	5	6
6	21	36	-	-	-
7	183	773	-	-	-
8	1135	10 837	949	-	-
9	5841	120 021	45 948	-	-
10	27 368	1 176 875	1 219 852	49 264	-
11	119 144	10 759 147	23 843 662	2 634 073	-
12	498 780	94 620 672	394 681 800	0	0
13	2 027 720	814 792 345	2 426 080 304	0	0
14	8 109 965	6 728 946 312	467 780 687	0	0

Table 12: Number of systems generated for that alphabet size and size of right-hand side

$\lvert\Sigma\rvert$ \ $\lvert r\rvert$	2	3	4	5	6
6	1	0	-	-	-
7	1	0	-	-	-
8	6	0	0	-	-
9	10	1	0	-	-
10	23	4	0	0	-
11	30	9	0	0	-
12	68	35	3	0	0
13	98	58	7	0	0
14	175	152	0	0	0

Table 13: Number of systems not filtered out by matchbox for that alphabet size and size of right-hand side

The smallest systems matchbox could not filter out at that time (within the given limit of one second) were the following:

$$abaa \to aaabab,\ abaaa \to aaaabab,\ abaa \to aaababab,$$
$$abaaa \to aaaabaab,\ abbaa \to aaabbabb,\ abaaaa \to aaaaabab,$$
$$ababaa \to aaababab,\ abaaba \to aababaab$$

8.2 The Hard Cases (after handing them off) / Post-Processing

As a last step, the 681 systems from the previous section were uploaded to StarExec [SST12], a computation cluster of the *University of Iowa*, that is used by a variety of logic solving communities to test *solvers*, in our case ttt2 and AProVE proving (non)termination, on *benchmarks*, in our case the string rewriting systems. The tools ttt2 and AProVE were run under conditions similar to those of the termination competi-

tion 2015 [TP16]. From the 681 systems remained 226 that still had an open termination problem.

In the run-up to the termination competition 2016, those 226 systems were filtered again using the newest version of matchbox that was available at that time. Only the following 17 systems remained that were submitted to the 2016 termination competition:

$$babbabbaba \to ababbabbabba,\ abbcacabca \to abcacabbcacab,$$
$$abbaaaabaa \to abaaaabbaaaab,\ bababbababa \to abababbababba,$$
$$ababaabaaba \to abaabaabaabab,\ babaaaaaaa \to aaaaaaababbaba,$$
$$babbabbaba \to abababbabbabba,\ bacaaaaaaa \to aaaaaaabacbaca,$$
$$bbcaaaaaaa \to aaaaaaabbcbbca,\ bcbaaaaaaa \to aaaaaaabcbbcba,$$
$$bccaaaaaaa \to aaaaaaabccbcca,\ baababababa \to abababbaababab,$$
$$aabbaaaabaa \to aabaaaabbaaaab,\ ababbbabbab \to abbabbbababbba,$$
$$aabaababaab \to aababaababaaba,\ abababaababa \to abaababaababab,$$
$$abababaababa \to ababaababaabab$$

After the termination competition 2016, following two systems still have no automated (non)termination proof[2] by any of the participating applications:

$$ababbbabbab \to abbabbbababbba,\ abababaababa \to ababaababaabab$$

8.3 Extension to Cycle Termination

The approach presented for generating rewriting systems in this thesis is easily applicable to the generation of cycle rewriting systems. Cycle rewriting is an extension to regular string rewriting where besides the normal rewriting reduction another reduction is possible. For any word $w = w_0w_1w_2 \dots w_n$ it is allowed to rotate the letters so that $w \to^o w'$ with $w' = w_nw_1w_2 \dots w_{n-1}$ [ZKB14]. This operation can be applied more than once and in the reverse as well (even though it is not necessary).

We can simply use our existing implementation, and switch off a few criteria. From the list of predicates (see Figure 13), we omit the following because their applicability needs further research: Kurth's criterion D, the grid criterion, and Sénizergues' criterion.

We can use the inhibitor criterion for reduction. If a rewriting system $l \to r$ has an inhibitor, then cycle termination of $l \to r$ is equivalent to standard termination of $l \to r$, which is (in that case) decidable. Once an inhibitor is produced it can not be used to do further reduction steps and the string is basically "cut" at that point. So even including the rotating relationship, there is no longer the name giving cycle.

We additionally include another predicate "letter counting" that has a similar implementation to the grid property predicate (see Section 6.2.5) but excludes the case where $|l|_a = |r|_a$ so that we can still filter out cases where $|l|_a > |r|_a$ for some a. It is also necessary to relax the conditions on useful shapes, to accomodate the removed grid predicate.

[2] `[http://termcomp.imn.htwk-leipzig.de/results/standard/noquery/18369]`

Cycle systems have been filtered up to $|r| = 9$ completely and for $|r| = 10$ incompletely. After that, the systems have been filtered using matchbox. These 13 systems were the systems that remained and we consider "interesting".

$$baba \to abaaabab,\ baba \to abaaaabab,\ abaaba \to aababaab,$$
$$ababba \to aabbabab,\ baba \to abaaaaabab,\ abaaba \to aababababaab,$$
$$ababba \to aabbababab,\ ababba \to abaabbabab,\ ababbab \to abbababba,$$
$$baaba \to aabaaabaab,\ baaba \to abaaaabaab,\ baabba \to aabbaaabb,$$
$$bababa \to abaaababab$$

After the termination competition 2016, following three systems still have no automated (non)termination proof[3] by any of the participating applications:

$$abaaba \to aababaab,\ abaaba \to aababababaab,\ ababbab \to abbababba$$

[3] [http://termcomp.imn.htwk-leipzig.de/results/standard/noquery/18370]

9 Conclusion

We have presented an approach for encoding single rule string rewriting systems and their properties as Boolean variables and formulas (Section 4), with an implementation that is straightforward, using the Haskell language and the Binary Decision Diagram library CUDD (Section 4.3). The implementation does not allow us the use quantifiers as they are, leading us to expand every quantification, which might not be practical in all cases. We also can not use additional memory besides constructing additional Boolean formulas over the variables used to encode the rewriting systems.

We were still able to construct quite a few predicates to filter out systems that are not a canonical representative or have a decision procedure for their termination problem (Sections 6.1 and 6.2). Some predicates that we constructed had a bad impact on the resource usage while other predicates' impact was almost negligible, with some predicates in combination being even faster than either predicate on its own (Section 7.1). We have also seen that variable ordering and even order of construction can have a huge impact on the run time of a BDD construction (Sections 7.3 and 7.2). We have not been able to give heuristics on how to estimate beforehand how a given predicate, variable ordering or construction order might performs in comparison to other orderings and implementations.

While we have not directly compared an implementation of the traditional sieve to the BDD approach, the results suggest that there is indeed an improvement. The generate and filter approach is foremost a combinatorial problem that is bounded by an exponential function with the alphabet size as the base ($\Sigma^{|lr|}$). We've seen (Section 7.4) that for simple predicates and larger alphabet the time usage is bounded by an exponential function $2^{|lr|}$, seemingly independent of the alphabet size. While this is already a better run time behaviour than the traditional sieve, we have already filtered out all grid systems at this point.

With more predicates the run time for at least alphabet size $|\Sigma| = 2$ was still bounded in a similar fashion. As we saw from Section 7.1, analyzing the efficiency of predicates, adding predicates does not necessarily lead to more but possible less run time and, of course, more filtered systems. In the traditional approach another predicate or filter criterion must necessarily have negative impact on the running time of the filter procedure. We've also seen that most predicates filtered proportionally more systems with growing system size, increasing in our definition of efficiency.

Therefore we can say that the BDD approach has a better run time behaviour than the traditional approach, if the filter criteria and configuration settings are chosen carefully. This approach is memory bound and any configuration or shape that leads to the construction of the main conjunction needing more memory than the host system can muster, memory swapping behaviour will surely lead to worse run times than the traditional approach. We've also presented an approach to alleviate this problem by splitting the computation over several processes (Section 7.5).

We've been able to reproduce some results from Geser (Section 8.1). While reproducing all results would not be practical with the BDD approach, we've shown that we can find interesting systems by systematically narrowing the search space. With the BDD approach we lost the ability of the traditional sieve, to count how many systems are filtered by which stage or criterion of the sieve. These numbers can be reproduced with our implementation by deliberately applying very specific filter criteria, but in most cases the interesting problems are those that don't fit any criteria.

Using the BDD implementation, several desktop machines were used to generate all interesting systems with a right-hand side size of up to 14 and an alphabet size of up to 3, with larger alphabets in some cases, but not exhaustively. Those systems, about 11×10^9, were filtered with matchbox, a termination prover. The remaining 681 systems were filtered using ttt2 and AProVE on the StarExec computational cluster. The remaining 226 systems were filtered again using a newer version of matchbox (leaving 17 problems open) and sent in as competition problems to the Termination Competition 2016, which left 2 problems open (Section 8.2).

This approach was extended to cycle termination and we can present 3 rewriting systems that have no automated cycle termination proof and have a right-hand side that is not larger than 10 (Section 8.3).

While the BDD approach is already useful, there are several ideas for possible improvements that lend themselves to further research:

The chosen encoding is not optimal with regards to how many variables are used for a given shape. A binary encoding would need less variables for a given shape (see Section 4.2). Going from one-hot encoding to binary encoding is possible without rewriting the existing μ-compatible predicates. Since variables and functions within a BDD are both nodes and interchangeable, functions that map the full binary encoding to one-hot can be substituted in place of the original variables (see Figure 22 for an example).

	l_0	l_1	$\rightarrow$	r_0	r_1	r_2
a	$\overline{l_{0,0}} \wedge \overline{l_{0,1}}$	$\overline{l_{1,0}} \wedge \overline{l_{1,1}}$		$\overline{r_{0,0}} \wedge \overline{r_{0,1}}$	$\overline{r_{1,0}} \wedge \overline{r_{1,1}}$	$\overline{r_{2,0}} \wedge \overline{r_{2,1}}$
b	$\overline{l_{0,0}} \wedge l_{0,1}$	$\overline{l_{1,0}} \wedge l_{1,1}$		$\overline{r_{0,0}} \wedge r_{0,1}$	$\overline{r_{1,0}} \wedge r_{1,1}$	$\overline{r_{2,0}} \wedge r_{2,1}$
c	$l_{0,0} \wedge \overline{l_{0,1}}$	$l_{1,0} \wedge \overline{l_{1,1}}$		$r_{0,0} \wedge \overline{r_{0,1}}$	$r_{1,0} \wedge \overline{r_{1,1}}$	$r_{2,0} \wedge \overline{r_{2,1}}$
d	$l_{0,0} \wedge l_{0,1}$	$l_{1,0} \wedge l_{1,1}$		$r_{0,0} \wedge r_{0,1}$	$r_{1,0} \wedge r_{1,1}$	$r_{2,0} \wedge r_{2,1}$

Figure 22: Using existing one-hot compatible predicates with full binary encoding

While this makes the existing predicates available for other encodings, those predicates might not be optimal for the chosen encoding. For example, an alphabet that has a size of

a power of two, the one-hot/consistency predicate (see Section 5.1) would be a tautology. Similarly equals or less-than for letters (see Section 5.4.1) would be easier to implement on the variables directly instead of the exponentially many functions translating to one-hot encoding. Other predicates might likewise have implementations that are much better on the BDD variables and the binary encoding directly instead of the mapped one-hot encoding.

Adapting the existing codebase for full binary encoding might therefore not be hard but the predicates need to be adapted to the new encoding scheme as well to get the best results in terms of resource usage.

The interface to the application leaves much room for improvement. A given shape is the base for any main conjunction construction. To get comparable results to the traditional sieve of Kurth's, the calculation has to be started many times with different shapes. This can be automated.

During the search for difficult problems, a memory limit was enforced to fit the whole calculation into memory without having to use swapping mechanisms. In order to still enumerate longer systems, prefixes have been fixed to reduce the number of variables in the calculation. The calculation was then done multiple times for each permutation of the fixed variables. This could be automated as well. When the calculation reaches a certain memory limit, it could be stopped automatically and new processes with differently fixed variables could be put into a queue and rerun, using less memory this time. This could be repeated until the operation fits into memory, trading memory usage and time (in number of calculation runs) bit by bit.

After every run the application is stopped and run with different parameters. There is no sharing of functions from the BDD base over different runs. Considering the shape (s, l, r) and the shape $(s, l-1, r+1)$, then both shapes have, when the main conjunction is calculated, basically the same consistency predicate. Since both shapes use the same number of variables, the BDD base from one run could be reused as the BDD base for a different run. Many predicates for slightly different shapes would have a lot of common functions. Sénizergues (see Section 6.2.6), grid (see Section 6.2.5), and loops of length one (see Section 6.2.1) come to mind immediately. The chosen variable ordering is favorable for that since most variables keep their semantics with different shapes but equal number of variables already.

During calculation of the main BDD-conjunction, only a single core is ever used by the CUDD-library. BDD operations can be efficiently parallelized on multiple processors. A BDD implementation that uses all available processors could spread the calculation time evenly across processors. Running the calculations on 6 cores at once, one could reasonably expect a 4x speedup over the use of a single core [He09]. The parallelization of the ite-operation $ite(a, b, c)$ works in principle by evaluating b and c in parallel and introducing a thread-safe data structure to allow sharing evaluation results between the b- and c-branches. [SB96] [MH98]

The approach can be adapted to string rewriting systems with multiple rules. Fixing the system shape as rigidly as it is done with one rule would significantly increase the number of possible and necessary shapes. This can be fixed by introducing an "empty" letter that is allowed to be at the end of sides of rules of a system with the adapted consistency predicate that only allows empty letters to follow an empty letter. The shape of the system would then be configured as the maximum lengths of the left-hand and right-hand side, the alphabet, and the number of rules. With this approach the length of words is no longer statically known and, of course, all predicates need to be adapted.

This approach may well be useful for term-rewriting if there is an encoding scheme that is both useful in encoding predicates as well as not too large with regards to the number of variables needed for this encoding scheme.

List of Tables

List of Figures

Listings

References

[Ake78] Sheldon B. Akers. "Binary Decision Diagrams". In: *IEEE Trans. Computers* 27.6 (1978), pp. 509–516. DOI: 10.1109/TC.1978.1675141. URL: http://doi.ieeecomputersociety.org/10.1109/TC.1978.1675141.

[AM04] Carlos Ansótegui and Felip Manyà. "Mapping Problems with Finite-Domain Variables into Problems with Boolean Variables". In: *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*. 2004. URL: http://www.satisfiability.org/SAT04/programme/53.pdf.

[Bry86] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819. URL: http://doi.ieeecomputersociety.org/10.1109/TC.1986.1676819.

[Bry91] Randal E. Bryant. "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication". In: *IEEE Trans. Computers* 40.2 (1991), pp. 205–213. DOI: 10.1109/12.73590. URL: http://doi.ieeecomputersociety.org/10.1109/12.73590.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. "Terminator: Beyond Safety". In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. 2006, pp. 415–418. DOI: 10.1007/11817963_37. URL: http://dx.doi.org/10.1007/11817963_37.

[Dra15] Chris Drake. *pyeda (pyeda/pyeda/boolalg/bdd.py)*. [https://github.com/cjdrake/pyeda/blob/88409715b21f9f719d63ac4eed2ee10f3db09783/pyeda/boolalg/bdd.py]. 2015.

[Dra16] Chris Drake. *PyEDA: a Python library for electronic design automation.* [https://pyeda.readthedocs.io/en/latest/]. 2016.

[EWZ08] Jörg Endrullis, Johannes Waldmann, and Hans Zantema. "Matrix Interpretations for Proving Termination of Term Rewriting". In: *J. Autom. Reasoning* 40.2-3 (2008), pp. 195–220. DOI: 10.1007/s10817-007-9087-9. URL: http://dx.doi.org/10.1007/s10817-007-9087-9.

[Fou16] Python Software Foundation. *Python – a programming language that lets you work quickly and integrate systems more effectively.* https://www.python.org/. 2016.

[Ges02] Alfons Geser. *Is Termination Decidable for String Rewriting with only One Rule?* habilitation thesis at Tübingen. 2002.

[GHW04] Alfons Geser, Dieter Hofbauer, and Johannes Waldmann. "Match-Bounded String Rewriting Systems". In: *Appl. Algebra Eng. Commun. Comput.* 15.3-4 (2004), pp. 149–171. DOI: 10.1007/s00200-004-0162-8. URL: http://dx.doi.org/10.1007/s00200-004-0162-8.

[Gie+04] Jürgen Giesl et al. "Automated Termination Proofs with AProVE". In: *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings.* 2004, pp. 210–220. DOI: 10.1007/978-3-540-25979-4_15. URL: http://dx.doi.org/10.1007/978-3-540-25979-4_15.

[Gie+06] Jürgen Giesl et al. "Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages". In: *Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings.* Ed. by Frank Pfenning. Vol. 4098. Lecture Notes in Computer Science. Springer, 2006, pp. 297–312. ISBN: 3-540-36834-5. DOI: 10.1007/11805618_23. URL: http://dx.doi.org/10.1007/11805618_23.

[Glo] *glob(7) Linux User's Manual.* 2012.

[Has16] Haskell.org. *Haskell – An advanced purely-functional programming language.* https://www.haskell.org/. 2016.

[He09] Yuxiong He. *Multicore-enabling a Binary Decision Diagram algorithm.* https://software.intel.com/en-us/articles/multicore-enabling-a-binary-decision-diagram-algorithm. 2009.

[HW06] Dieter Hofbauer and Johannes Waldmann. "Termination of String Rewriting with Matrix Interpretations". In: *Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings.* Ed. by Frank Pfenning. Vol. 4098. Lecture Notes in Computer Science. Springer, 2006, pp. 328–342. ISBN: 3-540-36834-5. DOI: 10.1007/11805618_25. URL: http://dx.doi.org/10.1007/11805618_25.

[KM09] Martin Korp and Aart Middeldorp. "Match-bounds revisited". In: *Inf. Comput.* 207.11 (2009), pp. 1259–1283. DOI: 10.1016/j.ic.2009.02.010. URL: http://dx.doi.org/10.1016/j.ic.2009.02.010.

[Knu09] D.E. Knuth. *The Art of Computer Programming: Bitwise Tricks and Techniques - Binary Decision Diagrams*. Art of Computer Programming. Addison-Wesley, 2009. ISBN: 9780321580504. URL: https://books.google.de/books?id=PkynSwAACAAJ.

[Kor+09] Martin Korp et al. "Tyrolean Termination Tool 2". In: *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings*. 2009, pp. 295–304. DOI: 10.1007/978-3-642-02348-4_21. URL: http://dx.doi.org/10.1007/978-3-642-02348-4_21.

[Kur90] Winfried Kurth. "Termination und Konfluenz von Semi–Thue–Systemen mit nur einer Regel". PhD thesis. Technische Universität Clausthal, 1990.

[McN01] Robert McNaughton. "Semi-Thue Systems with an Inhibitor". In: *J. Autom. Reasoning* 26.4 (2001), pp. 409–431. DOI: 10.1023/A:1010759024900. URL: http://dx.doi.org/10.1023/A:1010759024900.

[MH98] Kim Milvang-Jensen and Alan J. Hu. "BDDNOW: A Parallel BDD Package". In: *Formal Methods in Computer-Aided Design, Second International Conference, FMCAD '98, Palo Alto, California, USA, November 4-6, 1998, Proceedings*. 1998, pp. 501–507. DOI: 10.1007/3-540-49519-3_32. URL: http://dx.doi.org/10.1007/3-540-49519-3_32.

[Pfe06] Frank Pfenning, ed. *Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings*. Vol. 4098. Lecture Notes in Computer Science. Springer, 2006. ISBN: 3-540-36834-5.

[PJ11] Justyna Petke and Peter Jeavons. "The Order Encoding: From Tractable CSP to Tractable SAT". In: *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*. Ed. by Karem A. Sakallah and Laurent Simon. Vol. 6695. Lecture Notes in Computer Science. Springer, 2011, pp. 371–372. ISBN: 978-3-642-21580-3. DOI: 10.1007/978-3-642-21581-0_34. URL: http://dx.doi.org/10.1007/978-3-642-21581-0_34.

[Rud93] Richard Rudell. "Dynamic variable ordering for ordered binary decision diagrams". In: *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993*. 1993, pp. 42–47. DOI: 10.1109/ICCAD.1993.580029. URL: http://dx.doi.org/10.1109/ICCAD.1993.580029.

[SB96] Tony Stornetta and Forrest Brewer. "Implementation of an Efficient Parallel BDD Package". In: *DAC*. 1996, pp. 641–644. DOI: 10.1145/240518.240639. URL: http://doi.acm.org/10.1145/240518.240639.

[Som15] Fabio Somenzi. *CUDD: CU Decision Diagram Package Release 3.0.0*. http://vlsi.colorado.edu/~fabio/CUDD. 2015.

[SS11] Karem A. Sakallah and Laurent Simon, eds. *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*. Vol. 6695. Lecture Notes in Computer Science. Springer, 2011. ISBN: 978-3-642-21580-3. DOI: 10.1007/978-3-642-21581-0. URL: http://dx.doi.org/10.1007/978-3-642-21581-0.

[SST12] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. "Introducing StarExec: a Cross-Community Infrastructure for Logic Solving". In: *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, Manchester, United Kingdom, June 30, 2012*. 2012, p. 2. URL: http://ceur-ws.org/Vol-873/papers/inv_1.pdf.

[Sén96] Géraud Sénizergues. "On the Termination Problem for One-Rule Semi-Thue System". In: *Rewriting Techniques and Applications, 7th International Conference, RTA-96, New Brunswick, NJ, USA, July 27-30, 1996, Proceedings*. Ed. by Harald Ganzinger. Vol. 1103. Lecture Notes in Computer Science. Springer, 1996, pp. 302–316. ISBN: 3-540-61464-8. DOI: 10.1007/3-540-61464-8_61. URL: http://dx.doi.org/10.1007/3-540-61464-8_61.

[Thi16] René Thiemann. *Termination and Complexity Competition 2016*. http://www.termination-portal.org/wiki/Termination_and_Complexity_Competition_2016. 2016.

[TP16] Termination-Portal.org. *Termination Competition*. http://termination-portal.org/wiki/Termination_Competition. 2016.

[Wal04] Johannes Waldmann. "Matchbox: A Tool for Match-Bounded String Rewriting". In: *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*. 2004, pp. 85–94. DOI: 10.1007/978-3-540-25979-4_6. URL: http://dx.doi.org/10.1007/978-3-540-25979-4_6.

[Wal10] Johannes Waldmann. "Polynomially Bounded Matrix Interpretations". In: *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK*. Ed. by Christopher Lynch. Vol. 6. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010, pp. 357–372. ISBN: 978-3-939897-18-7. DOI: 10.4230/LIPIcs.RTA.2010.357. URL: http://dx.doi.org/10.4230/LIPIcs.RTA.2010.357.

[Wal15] Adam Walker. *cudd: Bindings to the CUDD binary decision diagrams library*. https://hackage.haskell.org/package/cudd. 2015.

[Wal16] Johannes Waldmann. *obdd: Ordered Reduced Binary Decision Diagrams*. https://hackage.haskell.org/package/obdd. 2016.

[Woo15] Andrew Wood. *pv - monitor the progress of data through a pipe*. 2015.

[Wra90] Celia Wrathall. “Confluence of One-Rule Thue Systems”. In: *Word Equations and Related Topics, First International Workshop, IWWERT '90, Tübingen, Germany, October 1-3, 1990, Proceedings*. 1990, pp. 237–246. DOI: 10.1007/3-540-55124-7_11. URL: http://dx.doi.org/10.1007/3-540-55124-7_11.

[ZG00] Hans Zantema and Alfons Geser. “A Complete Characterization of Termination of $0^p1^q \to 1^r0^s$”. In: *Appl. Algebra Eng. Commun. Comput.* 11.1 (2000), pp. 1–25. DOI: 10.1007/s002009900019. URL: http://dx.doi.org/10.1007/s002009900019.

[ZKB14] Hans Zantema, Barbara König, and H. J. Sander Bruggink. “Termination of Cycle Rewriting”. In: *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Ed. by Gilles Dowek. Vol. 8560. Lecture Notes in Computer Science. Springer, 2014, pp. 476–490. ISBN: 978-3-319-08917-1. DOI: 10.1007/978-3-319-08918-8_33. URL: http://dx.doi.org/10.1007/978-3-319-08918-8_33.