First-Order Unification on Compressed Terms

Adrià Gascón[1], Sebastian Maneth[2], and Lander Ramos[1]

1 **Universitat Politècnica de Catalunya**
Jordi Girona 1-3 08034 Barcelona, Spain
adriagascon@gmail.com, landertxu@gmail.com
2 **NICTA and University of New South Wales**
Sydney, Australia
sebastian.maneth@nicta.com.au

Abstract

Singleton Tree Grammars (STGs) have recently drawn considerable attention. They generalize the sharing of subtrees known from DAGs to sharing of connected subgraphs. This allows to obtain smaller in-memory representations of trees than with DAGs. In the past years some important tree algorithms were proved to perform efficiently (without decompression) over STGs; e.g., type checking, equivalence checking, and unification. We present a tool that implements an extension of the unification algorithm for STGs. This algorithm makes extensive use of equivalence checking. For the latter we implemented two variants, the classical exact one and a recent randomized one. Our experiments show that the randomized algorithm performs better. The running times are also compared to those of unification over uncompressed trees.

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.51

Category System Description

1 Introduction

Trees are a basic and very common data structure in computer science. In many applications, trees are stored in memory for fast processing. Some recent applications deal with *very large trees*. For instance, the nowadays ubiquitous data exchange format XML stores data in the form of unranked trees; typically, each data item is accompanied by several XML tree nodes describing its structure. This results in huge trees, often consisting of many millions of nodes. The problem arises that such trees do not fit into main memory, especially if stored as conventional (machine) pointer data structure. Therefore, *compressed* in-memory representations have been developed; for instance, succinct trees (see, e.g., [20]), or grammar-compressed trees [4, 17].

Here we deal with grammar-compressed trees. Grammar compression was invented for strings in the 1990s, see [19] for a survey. The idea is to find a small grammar that generates only the given string. It is a form of dictionary compression where grammar nonterminals represent repeated substrings. For instance, a smallest *context-free (cf) grammar* that generates a given string can be (at most) exponentially smaller than the given string. Finding a minimal cf grammar is NP-complete, but several well-behaved approximation algorithms exist [5]. While in general algorithms run slower when executed over a compressed representation, there are certain special algorithms which can execute in one pass (without decompression) through the grammar. This induces a *speed-up* that is proportional to the compression. For instance, testing whether two cf grammars generate the same string can be performed in cubic time with respect to the sizes of the grammars [13].

The idea of grammar-compression was generalized from strings to trees in [4], where they present an approximation algorithm that finds a small cf tree grammar. We call a cf tree

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 51–60



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

grammar that generates only one tree, a *Singleton Tree Grammar* (STG). Note that the classical idea of representing a tree by its minimal unique DAG is an instance of grammar compression: the DAG is equivalent to the minimal regular tree grammar of the tree. For typical XML documents, DAGs allow to shrink their tree structures to about 12% of the original number of edges [3]. The algorithm of [4] finds STGs that contain only 4% of the original edges. The new grammar compressor "TreeRePair" [15] compresses even further ($< 3\%$) and runs almost as fast as building a minimal DAG.

Examples of algorithms that run efficiently (without decompression) over STGs are tree automata evaluation [14], XPath query processing [17], and equivalence testing [4, 21]. STGs have also been used for complexity analysis of unification algorithms [12]. Recently, *first-order unification* was shown to be solvable in polynomial time over STGs [9, 10]. Note that an application domain for which unification over compressed terms can be useful are logic-programming languages for XML. Examples of such languages are Xcerpt [2] (it uses a form of asymmetric unification called "simulation) and Xcentric [6] (it uses the unification studied in [11]). Here, we present an implementation of the unification and matching algorithms of [10]. The algorithms run a variant of Robinson's standard unification algorithm [18] over two given STGs; it builds string grammars for the preorder traversals of the grammars, and then applies equivalence checking for singleton cf string grammars, while instantiating the encountered variables. For the equivalence check we implemented two competing algorithms: (1) the exact algorithm due to Lifshits [13], and (2) the recent randomized algorithms by Schmidt-Schauß and Schnitger [21]. Our tool is integrated with TreeRePair: it takes as input two terms represented in XML syntax and runs TreeRePair to build STGs. It then runs the unification algorithm. Through experiments we evaluate the performance of the resulting three unification algorithms and compare them to an implementation of a classical unification algorithm over uncompressed terms. Roughly speaking, unification over STGs is more efficient than over uncompressed terms, whenever the terms are well-compressible and larger than 100,000 nodes. At `www.lsi.upc.edu/~agascon/unif-stg` our system can be tested online. All our code is open source and will be available over the same web page.

2 Preliminaries

A *ranked alphabet* is a set $\mathcal{F}$ together with a function $\mathrm{ar} : \mathcal{F} \to \mathbb{N}$. Members of $\mathcal{F}$ are called function symbols, and $\mathrm{ar}(f)$ is called the *arity* of the function symbol f. Function symbols of arity 0 are called constants. Let $\mathcal{X}$ be a set disjoint from $\mathcal{F}$ whose elements have arity 0. The elements of $\mathcal{X}$ are called first-order variables. The set $\mathcal{T}(\mathcal{F} \cup \mathcal{X})$ of terms over $\mathcal{F}$ and $\mathcal{X}$, also denoted $\mathcal{T}(\mathcal{F}, \mathcal{X})$, is defined to be the smallest set having the property that $\alpha(t_1, \ldots, t_m) \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$ whenever $\alpha \in (\mathcal{F} \cup \mathcal{X})$, $m = \mathrm{ar}(\alpha)$ and $t_1, \ldots, t_m \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$. A *substitution* is a mapping $\sigma : X \to \mathcal{T}(\mathcal{F}, \mathcal{X})$ relating first-order variables to terms. Substitutions can also be applied to arbitrary terms by homomorphically extending them by $\sigma(f(t_1, \ldots, t_m)) = f(\sigma(t_1), \ldots, \sigma(t_m))$.

The *size* $|t|$ of a term t is the number of occurrences of variables and function symbols in t. The *height* of a term t, denoted $\mathrm{height}(t)$, is 0 if t is a constant or a first-order variable, and $1 + \max\{\mathrm{height}(t_1), \ldots, \mathrm{height}(t_m)\}$ if $t = \alpha(t_1, \ldots, t_m)$, with $m \geq 1$. The set $Pos(t)$ of *positions* of t is defined by $Pos(t) = \{\lambda\}$ if t is a constant or a variable, and $Pos(t) = \{\lambda\} \cup \{1 \cdot p \mid p \in Pos(t_1)\} \cup \ldots \cup \{m \cdot p \mid p \in Pos(t_m)\}$ if $t = \alpha(t_1, \ldots, t_m)$, where $m \geq 1$, λ denotes the empty sequence and $p \cdot q$, or simply pq, denotes the concatenation of p and q. If t is a term and p a position, then $t|_p$ is the subterm of t at position p. More formally defined, $t|_\lambda = t$ and $\alpha(t_1, \ldots, t_m)|_{i \cdot p} = t_i|_p$. We denote by $\mathrm{Pre}(t)$ the preorder

traversal (as a word) of a term t. It is recursively defined as $\mathrm{Pre}(t) = t$, if t has arity 0, and $\mathrm{Pre}(t) = \alpha \cdot \mathrm{Pre}(t_1) \cdot \ldots \cdot \mathrm{Pre}(t_m)$, if $t = \alpha(t_1, \ldots, t_m)$.

▶ **Definition 2.1.** A *Singleton (String) Grammar (SG)* G is a tuple $\langle \mathcal{N}, \Sigma, R \rangle$, where $\mathcal{N}$ is a finite set of non-terminals, Σ is a finite set of symbols (a signature), and R is a finite set of rules of the form $N \to \alpha$ where $N \in \mathcal{N}$ and $\alpha \in (\mathcal{N} \cup \Sigma)^*$. The sets $\mathcal{N}$ and Σ must be disjoint, and each non-terminal X appears as a left-hand side of just one rule of R. Let $N_1 >_G N_2$ for two non-terminals N_1, N_2, iff $(N_1 \to \alpha) \in R$, and N_2 occurs in α. The SG must be non-recursive, i.e. the transitive closure $>_G^+$ must be terminating. The word generated by a non-terminal N of G, denoted by $w_{G,N}$ or w_N when G is clear from the context, is the word in Σ^* reached from N by successive applications of the rules of G. SGs are also called *Straight Line Programs*.

With SG words of exponential length can be represented in linear space.

▶ **Example 2.2.** Let G be an SG with set of rules $\{A_0 \to a, A_1 \to A_0A_0, \ldots, A_n \to A_{n-1}A_{n-1}\}$. Then, $w_{A_n} = a^{2^n}$.

Let us fix a countable set $\mathcal{Y} = \{y_1, y_2, \ldots\}$ whose elements are function symbols of arity 0 called *parameters*. Given a ranked alphabet $\mathcal{F}$, we assume that $\mathcal{Y}$ and $\mathcal{F}$ are disjoint and define $\mathcal{T}(\mathcal{F} \cup \mathcal{Y})$ analogously to how $\mathcal{T}(\mathcal{F} \cup \mathcal{X})$ was defined in the preliminaries. We call the elements of $\mathcal{T}(\mathcal{F} \cup \mathcal{Y})$ *term patterns*.

▶ **Definition 2.3.** A *Singleton Tree Grammar (STG)* G is a 4-tuple $\langle \mathcal{N}, \Sigma, R, S \rangle$, where

- $\mathcal{N}$ is a ranked alphabet whose elements are called non-terminals.
- Σ is a ranked alphabet called signature.
- R is a finite set of rules of the form $N \to t$ where $N \in \mathcal{N}$, $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{N} \cup \{y_1, \ldots, y_{\mathrm{ar}(N)}\})$, $t \notin \mathcal{Y}$, and each of the parameters $\{y_1, \ldots, y_{\mathrm{ar}(N)}\}$ appears in t.
- S is the initial non-terminal of rank 0.

The sets $\mathcal{N}$ and Σ must be disjoint, each non-terminal N appears as a left-hand side of just one rule of R. Let $N_1 >_G N_2$ for two non-terminals N_1, N_2, iff $(N_1 \to \alpha) \in R$, and N_2 occurs in α. The STG must be non-recursive, i.e., the transitive closure $>_G^+$ must be terminating. The *depth* of G is the maximal length of a chain in $>_G^+$. We define the derivation relation $\Rightarrow_G$ on $\mathcal{T}(\mathcal{F} \cup \mathcal{N} \cup \mathcal{Y})$ as follows: $t \Rightarrow_G t'$ iff there exists $(A \to s) \in R$ with $\mathrm{ar}(A) = n, t = C(A(t_1, \ldots, t_n))$ and $t' = C(\sigma(s))$, where $\sigma = \{y_1 \to t_1, \ldots, y_n \to t_n\}$ and C is a context, i.e. a term in $\mathcal{T}(\mathcal{F} \cup \mathcal{N} \cup \mathcal{Y} \cup \{\bullet\})$ such that $\bullet$ appears only once in the positions of C, and $C(A)$ is simply the replacement of $\bullet$ by A in C. The term pattern generated by a non-terminal N of G, denoted by $t_{G,N}$ or t_N when G is clear from the context, is the term pattern in $\mathcal{T}(\mathcal{F} \cup \mathcal{Y})$ such that $N \Rightarrow_G^* t_N$.

Note that the rules of our grammars will always have ≤ 2 occurrences of non-parameter symbols in their right-hand sides. This is not a loss of generality since every STG can be efficiently normalized to satisfy this constraint (see [16]). Hence, we define the *size* of an STG/SG G, denoted $|G|$, as its number of nonterminals. An STG is *linear* if, for every rule $(N \to t)$, the term t is linear in $\mathcal{Y}$. An STG is called *k-bounded* if every non-terminal has arity $\leq k$. Finally, an STG is called *monadic* if it is 1-bounded. As shown by the next examples, STGs can represent terms of exponential height.

▶ **Example 2.4.** Let G be a monadic (and linear) STG with the following set of rules. $\{A_a \to a, A_0(y_1) \to f(y_1), A_1(y_1) \to A_0(A_0(y_1)), A_2(y_1) \to A_1(A_1(y_1)) \ldots, A_n(y_1) \to A_{n-1}(A_{n-1}(y_1)), S \to A_n(A_a)\}$. Then t_S generates a monadic tree $f(f(\ldots(a)\ldots))$ of size $2^n + 1$.

It is not difficult to prove that, for any linear STG $G = (\mathcal{N}, \Sigma, R, S)$, it holds that $|t_S| \leq 2^{O(|G|)}$. STGs can be considered as a generalization of directed acyclic graphs in which not only repeated subterms are shared but also repeated term patterns. In fact, DAGs can be seen as 0-bounded STGs.

In this work, STGs are used in the context of first-order unification. Hence, we want to represent terms containing first-order variables. From the point of view of the grammar, every first-order variable X initially is just a terminal symbol. As will be explained in the next section, they will be transformed in non-terminals as soon as they get instantiated due to the unification process by adding a rule of the form $X \to A$, where A is a non-terminal of rank 0. We call this kind of rules λ rules.

3 First-order unification and matching

Consider a ranked alphabet $\mathcal{F}$ and a set of first-order variables $\mathcal{X}$. The first-order unification problem consists of, given two terms $s, t \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$, finding a substitution σ such that $\sigma(s)$ and $\sigma(t)$ are syntactically equal. The first-order matching problem is a particular case of first-order unification in which only one of the terms given as input may contain variables. Both first-order unification and matching are common problems in areas like functional and logic programming, automated deduction, deductive databases, and compiler design.

Our tool Unif-STG offers three algorithms for solving first-order unification where the input terms s, t are represented compressed using STGs. Moreover, as a yardstick for comparison we implemented a variant of Corbin-Bidoit [7] that uses directed acyclic graphs for term representation. All four algorithms correspond, essentially, to the schema presented in Figure 1. Note that in this schema we consider indexes in the preorder traversal words of terms instead of just positions. This is not relevant for the algorithm that works on uncompressed terms but will make a difference in the compressed case. Note that two arbitrary different trees may have the same preorder traversal, but when they represent terms over a fixed signature where the arity of every function symbol is fixed, the preorder traversal is unique for every term. Thus, we can recursively define the mapping $\text{iPos}(t, i) \to Pos(t)$ relating positions in a term with indexes in its preorder traversal word as follows: $\text{iPos}(\alpha(t_1, \ldots, t_m), 1 + |t_1| + \ldots + |t_{i-1}| + k) = i.\text{iPos}(t_i, k)$ for $1 \leq k \leq |t_i|$ and $\text{iPos}(t_i, 1) = \lambda$. This observation was crucial in [10] to improve the algorithm of first-order unification with STGs since positions of a term represented with an STG G may have exponential size w.r.t. $|G|$ and need to be compressed which makes the computation of subterms inefficient. On the other hand, computing a subterm $t|_{\text{iPos}(t,i)}$ given the index i can be done in a much more efficient way. It is important to remark that XML trees are unranked in general and hence two different trees may have the same preorder traversal. Hence, it is important to transform XML trees to ranked trees (terms) to apply the approach mentioned above. Note that the basic operations in that schema are to decide equality between two terms, apply a substitution, compute the preorder traversal word of a term, compute a subterm of a term given an index of its preorder traversal word, find the first different positions of two words, and check whether a certain symbol occurs within a term. In the setting of Unif-STG, the input terms s, t are represented using STGs. Since STGs can represent terms of exponential size, the difficulty of applying that schema to compressed terms relies on being able to solve all these subproblems in polynomial time with respect to the size of the input STG. In [9], this problem was solved in time $\mathcal{O}(|V|(m + |V|n)^4)$, where m represents the size of the input STG, n represents its depth, and V represents the set of different first-order variables occurring in the input terms. Then this result was improved in [10] to $\mathcal{O}(|V|(m + |V|n)^3)$.

```
Unify(s : term, t : term):
  σ: substitution
  σ := ∅
  While (σ(s) ≠ σ(t)):
    Look for the first position k such that Pre(σ(s))[k] ≠ Pre(σ(t))[k].
    If both Pre(σ(s))[k] and Pre(σ(t))[k] are function symbols, Then
      Return false (clash)
    // Assume w.l.g that Pre(σ(s))[k], is a variable x.
    If x occurs in σ(t)|_p, where p = iPos(σ(t), k), Then
      Return false (occur-check)
    σ := σ ∪ {x ↦ σ(t)|_p}
  EndWhile
  Return true
```

Figure 1 General schema for first-order unification

From [10], we know that the following problems can be solved in linear time:

- Given a SG/STG G, compute the number $|t_N|/|w_N|$ for every non-terminal N of G.
- Given a STG G and a non-terminal N, construct an SG of linear size for $\mathrm{Pre}(t_{G,N})$.
- Given an STG G, a non-terminal N of G, and an integer k, compute an extension G' of G such that G' generates $t_{G,N}|_{\mathrm{iPos}(t_{G,N},k)}$.

Also from [10], we know that, given two words compressed with SGs, we can find the first position in which they differ in linear time using a data structure computed by Lifshit's algorithms for checking equality. This problem can also be solved probabilistically in linear time as sketched in the following subsection. Moreover, an application of a substitution $\{X \to t\}$ is simulated by transforming X from a terminal to a non-terminal of the grammar and adding the rule $X \to N$, where the non-terminal N generates t. In this way, the grammar may be extended with at most n new non-terminals after each variable assignment as proved in [10]. Hence, the final size of the grammar is bounded by $m + |V|n$. Thus, the resulting running time corresponds to decide equality for STGs $|V|$ times on a grammar of size $\mathcal{O}(m + |V|n)$. In [10], the problem of deciding equality between two terms represented by STGs is reduced to equality between words represented by SGs. Lifshits' algorithm [13] is, with respect to big-O complexity, the most efficient known exact algorithm for checking equality between two words represented by SGs. It is cubic with respect to the sizes of the input SGs. Our three implementations of first-order unification with STGs correspond to different algorithms for solving this subproblem. Unif-STG allows the user to choose among Lifshits' algorithm and two of the recent randomized algorithms by Schmidt-Schauß and Schnitger [21]. Since our implementation of the randomized algorithms runs in linear time, the cost of first-order unification is $\mathcal{O}(|V|(m + |V|n))$ when they are used. See below where we describe these equality testing algorithms.

With respect to first-order matching with STGs, an algorithm with cost $\mathcal{O}((m + |V|n)^3)$ was presented also in [10]. The improvement with respect to the unification case relies in the fact that, in contrast to unification schema presented in Figure 1, in the matching case we just need to look for the index of first occurrence of a variable in $\mathrm{Pre}(s)$ instead of looking for the index of the first difference between $\mathrm{Pre}(s)$ and $\mathrm{Pre}(t)$ and do the corresponding assignment until every variable is replaced. At the end we just perform equivalence testing *once*. For the details of that algorithm we refer the reader to [10]. Using the randomized algorithms of [21] first-order matching can be solved in $\mathcal{O}(m + |V|n)$.

Note that in [9] and [10] only monadic grammars were considered. This is not a loss of generality since every linear STG can be transformed in polynomial time into a monadic (and linear) one [16]. However, their algorithm is rather involved and difficult to implement. We

therefore extended the unification algorithm of [10] to unbounded grammars. This mainly consists of generalizing the construction for the computation of a subterm to unbounded grammars. The solutions implemented in Unif-STG for the rest of the subproblems are straightforward adaptations of those in [10] and are not further discussed here.

Equality testing. Given an SG $G = (\mathcal{N}, \Sigma, R)$ and two non-terminals A, B, equality testing consists of deciding whether $w_A = w_B$. Let us assume that $|w_A| = |w_B|$ since otherwise inequality is easily stated in linear time. As commented above, the fastest known exact algorithm for equality testing for SGs is Lifshits' algorithm [13]. In Unif-STG we implemented, in addition to Lifshits' algorithm, two new algorithms of [21]. These algorithms run faster than Lifshits' by using a randomized approach. They work by considering an SG to generate a natural number, in addition to a word. The number coded by $w_A = w'a$, where $w' \in \Sigma^*$ and $a \in \Sigma$, is defined in terms of a fixed mapping $f : \Sigma \to \{0, \ldots, |\Sigma| - 1\}$, as $code(w_A) = code(w') * |\Sigma| + f(a)$. The main idea of the algorithm is very simple. If we want to check whether A and B represent the same word, we choose a natural number m satisfying certain properties, and compute $\alpha = code(w_A) \mod m$ and $\beta = code(w_B) \mod m$. If $\alpha \neq \beta$ then the words are obviously different. Otherwise, it is possible that $w_A \neq w_B$, but $\alpha = \beta$. In this case we do not detect inequality. In [21], two upper bounds for the choice of the m that guarantee that we detect inequality with a probability $\geq \frac{1}{2}$ for any pair of words are given: either $m \leq |w_A|^2 * c$ or $m \leq |w_A| * c$ if m is prime, for a certain constant c. We implemented both options in Unif-STG. By repeating the test k times the probability of not detecting inequality is $< \frac{1}{2^k}$. In Unif-STG the value of k is set to 10 by default.

In order to assure that the chosen m is prime we implement a simple algorithm: generate a random number, and test primality. If the number is not prime, then generate another number, and so on. We test primality with the Fermat primality test, checking if $a^{p-1} \equiv 1 \mod p$ for $a \in \{2, 3, 5, 7\}$. Due to the Prime number theorem, the average number of times we generate a number until getting a prime is the logarithm of m, and hence linear in $|G|$, and the Fermat primality test is also performed in logarithmic time.

We also need an algorithm to compute if w_A is a prefix of w_B, in order to find the first difference between two words represented with SGs (see Figure 1). This problem can be reduced to computing $code(w_B[1 \ldots |w_A|])$ and applying the probabilistic algorithm. To perform this task in linear time it is enough to precompute, for each non-terminal A of the grammar, the numbers $code(w_A)$ and $|w_A|$, and to compute $code(w_B[1 \ldots |w_A|])$ recursively.

Finally, it is important to remark a certain peculiarity of the version of the probabilistic algorithms implemented in Unif-STG. They run in linear time thanks to the fact that $|w_A|$ is limited by default to $\sqrt{L}$ where $L = 2^{64}$, the maximum value for a long long int, in the case of the algorithm using primes; and to $\sqrt[4]{L}$, in the algorithm using natural numbers. Otherwise, computing $code(w_A)$ modulo m is not guaranteed to run in linear time. The current implementation allows bigger values, but then does not guarantee an error probability of less than $\frac{1}{2}$ for every possible instance of the problem. In our experiments we never encountered a false reply by the probabilistic algorithm.

Note that Unif-STG has been built to work with arbitrary arithmetic; the size limitation has been added for efficiency reasons only and can be removed at any time.

4 Unif-STG

Unif-STG is written in C++ using the standard template library. The system implements three algorithms for solving the equality testing with SGs: Lifshits' exact algorithm, plus two versions of the randomized algorithm by Schmidt-Schauß and Schnitger (one with integers and

one with primes). We refer to the corresponding three versions of the unification algorithm for STG grammars by STG-exact, STG-rand, and STG-rand-prime. Our implementation of unification over uncompressed terms is denoted "tUnif". As commented before, this is a variant of the Corbin-Bidoit algorithm. We refer the reader to Chapter 8, Section 2.3 of [1] for the details of this algorithm. Note that Unif-STG outputs a compressed representation of the solution (again compressed with STGs).

5 Experiments

Experimental Setup. All tests are executed on a machine with Intel Xeon Core 2 Duo, 3 Ghz processor, with 4GB of RAM. We use the Ubuntu Linux 9.10 distribution, with kernel 2.6.32 and 64 bits userland. Our implementation was compiled using g++ 4.4.1. We used TreeRePair (build data 01-19-2011) which was kindly made available to us by Roy Mennicke. It is essentially the version available at `http://code.google.com/p/treerepair`, with the only difference that it allows to compress without prior applying a binary tree encoding. We run TreeRePair with the switches "-multiary -bplex -c -nodag -optimize edges" and default value (4) for maxbound. The latter means that only 4-bounded STGs are generated.

Protocol. Each test is executed three times, and the fastest time of the three runs is reported. We only measure the pure unification time, i.e, we ignore loading time and setup of basic data structures, etc.

Design of the Experiments. Instead of trying to find instances of unification problems with large terms that are realistic and likely to appear in practice, we present results over artificial examples. The idea behind these examples is to test the behaviour of our algorithms in the different extreme corner cases. The main aspects that make up these cases are (a) are the terms well-compressible by TreeRePair? (b) do they unify or not? (c) how many variables? (d) is it only matching; how much copying of variables? For question (a) we need to distinguish further: (a1) is the "top-matching part", i.e., parts where both terms do not have variables (and therefore must match exactly) well-compressible? And (a2) is the "binding part", that is, parts that will be bound to variables during unification well compressible? We constructed a family of instances that allows to test many of these aspects. First we show two simple examples which compress well.

Bin and Mon. For a natural number n, let $f^n(a,b)$ denote a full binary tree with leaf sequence $ababab\ldots$. Given a natural number n, $\mathrm{Bin}(n)$ consists of the pair of trees

$$\mathrm{Bin}(n) = (g(g(f^n(a,b), f^n(a,b)), g(f^n(a,b), f^n(a,b))), \quad g(g(X,X), g(X, f^n(a,Y)))).$$

Similarly, $f^n(a)$ denotes a monadic tree of height n with internal nodes labeled f and leaf a. The second example, called $\mathrm{Mon}(n)$ consists of this pair of trees

$$\mathrm{Mon}(n) = (h(f^n(X), f^n(Y), Y), \quad h(T,Z,T)).$$

Clearly, both Bin and Mon are unifiable for every n. Moreover, they are well compressible with TreeRePair. To see this, consider the right part of Table 1 which shows the compression time, the number of edges in the original tree and in the STG grammar, plus the file sizes of the original tree (in XML format) and of the grammar (in text format). It also shows the file size of the grammar in CNF in the special format that our unification program uses.

For both Bin and Mon, the TreeRePair algorithms achieves exponential compression rates. As can be seen, for $n > 20000$, the STG-rand algorithm is the fastest. Interestingly, for such small grammars we are punished for using prime numbers and STG-rand-prime is slower than STG-rand. This is different for larger grammars as the later examples show.

	Runtime (in ms)				Input			
n/1000	tUnif	STG-randp	STG-rand	STG-exact	edges	compr. time	STG edges	CNF file
5	**2**	8	8	24	10008 (69K)	55ms	38 (388B)	1K
10	**5**	10	8	28	20008 (137K)	62ms	40 (398B)	1.1K
20	11	11	**9**	30	40008 (157K)	140ms	42 (420B)	1.1K
50	44	11	**9**	30	100T (684K)	341ms	45 (434B)	1.2K
100	107	12	**10**	31	200T (1.4M)	681ms	47 (457B)	1.3K
200	232	13	**10**	32	400T (2.7M)	1387ms	49 (467B)	1.3K

Table 1 The example Mon(n)

	Runtime (in ms)				Input		
randSize	tUnif	STG-rp	STG-r	STG-e	edges	STG edges	CNF file
10	**3**	18	19	78	20484 (111K)	62 (578B)	1.9K
11	**7**	20	20	96	40964 (221K)	66 (596B)	2.1K
12	**16**	22	22	108	81924 (441K)	70 (638B)	2.2K
13	35	**23**	**23**	131	163844 (881K)	74 (656B)	2.3K
14	72	26	**25**	146	327684 (1.8M)	78 (698B)	2.4K
16	290	30	**28**	(*)	1310724 (6.9M)	86 (758B)	2.7K

(*) STG-exact ran out of (int) bounds.

Table 2 The example Bin(n)

Note that here the exact algorithm still shows reasonable performance. This will not be the case for larger grammars. Note that, in terms of XML, Mon is actually quite relevant: a long list of items usually becomes a long list of siblings in XML. Using the common "first-child/next-sibling"-encoding of unranked into binary trees, such a list becomes a long path, similar to Mon.

Bad Instances for STG-Unif. Here consider instances where the STG-based unification algorithm does *not* perform well. In general, this is the case when the terms are *not well compressible* (see below). But, there are even simpler reasons for this to happen. Consider unifying the trees $f(t)$ and $g(t')$ for large (arbitrary) terms t and t'. The run time of tree-based unification is only 0.005ms for this instance. While, even for highly compressible $t = t'$, STG-Unif will take >15ms. This is due to the fact that STG-Unif always needs to traverse the whole grammar to find the position of the first difference between $f(t)$ and $g(t')$ and tUnif traverses the input tree *only* until the position of the first difference is reached.

Meta. We now define a highly configurable example instance. Consider the pair of trees (t_1, t_2), where both t_1 and t_2 are full binary trees (with internal nodes labeled f) of height n. At the leaves of t_1 and t_2 appear monadic trees of random height h, with $\mathit{minHeight} \leq h \leq \mathit{maxHeight}$. These monadic trees are identical in t_1 and t_2. Now, t_1 contains variables as leaves of the monadic trees, randomly chosen from a given set *Vars* of variables. While t_2 contains random trees at those leaf positions, chosen over a given signature Σ, and maximal size of up to *randSize*. Moreover, a Boolean determines whether at variable copies we force the random trees in t_2 to be equal (which will guarantee that the instance is unifiable). Thus, the specification of an instance is as follows: Meta(n, *minHeight*, *maxHeight*, *Vars*, *randSize*, Σ, Bool U).

Number of Variables. Using Meta, we experimented with the number of different unification variables. The results were convoluted and no clear trends were observable; both algorithms seemed similarly impacted by the number of variables. For instance, for $n = 4$, $\mathit{maxHeight} = \mathit{minHeight} = 1000$, $\mathit{randSize} = 1$ and $|\Sigma| = 3$ we obtain, for 3 variables: 3ms/18ms (tUnif/STG-rand-prime), for 5 variables: 7ms/32ms, and for 10 variables: 10ms/44ms.

Incompressible Terms. An interesting case is if large incompressible terms appear

at positions that will instantiate variables. In terms of Meta, it suffices to take $n = 1$, and to use large random trees. For the other parameters we use $minHeight = maxHeight = 0$, $Vars = \{X, Y\}$, $\Sigma = \{g^{(2)}, f^{(1)}, a^{(0)}\}$, and Boolean U set to true. As Table 3 shows, STG-Unif is indeed about 100-times slower than tree-based unification. The difference in speed seems to get slightly smaller for very large inputs. As comparison, if we add a larger binary tree on top of t_1, i.e., use a larger n, then the tree becomes more compressible and therefore STG-based unification becomes efficient. This is shown in the right of Figure 2, where we pick $randSize = 20000$, but now use monadic trees of size 0–1000.

	Runtime (in ms)				Input		
randSize	tUnif	STG-rp	STG-r	STG-e	edges	STG edges	CNF file
1000	**0.1**	21	34	222	981 (5.9K)	214 (1.5K)	6.6K
5000	**0.6**	78	116	2430	4778 (29K)	774 (15K)	25K
20000	**2.4**	405	598	43632	26114 (157K)	3308 (21K)	107K
50000	**12**	1396	2074	(*)	94280 (564K)	9975 (63K)	327K
200000	**43**	5464	8036	(*)	334586 (2M)	30740 (196K)	1.1M

Table 3 Incompressible Terms in Substitution Positions

With respect to *unifiability*, we observed that changing a few nodes to make the input non-unifiable, causes STG-rand to take ca. twice the time given in Table 3, while tUnif gets slightly faster.

There are also examples where the solution consists of deeper trees than the input. This works well for the uncompressed algorithm too. But, we can see the effect of compression: consider $t_1 = h(X, Y, Z)$ and $t_2 = h(s_1, s_2, s_3)$, where s_1 is a full binary tree (over f's) of height n with all leaves labeled Y, s_2 is a full binary tree (over f's) of height n with all leaves labeled Z, and s_3 the same but with leaves labeled X. Note that X will be assigned to s_1. Hence, all the X's in s_3 will be replaced by s_1. Then, Y will be replaced by s_2 everywhere (also in s_3). So finally $t' = Y \to s_2(X \to s_1(s_3))$ will be compared to Z. Note that t' is the complete tree of depth $3 * 20$ whose leaves are all labeled Z. Since Z occurs in t' unification fails. This example is called "3-Stack" and timings are shown in Figure 2.

n	tUnif	STG-rp	STG-r	STG-e
18	99	**7**	9	35
19	200	**8**	9	38
20	401	**8**	10	(*)

n	tUnif	STG-rp	STG-r	STG-e
7	**369**	1694	2130	(*)
8	**688**	1726	2149	(*)
9	**1730**	2391	2982	(*)

Figure 2 The example 3-Stack (left) and randSize=20000 of Table 3 (right)

6 Conclusion and Further Work

Besides the rather immediate application of our work to logic programming with XML (mentioned in the Introduction), it might also be possible to apply compressed terms within theorem provers. The latter do not usually store very large trees, but store many trees. Of course, many small trees could be combined into one large tree, prior to grammar compression. However, this might induce extra costs for referencing those trees. In future work it should be studied how unification and matching (and other operations needed in theorem provers) can be applied to a set of trees represented by a cf tree grammar. Moreover, efficient updates need to be supported over such grammars. Updates on STGs have been considered in [8], but have not been implemented in any large system yet.

Acknowledgments The authors would like to thank Miguel Florido for his help during this work.

References

1 F. Baader and J.H. Siekmann. Unification theory. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 2*, pages 41–125. Oxford Univ. Press, 1994.

2 F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *ICLP*, pages 255–270, 2002.

3 P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *VLDB*, pages 141–152, 2003.

4 G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML document trees. *Inf. Syst.*, 33:456–474, 2008.

5 M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

6 J. Coelho and M. Florido. XCentric: logic programming for XML processing. In *WIDM*, pages 1–8, 2007.

7 J. Corbin and M. Bidoit. A rehabilitation of Robinson's unification algorithm. In *IFIP Congress*, pages 909–914, 1983.

8 D. K. Fisher and S. Maneth. Structural selectivity estimation for XML documents. In *ICDE*, pages 626–635, 2007.

9 A. Gascón, G. Godoy, and M. Schmidt-Schauß. Unification with singleton tree grammars. In *RTA*, pages 365–379, 2009.

10 A. Gascón, G. Godoy, and M. Schmidt-Schauß. Unification and matching on compressed terms. *CoRR*, abs/1003.1632, 2010.

11 T. Kutsia. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In *AISC*, pages 290–304, 2002.

12 J. Levy, M. Schmidt-Schauß, and M. Villaret. The complexity of monadic second-order unification. *SIAM J. Comput.*, 38:1113–1140, 2008.

13 Y. Lifshits. Processing compressed texts: A tractability border. In *CPM*, pages 228–240, 2007.

14 M. Lohrey and S. Maneth. The complexity of tree automata and XPath on grammar-compressed trees. *Theor. Comput. Sci.*, 363(2):196–210, 2006.

15 M. Lohrey, S. Maneth, and R. Mennicke. Tree structure compression with RePair. *CoRR*, abs/1007.5406, 2010. Short version to appear as paper in *Proc. DCC'2011*.

16 M. Lohrey, S. Maneth, and M. Schmidt-Schauß. Parameter reduction in grammar-compressed trees. In *FOSSACS*, pages 212–226, 2009.

17 S. Maneth and T. Sebastian. Fast and tiny structural self-indexes for XML. *CoRR*, abs/1012.5696, 2010.

18 J.A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.

19 W. Rytter. Grammar compression, LZ-encodings, and string algorithms with implicit input. In *Proc. ICALP*, volume 3142 of *LNCS*, pages 15–27, 2004.

20 K. Sadakane and G. Navarro. Fully-functional succinct trees. In *SODA*, pages 134–149, 2010.

21 M. Schmidt-Schauß and G. Schnitger. Fast equality test for straight-line compressed strings. unpublished manuscript, October 2010.