Tree Transducers and Tree Compressions

Sebastian Maneth[1] and Giorgio Busatto[2]

[1] EPF Lausanne, School of Computer and Communication Sciences
[2] Universität Oldenburg, Department für Informatik

Abstract. A tree can be compressed into a DAG by sharing common subtrees. The resulting DAG is at most exponentially smaller than the original tree. Consider an attribute grammar that generates trees as output. It is well known that, given an input tree s, a DAG representation of the corresponding output tree can be computed in time linear in the size of s. A more powerful way of tree compression is to allow the sharing of tree patterns, i.e., internal parts of the tree. The resulting "sharing graph" is at most double-exponentially smaller than the original tree. Consider a macro tree transducer and an input tree s. The main result is that a sharing graph representation of the corresponding output tree can be computed in time linear in the size of s. A similar result holds for macro forest transducers which translate unranked forests, i.e., natural representations of XML documents.

1 Introduction

Consider a finite, labeled, ranked, and ordered tree. A tree of this type can for example be represented by a bracketed expression of the form $c(g(a, b, b), c(a, a))$. How can such a tree be compressed? Or, what is its smallest representation? Certainly, the smallest Turing Machine (or C program) that generates the tree is a good answer to the latter question. However, not only is such a representation very difficult to obtain, but it is also hard to alter or merely query it (without decompressing it first).

Instead of this general approach to compression we are interested in a representation of trees in which the functionality of the basic tree operations (such as the movement on nodes along the edges) are preserved. This type of compression is called "data optimization" in [13].

In the context of XML there has been some recent work on tree compression. XML documents represent trees that are slightly different from the ones discussed above. First, they are unranked, i.e., a node in an XML tree can have arbitrarily many children, and second, labels are typed (in the sense that there are tags for internal nodes, data values for leaves, and the latter can be of various primitive types such as integers, strings, etc.). The XMill compression tool [16] takes proper care of the type issue by grouping all values of the same type into one container. The containers are then compressed using known methods (such as `gzip` for string values). However, XMill is *not* a data optimization tool: the resulting output cannot be queried or processed without prior decompression.

I. Walukiewicz (Ed.): FOSSACS 2004, LNCS 2987, pp. 363–377, 2004.
© Springer-Verlag Berlin Heidelberg 2004

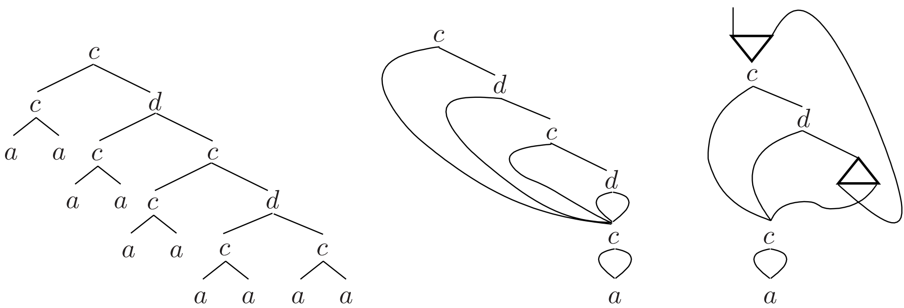

Fig. 1. The tree t, its minimal DAG g, and a sharing graph h.

XML data optimization is considered in [2,10]. There it is shown that (by sharing of common subtrees) a minimal DAG representation of a tree can be obtained in time linear in the size of the tree. A tree t together with its DAG representation g can be seen in Fig. 1. Moreover, they consider the problem of evaluating tree queries on DAG representations. Note that in their approach only internal nodes of an XML documents are collapsed in the DAG.

In this paper we consider a tree optimization method that is based on *sharing graphs* (for short, sgraph). The latter were used by Lamping [14] to implement optimal reductions of λ-calculus (see also [1,12]). Sgraphs can be seen as a generalization of DAGs in the following sense: consider a node of a DAG that is shared, i.e., with $k > 1$ incoming edges. This node can be seen as a special "begin sharing" marker because the tree rooted at that node is being shared by several other nodes. The sgraph now generalizes this idea by adding a symmetric "end sharing" marker. Consequently, such a marker has one incoming and k outgoing edges. Begin and end markers are also called fan-ins and fan-outs (or multiplexers and demultiplexers) and are depicted by a triangle pointing down (with k incoming edges from above, ordered) and its vertical mirror image, respectively. An sgraph representation of the tree t is shown in the right of Fig. 1. An sgraph is unfolded by following its paths starting at the root node; if, on such a path, the ith input of a fan-in is entered then the next fan-out must be exited by the ith output.

Tree compression using DAGs has a maximal compression ratio of $1/\log n$ (achieved, e.g., when representing a full binary tree of height n by a DAG with n nodes). Tree compression using sgraphs has a maximal compression ration of $1/\log\log n$. It can be achieved by representing a full binary tree of height 2^n by n pairs of nested fan-in/fan-outs which share a binary node Such an sgraph is shown in the right of Fig. 4 (for $n = 3$).

The minimal DAG that represents a tree t can be seen as the minimal finite state tree automaton that accepts t, or as the minimal regular tree grammar that generates t. In this sense the generalization of DAGs to sgraphs just corresponds to moving from finite state automata/regular grammars to push-down down automata/context-free grammars. In fact, it is not difficult to interpret an sgraph as a particular push-down tree automaton, or as a particular context-free tree

$S \to c(A, B)$	$S \to B(B(A))$
$A \to c(a, a)$	$A \to c(a, a)$
$B \to d(A, C)$	$B(y) \to c(A, d(A, y))$
$C \to c(A, D)$	
$D \to c(A, A)$	

Fig. 2. Regular and context-free tree grammars that generate $\{t\}$.

grammar (the latter is discussed in Sect. 4). In Fig. 2 we see grammars for g and h of Fig. 1.

For strings the idea of using context-free grammars to compress is well known. In fact, as shown in [15], the famous LZ78 compression is just a particular instance of compression by context-free grammars. It is shown in [15] that even though the problem of finding for a string the minimal context-free grammar is NP-complete, there exist quite a number of good approximation algorithms for this problem. Context-free grammars which generate exactly one string are also known as “straight-line programs”, and in [21] it was proved that deciding their equivalence can be done in polynomial time. Thus, the corresponding strings need *not* be decompressed in order to test their equivalence.

For trees the idea of grammar-based compression seems to be new. Clearly, the problem of finding a minimal context-free tree grammar is NP-complete (because it can be reduced to the corresponding string problem). Can the approximation algorithms discussed in [15] be generalized to trees? Is the equivalence problem for straight-line context-free tree grammars solvable in polynomial time? Both questions are subject of further research and are not addressed in this paper. Rather, we consider tree translation formalisms that can be altered in order to generate sgraphs.

In particular we consider the macro tree transducer [8] which is a powerful model of syntax directed translation. It can be obtained by combining the top-down tree transducer with the macro grammar. Recently it has been shown that (compositions of) macro tree transducers can simulate pebble tree transducers [7]. The latter were introduced in [19] and model the “tree translation core” of all known XML query and transformation languages (including, e.g., XQuery and XSLT).

Macro tree transducers (mtts) can be simulated by the top-down tree to graph transducers of [9]. In fact, the output graphs are DAG representations of the corresponding output trees. Now, what happens if we use top-down tree to graph transducers in order to generate sgraphs? Our main result is that in this way even *linear* top-down tree to graph transducers can simulate mtts. Linearity means that every node of the input tree is processed at most once. As consequence we obtain that an sgraph representation of the output tree of an mtt can be computed in time linear in the size of the corresponding input tree.

Previous complexity results about mtts [17] are based on simulations by attribute grammars. For the latter it is folklore that a DAG representation of the output tree can be computed in time linear in the size of the corresponding

input tree. In fact, the involved attribute grammars generate trees by using tree concatenation (= "first-order tree substitution") as only operation. The same operation is used in the derivation of a regular tree grammar. This explains why DAGs are particularly well suited for representing outputs of such attribute grammars (or, attributed tree transducers as they are called). Since macro tree transducers can be seen as a generalization of context-free tree grammars, it is not surprising that for them sgraphs are well suited to represent their outputs. Hence, our result is the generalization of the linear time computibility from attributed tree transducers on DAGs to macro tree transducers on sgraphs.

Our simulation of mtts by linear top-down tree to sgraph transducers implies that every output language of an mtt can be represented as a context-free sgraph language. This is in accordance with the fact that output tree languages of attributed tree transducers can be represented as context-free DAG languages [4].

It should be noted that there is a price to be paid for the fact that sgraphs can compress better than DAGs: their unfolding is more difficult. The same holds for querying/processing an sgraph because, intuitively, a stack containing the history of entered fan-ins has to be maintained at all times. In the last section of this paper we address the problem of querying/processing an sgraph and give some bounds on the amount of overhead needed.

2 Trees, DAGs, and Sharing Graphs

We assume the reader to be familiar with trees, tree automata, and tree translations (see, e.g., [11]). A set Σ together with a mapping rank: $\Sigma \to \mathbb{N}$ is called a *ranked* set. For $k \geq 0$, $\Sigma^{(k)}$ is the set $\{\sigma \in \Sigma \mid \text{rank}(\sigma) = k\}$; we also write $\sigma^{(k)}$ to denote that $\text{rank}(\sigma) = k$. The set $\text{inc}(\Sigma)$ consists of all symbols $\sigma \in \Sigma$, but now with rank $1 + \text{rank}_\Sigma(\sigma)$. For a set A, $\langle \Sigma, A \rangle$ is the ranked set $\{\langle \sigma, a \rangle \mid \sigma \in \Sigma, a \in A\}$ with $\text{rank}(\langle \sigma, a \rangle) = \text{rank}(\sigma)$. The set of all (ordered, ranked) trees over Σ is denoted T_Σ. For a tree t, $V(t)$ is the set of nodes of t. The size of t is its number $|V(t)|$ of nodes. For a set A, $T_\Sigma(A)$ is the set of all trees over $\Sigma \cup A$, where all elements in A have rank zero. We fix the *set of input variables* as $X = \{x_1, x_2, \dots\}$ and the *set of parameters* as $Y = \{y_1, y_2, \dots\}$. For $k \geq 0$, $X_k = \{x_1, \dots, x_k\}$ and $Y_k = \{y_1, \dots, y_k\}$.

For the representation of DAGs and sharing graphs we use hypergraphs. The reader is assumed to be familiar with hypergraphs and hyperedge replacement, see, e.g., [3]. For a ranked alphabet Γ and $m \geq 0$, a *hypergraph g of rank m over Γ* consists of finite sets of nodes and hyperedges. Every hyperedge e of rank k is incident with a sequence $\text{nod}(e)$ of k nodes ("the nodes of e") and is labeled by a symbol of rank k, i.e., in $\Gamma^{(k)}$. Furthermore, there is a sequence ext of "external nodes" which has length m.

To represent trees, DAGs, and sharing graphs by hypergraphs we use the following order on an edge e of rank $m \geq 1$: if $\text{nod}(e) = v_1 \cdots v_m$ then $v_1 \cdots v_{m-1}$ is the sequence of *argument nodes* of e, denoted $\text{ar}(d)$, and v_m is the *result node* of e, denoted $\text{res}(d)$. Let g be a hypergraph. A *path* of g is a sequence $u_1 \cdots u_n$ of nodes such that there are hyperedges $e_1, \dots, e_n$ with $u_1 = \text{res}(e_1)$ and, for

every $2 \leq i \leq n$, $u_i = \text{res}(e_i)$ and u_i appears in $\text{ar}(e_{i-1})$. If all paths ρ of g are acyclic, i.e., no node appears more than once in ρ, then g is a *directed acyclic (hyper)graph* (DAG).

Let Δ be a ranked alphabet and $m \geq 2$. Define $\Gamma_m = \{2^{(3)}, \underline{2}^{(3)}, \ldots, m^{(m+1)}, \underline{m}^{(m+1)}\}$. A hypergraph g over $\Delta \cup \Gamma_m$ is a *sharing graph* (*sgraph*) if applying the following rewrite rules results in a tree t (recall from the Introduction our conventions on how to draw sgraphs). The left rule generates i copies of a symbol f which is shared by a fan-in. In this way the fan-in is moved down (and split into n copies, where n is the number of arguments of f). If a fan-in meets a corresponding fan-out (right rule), then both are deleted and their inputs and outputs are melted together appropriately. The tree t is the *unfolding* of g, denoted tree(g). Notice that the rewriting system is confluent but not terminating.

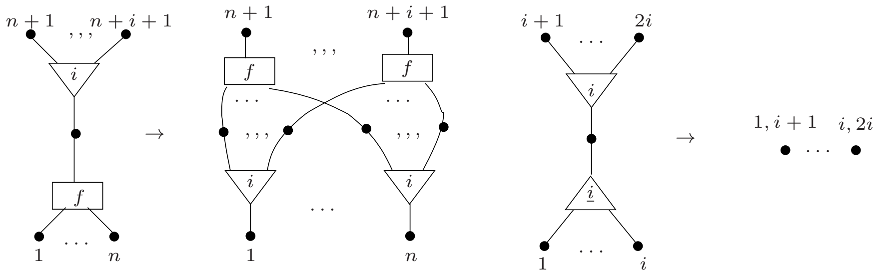

Fig. 3. The rules of the rewriting system that unfolds a sharing graph.

3 Sharing Graph Implementation of Tree Transducers

In this section our two main results are proved. First, for every macro tree transducer and given an input tree of size n, an sgraph representation of the corresponding output tree can be computed in time linear in n. The second result is about macro forest transducers. There we can only show that an sgraph representation of an output forest can be computed in time exponential in n. However, if the macro forest transducer does not copy by means of its input variables x_i, then the output sgraph can be computed in time linear in n. Sgraphs generated by the above two implementations can contain garbage. Last we discuss how to avoid the generation of garbage.

Macro Tree Transducers are finite state devices that take trees over a ranked alphabet as input and produce trees over another ranked alphabet. Here we only deal with total deterministic macro tree transducers which realize total functions on trees.

Definition 1. A (total, deterministic) *macro tree transducer* (mtt) is a tuple $M = (Q, \Sigma, \Delta, q_0, R)$, where Q is a ranked alphabet of *states*, Σ and Δ are ranked alphabets of *input* and *output symbols*, respectively, $q_0 \in Q^{(0)}$ is the

initial state, and R is a finite set of *rules*. For every $q \in Q^{(m)}$ and $\sigma \in \Sigma^{(k)}$ with $m, k \geq 0$ there is exactly one rule of the form $\langle q, \sigma(x_1, \ldots, x_k)\rangle(y_1, \ldots, y_m) \rightarrow \zeta$ in R, where $\zeta \in T_{\langle Q, X_k\rangle \cup \Delta}(Y_m)$; the tree ζ is denoted by $\mathrm{rhs}_M(q, \sigma)$. □

The rules of M are used as term rewriting rules in the usual way. The derivation relation of M (on $T_{\langle Q, T_\Sigma\rangle \cup \Delta}$) is denoted by $\Rightarrow_M$ and the *translation realized by* M, denoted τ_M, is the total function $\{(s, t) \in T_\Sigma \times T_\Delta \mid \langle q_0, s\rangle \Rightarrow_M^* t\}$.

Example 2. We define the mtt M_{dexp} which translates a monadic tree of height n into a full binary tree of height 2^n. Let $M_{\mathrm{dexp}} = (Q, \Sigma, \Delta, q_0, R)$ with $Q = \{q_0^{(0)}, q^{(1)}\}$, $\Sigma = \{a^{(1)}, e^{(0)}\}$, and $\Delta = \{\sigma^{(2)}, e^{(0)}\}$. The set R consists of the following rules.

$$
\begin{array}{ll}
\langle q_0, a(x_1)\rangle & \rightarrow \langle q, x_1\rangle(\langle q, x_1\rangle(e)) \\
\langle q_0, e\rangle & \rightarrow \sigma(e, e) \\
\langle q, a(x_1)\rangle(y_1) & \rightarrow \langle q, x_1\rangle(\langle q, x_1\rangle(y_1)) \\
\langle q, e\rangle(y_1) & \rightarrow \sigma(y_1, y_1)
\end{array}
$$

Let us take a look at a computation of M_{dexp} for the input tree $s = aaae$ (for better readability we sometimes omit brackets in monadic trees):

$$
\begin{array}{ll}
\langle q_0, aaae\rangle & \Rightarrow_{M_{\mathrm{dexp}}} \langle q, aae\rangle(\langle q, aae\rangle(e)) \\
& \Rightarrow^2_{M_{\mathrm{dexp}}} \langle q, ae\rangle(\langle q, ae\rangle(\langle q, ae\rangle(\langle q, ae\rangle(e)))) \\
& \Rightarrow^4_{M_{\mathrm{dexp}}} \langle q, e\rangle^8(e) \\
& \Rightarrow_{M_{\mathrm{dexp}}} \sigma(\langle q, e\rangle^7(e), \langle q, e\rangle^7(e)) \\
& \Rightarrow_{M_{\mathrm{dexp}}} \sigma(\sigma(\langle q, e\rangle^6(e), \langle q, e\rangle^6(e)), \sigma(\langle q, e\rangle^6(e), \langle q, e\rangle^6(e))) \\
& \Rightarrow^{24}_{M_{\mathrm{dexp}}} \mathrm{fbt}_\Delta(8)
\end{array}
$$

where $\mathrm{fbt}_\Delta(8)$ denotes a full binary tree over Δ of height 8. □

Instead of computing output trees (which can be double exponentially bigger than the input tree, as seen in the example) we now want to generate sgraph representations of linear size with respect to the input tree. As computation model for sgraphs we use the top-down tree to graph transducer of [9].

A macro tree transducer generates an output tree by successively applying its term rewrite rules to a sentential form (starting with $\langle q_0, s\rangle$). Due to the presence of parameters the application of such a rewrite rule carries out a second-order tree substitution. The *top-down tree to graph transducer* (ttgt) generalizes the mtt from tree substitution to hypergraph substitution. It is defined just as an mtt, i.e., it consists of ranked alphabets of states, input symbols, and output symbols, an initial state q_0, and a finite set of rewrite rules. For a state q of rank $m \geq 0$ and an input symbol σ of rank k the ttgt has exactly one rule

$$\langle q, \sigma(x_1, \ldots, x_k)\rangle \rightarrow g$$

where g is a hypergraph of rank m over $\langle Q, X_k\rangle \cup \Delta$.

Given an input tree s, the ttgt G generates an output hypergraph by applying its rewrite rules, starting with the initial graph g_0 consisting of a single hyperedge

e labeled $\langle q_0, s\rangle$ and m distinct nodes incident with e, where m is the rank of q_0. A rule $\langle q, \sigma(x_1, \ldots, x_k)\rangle \to g$ can be applied to a hypergraph h that contains a hyperedge e labeled $\langle q, \sigma(s_1, \ldots, s_k)\rangle$, where $s_1, \ldots, s_k$ are input trees. The new hypergraph h' with $h \Rightarrow_G h'$ is obtained from h by removing the edge e and gluing in its place the right-hand side g, in which x_i is replaced by s_i. The gluing is done in such a way that the ith node of e (i.e., the ith node in the sequence $\text{nod}(e)$) is identified with the ith external node of g. Since our ttgts are total deterministic, there is for every input tree s a unique hypergraph $\tau_G(s) = g$ over Δ derived from the initial hypergraph g_0 by $\Rightarrow_G$. We need two restrictions on ttgts: If the right-hand side of every rule is linear in the input variables X_k. then G is *linear*. Obviously, a linear ttgt translates each node of the input tree at most once, i.e., for an input tree s of size n the output tree is obtained from g_0 by at most n rule applications. If all hypergraphs generated by G are sgraphs then G is a *top-down tree to sharing graph transducer* (ttst). Note that, for a ttst G, the function $\tau_G \circ \text{tree}$ is a tree translation (we use *non*standard order for composition $\circ$).

Note that since hypergraph replacement is a generalization of second-order tree substitution, it is not difficult to simulate any mtt by a ttgt such that every output graph is a DAG representation of the corresponding output tree. Parameters are represented by external nodes and parameter copying becomes DAG sharing of a node. This was proved in [9]. We now take that construction and remove all state copying from the mtt by introducing appropriate fan-in/outs. Thus, parameter copying of the mtt becomes DAG sharing of the ttst and state copying of the mtt become fan sharing of the ttst.

Lemma 3. For every mtt M there exists effectively a linear tree to sharing graph transducer G such that $\tau_G \circ \text{tree} = \tau_M$.

Proof. Let $M = (Q, \Sigma, \Delta, q_0, R)$ and let $<_Q$ be a total order on Q. We first fix some auxiliary notions. For every $\sigma \in \Sigma^{(k)}$, $k \geq 1$, $i \in [k]$, and $Q' \subseteq Q$ let

$$Q_{\sigma,i}(Q') := \{q \in Q' \mid \langle q, x_i\rangle \text{ occurs in } \text{rhs}_M(Q, \sigma)\}.$$

For $m \geq 0$, a tree $\zeta \in T_\Delta(Y_m)$, and a natural number i define $\text{graph}(\zeta, i)$ as the hypergraph representation g of ζ, where the jth parameter y_j, $j \in [m]$ is represented by the $(i + j - 1)$th external node of g, and the root node of ζ is represented by the $(i + m)$th external node of g. Note that a symbol δ of rank k is represented by a hyperedge of rank $k + 1$ (with argument and result nodes, see the definition of DAGs in Section 2; for details see, e.g., [6] or [9]).

The idea of the construction of G is as follows. Consider a q_0-rule of M with right-hand side ζ and let g be the graph representation $\text{graph}(\zeta, 1)$. The corresponding rule of G is obtained by merging in g all state calls that are on the same variable x_i. There might be several different states $q_1, \ldots, q_l$ on x_i, and there might be several occurrences of $\langle q_j, x_i\rangle$ in g. We remove from g all edges labeled x_i and add a new handle labeled $\tilde{Q} = \{q_1, \ldots, q_l\}$ in G which denotes the merging of those states. The rank of $\tilde{Q}$ is the sum of the ranks of $q_1, \ldots, q_l$ (for the parameters) plus l (for the root nodes). If there were $\nu > 1$ distinct edges

labeled $\langle q_j, x_i\rangle$, then we use a fan-in and ν fan-outs each of rank ν. The input to the fan-in are the result nodes of these edges and the output of the fan-in is the node incident with the $\tilde{Q}$-edge that corresponds to the root of q_j. The inputs of the fan-outs are the q_j-parameter nodes of the $\tilde{Q}$-edge and their outputs are the argument nodes of the $\langle q_j, x_i\rangle$ labeled nodes.

Let $G = (P, \Sigma, \text{inc}(\Delta), \{q_0\}, U)$. The states and rules of G are defined by applying the recursive procedure **make_rules** to the initial state $\{q_0\}$.

make_rules$(\tilde{Q})$**{**
Let $\tilde{Q} = \{q_1, \ldots, q_\ell\}$ with $q_1 <_Q q_2 <_Q \cdots <_Q q_\ell$ and $r_j = \text{rank}_Q(q_j)$ for $j \in [\ell]$.
Let $\tilde{Q}$ be a state in P of rank $m = \ell + r_1 + r_2 + \cdots + r_\ell$.
For every $\sigma \in \Sigma^{(k)}$, $k \geq 0$ do{
  Let g be the disjoint union of the graphs
  graph(rhs$_M(q_1, \sigma), 1)$, graph(rhs$_M(q_2, \sigma), 2 + r_1)$, graph(rhs$_M(q_3, \sigma), 3 + r_1 + r_2)$, $\ldots$, graph(rhs$_M(q_\ell, \sigma), \ell + r_1 + r_2 + \cdots + r_{\ell-1})$.
  For every $i \in [k]$ do{
    Add to g new nodes $v_1, \ldots, v_m$ and a new edge e (of rank m)
    labeled $\langle Q_{\sigma,i}(\tilde{Q}), x_i\rangle$ with $\text{nod}(e) = v_1 \cdots v_m$.
    Let $Q_{\sigma,i}(\tilde{Q}) = \{q'_1, \ldots, q'_n\}$ with $q'_1 <_Q \cdots <_Q q'_n$ and let $r'_j = \text{rank}_Q(q'_j)$.
    For $j \in [n]$ let $E_{i,j}$ be the edges in g that are labeled $\langle p_j, x_i\rangle$.
    For every $j \in [n]$ do{
      Remove all edges in $E_{i,j}$ from g.
      Let $E_{i,j} = \{e_1, \ldots, e_\nu\}$.
      Add to g a fan-in of rank ν with output node v_{1+r_1} and
      input nodes $\text{res}(\text{nod}(e_1)), \text{res}(\text{nod}(e_2)), \ldots, \text{res}(\text{nod}(e_\nu))$.
      For every $\mu \in [r'_j]$ do{
        Add to g a fan-out of rank ν with input node v_κ
        for $\kappa = j + (r'_1 + \cdots + r'_{j-1}) + (\mu - 1)$ and with output
        nodes $\text{ar}(\text{nod}(e_1))[\mu], \text{ar}(\text{nod}(e_2))[\mu], \ldots, \text{ar}(\text{nod}(e_\nu))[\mu]$.
      }
    }
  }
  Let the rule $\langle \tilde{Q}, \sigma(x_1, \ldots, x_k)\rangle \to g$ be in U.
}**}**

Correctness of G can be proved as follows. For an sgraph g of rank m and $k \in [m]$ let $\text{tree}(g, k)$ denote the tree obtained by starting the unfolding at the k-th external node of g. For $q \in Q^{(m)}$ let $M_q(s)$ be the normal form of $\langle q, s\rangle(y_1, \ldots, y_m)$ w.r.t $\Rightarrow_M$ (and similarly for G_p, $p \in P$). Correctness, namely $\text{tree}(\tau_G(s)) = \tau_M(s)$ follows from the claim for $q = q_0$ and $\tilde{Q} = \{q_0\}$.

Claim: Let $s \in T_\Sigma$, $\tilde{Q} = \{q_1, \ldots, q_n\} \in P^{(m)}$ with $q_1 < Q \cdots <_Q q_n$, and $j \in [n]$. Then $\text{tree}(G_{\tilde{Q}}(s), m) = M_q(s)$.

Note that in $\text{tree}(G_{\tilde{Q}}, m)$ the $(m - \text{rank}_Q(q) + j)$th external node of $G_{\tilde{Q}}(s)$ is interpreted as y_j. This claim can be proved by induction on the structure of s. The main point is to show how the application of 'tree' distributes through a graph that is obtained by hyperedge replacement (hr). Roughly speaking,

hr become second-order tree substitution. The proof is similar to the one of Lemma 5.3 of [9]. □

Example 4. Consider the mtt M_{dexp} of Example 2. We now apply to it the construction presented in the proof of Lemma 3 to obtain a linear ttst G that computes for every input tree s an sgraph g such that $\text{tree}(g) = \tau_M(s)$. Define $G = (P, \Sigma, \text{inc}(\Delta), \{q_0\}, U)$. The only possible state sets are $\{q_0\}$ of rank 1 and $Q_{a,1}(\{q_0\}) = \{q\}$ of rank 2. Hence $P = \{\{q_0\}^{(1)}, \{q\}^{(2)}\}$. It remains to define the rules in U.

Let us start with the a-rules, and in particular with the $(\{q_0\}, a)$-rule. Let g be the graph representation $\text{graph}(\text{rhs}_M(q_0, a), 1)$ of the tree $\text{rhs}_M(q_0, a) = \langle q, x_1\rangle(\langle q, x_1\rangle(e))$. We now enter the part of the construction that is presented in pseudo code. Since a is of rank $k = 1$ there is only the choice $i = 1$ in the first loop. We now add to g a new edge e labeled $\langle Q_{a,1}(\{q_0\}), x_1\rangle$. The corresponding graph is shown on the left of Figure 4. Now $Q_{a,1}(\{q_0\}) = \{q'_1\}$, $q'_1 = q$, and

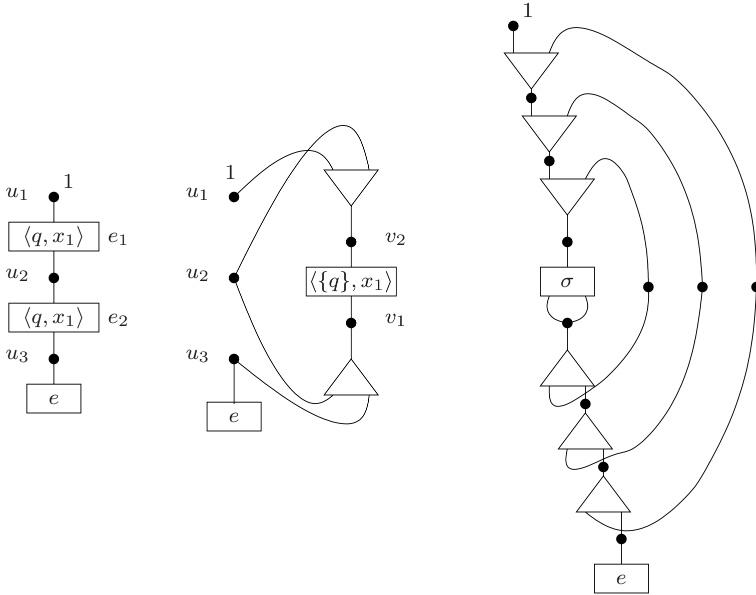

Fig. 4. The graphs $\text{graph}(\text{rhs}_M(q_0, a), 1)$, $\text{rhs}_G(\{q_0\}, a)$, and the sgraph $\tau_G(aaae)$.

$r'_1 = 1$. Thus there is only the choice $j = 1$ in the second loop. Let $E = \{e_1, e_2\}$ be the edges as in the left of Figure 4. We now add a fan-in of rank 2 with output node $v_{1+r'_1} = v_2$ and input nodes $\text{res}(\text{nod}(e_1)) = u_1$ and $\text{res}(\text{nod}(e_2)) = u_2$ (see the figure). For the final loop the only choice is $\mu = 1$. Then $\kappa = 1$. Hence, we add a fan-out of rank 2 with input node v_1 and output nodes $\text{ar}(\text{nod}(e_1)) = u_2$ and $\text{ar}(\text{nod}(e_2)) = u_3$. The final right-hand side g of the $(\{q_0\}, a)$-rule of G is shown in the middle of Figure 4.

It should be clear how to construct the $(\{q\}, a)$-rule of G. Let us consider the computation of G for the input tree $s = aaae$. The resulting sgraph is shown in the right of Fig. 4. □

It should be intuitively clear how to realize a linear ttst G on a RAM A. The input to A is a tree s represented as pointer structure. Then A computes the output graph $\tau_G(s)$ (as a pointer structure) by applying the rules of G successively. Clearly, the application of a rule $l \to r$ can be done in time $O(|r|)$. Since a linear ttst computes the output sgraph for an input tree of size n by application of at most n rules, we obtain our first main result.

Theorem 5. For every mtt M there is effectively a RAM that computes, given an input tree s, an output sgraph g with $\text{tree}(g) = \tau_M(s)$ in time $O(|s|)$.

XML Translations. In an XML document a node (denoted by `<tag>` and `</tag>`) has arbitrarily many subtrees (viz., the sequence of trees between `<tag>` and `</tag>`). Hence, an XML document naturally represents an unranked tree (seen as a graph with two sorts of edges: child edges and sibling edges). In contrast to that, classical tree language theory is mainly concerned with ranked trees. Of course every unranked tree can be represented by a binary tree (obtained from the unranked tree by simply deleting all child edges to non-first children).

Several tree transducer models that work directly on unranked trees are more powerful than their ranked counterparts. E.g., for the top-down tree transducer this was proved in [18]. For the macro tree transducer this was recently proved in [20]: their unranked version of mtt, the *macro forest transducer* (mft), is strictly more powerful than mtts on binary encodings. Even though every mft can be simulated on binary trees by the composition of two mtts, the complexity of type checking is the same for an mft as for just one mtt. Therefore the mft deserves attention. In this section we show how to generate sgraph representations of output forests of mfts.

Let us consider an example of an mft. In fact, it is the transducer F_{dexp} used in [20] to prove that mfts are more powerful than mtts on encodings. For an alphabet Σ the *set $\mathcal{F}_\Sigma$ of (unranked) forests over Σ* is defined by the context-free grammar with productions

$$\begin{array}{llll} F \to T & | & FF & \\ T \to \bot & | & a(F) & \quad a \in \Sigma \end{array}$$

The mft is the natural generalization of the mtt to the forest defined above: a rule is of the form $\langle q, \sigma(x_1)x_2\rangle(y_1, \ldots, y_m) \to f$ where f is a forest over $\Sigma \cup Y_m$ plus elements of $\langle Q, X_2\rangle$ which occur ranked. The mft F_{dexp} has $\Sigma = \{a\}$ and $\Delta = \{b\}$ as input and output alphabets, respectively, and the following two rules.

$$\begin{array}{ll} \langle q, a(x_1)x_2\rangle(y_1) & \to \langle q, x_2\rangle(\langle q, x_2\rangle(y_1)) \\ \langle q, \bot\rangle(y_1) & \to y_1 y_1 \end{array}$$

It starts computing with $\langle q, s\rangle(b)$ and translates the input forest s of width n (i.e, consisting of n concatenated trees) into the forest f_n consisting of the concatenation of 2^{2^n} trees b. Using the usual encoding of forests by binary trees, the corresponding ranked translation has double exponential height increase (take the forest $s_n = a^n$ as input) and therefore cannot be realized by any mtt.

If we try to construct a linear top-down tree to sgraph transducer for F_{dexp} following the construction of Lemma 3 then we get a wrong ttst which generates the concatenation of only 2^{n+1} trees b. The reason is that the copying by concatenation present in the second rule (with rhs yy) cannot be realized by DAG sharing. In fact, it is easy to see that no linear ttst can generate sgraph representations of f_n, taking s_n as input. This is because sgraphs for linear structures (like strings or monadic trees) have a compression rate of at most $1/\log n$.

Now let us try to simulate an mft by a *non* linear ttst. Instead of sharing states on the same input variable x_i (as in the construction of Lemma 3) we now simply take the state copying of the mft over into the rules of the ttst. In order to realize the copying of parameters we use fan-in/outs. A state now has two tentacles for each parameter (thus it has $2m$ argument nodes if there are m parameters). In a computation they will be incident with the begin and end nodes of the string of trees of the actual parameter forest. Similarly, every state has two result nodes which correspond to the begin and end nodes of the forest it will compute. If we apply this construction to the mft F_{dexp} then we obtain the ttst G_{dexp} which has the two rules depicted in Fig. 5. Obviously, if the original

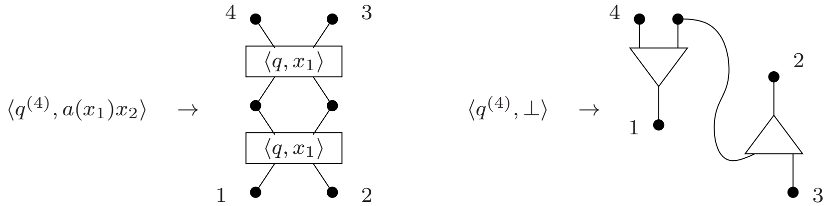

Fig. 5. The rules of the ttst G_{dexp}.

mft F is linear in the input variables, then so is the resulting ttst G. We obtain the following lemma.

Lemma 6. For every mft F there exists effectively a top-down tree to sgraph transducer G such that $\tau_G \circ \text{tree} = \tau_F$. If F is X-linear then G is linear.

Consider an mft F that is linear in the parameters. Maybe a construction similar to the one of Lemma 3 can be used to show that there is *linear* ttst that computes corresponding sgraphs. But this remains to be proved.

Instead, let us try to find a more liberal condition on the input variables. It turns out that the linearity condition can be weakened into the "finite copying" restriction, without changing the corresponding class of translations. An mft is *finite copying in the input* if there is a $c > 0$ such that the number of states that translate a certain node of the input tree is bounded by c.

Theorem 7. For every macro forest transducer M there is effectively a RAM that computes, given an input tree s, an output sgraph g with $\text{tree}(g) = \tau_M(s)$ in time $O(2^{|s|})$. If M is finite copying in the input, then g is computed in time $O(|s|)$.

Proof. (sketch) For the general case the result follows from Lemma 6. Assume now that M is finite copying in the input. We can decompose M into a finite copying top-down forest transducer T followed by an mft M' that is linear in the input variables. The idea is the one of Theorem 4.8 of [8]: every mtt can be decomposed into a top-down tree transducer followed by a so called "YIELD" mapping (it interprets its input symbols as substitution operations and in this way realizes the second-order tree substitution inherent in an mtt). In fact, for mtts that are linear in the parameters this was shown in Lemma 2 of [6]. Its generalization to nonlinear parameters and to forests should be straightforward. Finite copying top-down forest transducers are obviously of linear size increase. Since every mft can be simulated by the composition of two mtts by Theorem 9 of [20] $\tau_T(s)$ can be computed in time linear in $\ell = |s| + |\tau_T(s)|$ by Theorem 15 of [17]. By Lemma 6 and the fact that M' is linear, $\tau_{M'}(\tau_T(s)) = \tau_M(s)$ can be computed in time linear in ℓ. □

Removal of Garbage. Consider a ttst G. For an input tree s the output $\tau_G(s)$ is an sgraph with one external node v. The tree represented by $\tau_G(s)$ is obtained by unfolding the sgraph rooted at the node v. However, $\tau_G(s)$ might contain subgraphs that are not at all connected to the sgraph rooted at v. Such parts of $\tau_G(s)$ are called *garbage*. But in an sgraph there can be even more garbage: nested pairs of fan-ins/outs (as in the right of Fig. 4) might share nothing whatsoever, i.e., the innermost fan-in is directly connected to the innermost fan-out.

If we consider the translation of Lemma 3 then these forms of garbage will be generated when if the underlying mtt (1) *deletes* a parameter, i.e., if y_j does not occur in the right-hand side of a state (of rank $\geq j$) or when it (2) *erases* a state, i.e., if a state of rank one has a rule with right-hand side y_1. Thus, if an mtt is nondeleting in the parameters and nonerasing in the states, then the corresponding ttst will not generate garbage. It was proved in Lemma 7.11 of [5] that for every mtt there is an equivalent mtt (with regular look-ahead) that is nondeleting and nonerasing. Thus we obtain the following lemma.

Lemma 8. For every mtt M there is a linear garbage-free ttst G such that $\tau_G \circ \text{tree} = \tau_M$.

It is probable (but remains to be proved) that a similar result holds for mfts (with "linear garbage-free" replaced by "garbage-free").

4 Exploring Sharing Graphs

Sgraphs can be used as compressed representations of trees. In the previous section it was shown how to generate sgraphs by means of tree to graph transducers. But once we have computed an sgraph, what can we do with it (other

than decompressing)? In the Introduction we have claimed that the basic tree operations are preserved when moving from a tree to a compressed sgraph. In this section we want to make this claim more precise by showing that any algorithm that reads the tree (by moving along its edges) can also be realized on the sgraph with a slow down that is (per move) linear in the size of the sgraph. In order to do this we first show that any sgraph can be represented by a particular context-free tree grammar (in the sense that they represent the same tree).

In a context-free tree grammar the set N of nonterminals is ranked and a production is of the form $A(y_1, \ldots, y_m) \to t$ where $A \in N^{(m)}$, $m \geq 0$, and t is a tree over N and the terminal ranked alphabet Δ. The grammar is *simple* it is linear, nondeleting, and nonerasing, i.e., if each parameter y_j occurs exactly in t and $t \neq y_1$. In a *straight-line* grammar the set of productions can be written as $A_1 \to r_1, \ldots, A_n \to r_n$ such that all A_i are pairwise different and nonterminals A_j occurring in r_i have $j > i$. We now illustrate how an sgraph g can be transformed into a straight-line simple context-free tree grammar G.

The *scope* of a fan-in f in g is the sharing subgraph of g rooted at the output of f and ending at leaves or fan-outs matching f. For an sgraph g we introduce a non-terminal A and the production $A \to t$ where the tree t is obtained as follows. Starting at the root of g we copy the tree top-down until a fan-in edge e is encountered (the same for DAG copying). We introduce a new nonterminal e of rank m, where m is the number of matching fan-outs for e. At the i-th input to e we stop copying and add the subtree $e(t_1, \ldots, t_m)$ to t, where t_j is generated by applying the copy procedure to the i-th output node of the j-th fan-out of e. If e is encountered again no new non-terminal is introduced. After t has been generated we apply the same procedure to the new non-terminals $e_1, \ldots, e_k$ (one for each fan-in) and their respective scopes; then, when the j-th matching fan-out (thus, they should be ordered) is encountered, the parameter y_j is generated. As the reader may verify, application of this procedure to the sgraph of Fig. 1 (where we assume that the lower shared c-node has implicitly a fan-in with no fan-outs) produces (up to renaming) precisely the context-free tree grammar shown in Fig. 2.

Given a straight-line simple cf tree grammar G (that generates $\{t\}$) we now show how to simulate the movement through the nodes of t. This is done without unfolding the tree, but by using a stack the size of which is bounded by the size of G. Let $A_1, \ldots, A_n$ be the nonterminals of G, $t_1, \ldots, t_n$ their respective right-hand sides and A_1 the initial nonterminal. We will consider *nested derivations* of G in order to simulate the navigation through the nodes of t. This means that every nonterminal to which a production is applied (except A_1) was introduced by the previous production. By definition the length of such a derivation is at most n. If $i_1, \ldots, i_m$ are the indices of the productions in such a derivation then $i_1 < \cdots < i_m$ and $m \leq i_m - i_1 + 1 \leq n$.

Nested derivations have the useful property that they can be represented in a compact way by storing the indices of the applied productions and, for each right-hand side, a pointer to the non-terminal that will be replaced in the following step. To this purpose we use *pointed productions*, i.e. productions where

we have selected one node in the right-hand side. We denote pointed productions as pairs $p = (j, u)$, where j is the index of the production and u is the path to a node in its right-hand side. A *stack* $\sigma = [(j_1, u_1) \dots (j_m, u_m)]$ is a sequence of pointed productions where $j_1 = 1$, each pointer u_l $(1 \leq l < m)$ refers to a node in the right-hand side of the A_{j_l}-production labeled $A_{j_{l+1}}$, and the last (top) pointer u_m refers to a terminal node. It is clear that such a stack corresponds to a sentential form with one selected terminal node, and therefore it identifies a unique node of the tree t. The empty stack [] has no node corresponding to it and will be interpreted as "find the root node". We implement the operations $\mathbf{down}_i$ and **up** as operations on stacks.

For a stack σ we implement the operation $\mathbf{down}_i(\sigma)$, $i \in \mathbb{N}$ as follows. If σ is the empty stack then we look for the shortest left-most derivation that starts from the initial symbol A_1 and produces a sentential form with the root labeled by a non-terminal. Notice that the number of steps required and the size of the stack are bounded by n. If $\sigma = [(j_1, u_1) \dots (j_m, u_m)]$ is not empty we move the pointer u_m to the i-th son of the selected node. Now, we have three possibilities:

(1) If the new pointed node has a terminal label we are finished.
(2) If the new pointed node has a non-terminal label $A_{j_{m+1}}$ we extend the stack by performing a left-most derivation starting from $A_{j_{m+1}}$ and ending with a sentential form that has a terminal node at the root. The resulting stack represents also a nested derivation, therefore the size of the stack and the cost of the $\mathbf{down}_i$ operation are bounded by n.
(3) If the new pointed node is a parameter, then we must backtrack (pop) in order to find an earlier right-hand side in which the parameter is instantiated. We may need several backtracking steps, since a variable can always be replaced by another variable. The backtracking stops eventually because the initial symbol contains no variables. Now we proceed as in (2).

Given a stack σ, we implement **up** in the following way. If $\sigma = \varepsilon$, then there is nothing to be done. Otherwise, we need to find a position in a right-hand side where the parent node of the current node is a terminal symbol. We first backtrack, trying to find a (j, u) with $u \neq \varepsilon$. If we end up with the empty stack then we are finished (because the current node *was* the root). Otherwise let $u = u'i$ with $i \in \mathbb{N}$ and change u into u'. If u' is terminal then we are finished. Otherwise we rewrite the nonterminal at u', extend the stack appropriately, and position the pointer in the right-hand side on the father u'' of the unique occurrence of y_i. Note that u'' exists because G is simple. We repeat this rewrite procedure until we obtain a terminal node u''.

The above construction has shown how to simulate on a simple context-free tree grammar G the tree operations $\mathbf{down}_i$ and **up** in time bounded by the size of G.

Acknowledgments. The first author is grateful to Markus Lohrey for providing many useful references, and in particular for the introduction to the concept of sharing graphs.

References

1. A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.
2. P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In J. C. Freytag et al., editor, *Proc. VLDB'2003*. Morgan Kaufmann, 2003.
3. J. Engelfriet. Context-free graph grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*. Springer-Verlag, 1997.
4. J. Engelfriet and L. Heyker. Context-free hypergraph grammars have the same term-generating power as attribute grammars. *Acta Informatica*, 29:161–210, 1992.
5. J. Engelfriet and S. Maneth. Macro tree transducers, attribute grammars, and MSO definable tree translations. *Inform. and Comput.*, 154:34–91, 1999.
6. J. Engelfriet and S. Maneth. Tree languages generated by context-free graph grammars. In H. Ehrig et. al., editor, *Proc. TAGT'98*, volume 1764 of *LNCS*, pages 15–29. Springer-Verlag, 2000.
7. J. Engelfriet and S. Maneth. A comparison of pebble tree transducers with macro tree transducers. *Acta Informatica*, 39:613–698, 2003.
8. J. Engelfriet and H. Vogler. Macro tree transducers. *J. of Comp. Syst. Sci.*, 31:71–146, 1985.
9. J. Engelfriet and H. Vogler. The translation power of top-down tree-to-graph transducers. *J. of Comp. Syst. Sci.*, 49:258–305, 1994.
10. M. Frick, M. Grohe, and C. Koch. Query evaluation on compressed trees (extended abstract). In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science – LICS'2003*, pages 188–197. IEEE, 2003.
11. F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*. Springer-Verlag, 1997.
12. S. Guerrini. A general theory of sharing graphs. *TCS*, 227:99–151, 1999.
13. J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. *Intern. J. of Foundations of Comput. Sci.*, 1:425–447, 1990.
14. J. Lamping. An algorithm for optimal lambda calculus reductions. In *Proc. POPL'1990*, pages 16–30. ACM Press, 1990.
15. E. Lehman and A. Shelat. Approximation algorithms for grammar-based compression. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'2002)*, pages 205–212. SIAM Press, 2002.
16. H. Liefke and D. Suciu. XMill: An efficient compressor for xml data. In W. Chen et. al., editor, *Proc. ACM Conference on Management of Data*, pages 153–164. ACM, 2000.
17. S. Maneth. The complexity of compositions of deterministic tree transducers. In M. Agrawal and A. Seth, editors, *Proc. FSTTCS 2002*, volume 2556 of *LNCS*, pages 265–276. Springer-Verlag, 2002.
18. S. Maneth and F. Neven. Recursive structured document transformations. In R. Connor and A. Mendelzon, editors, *Revised Papers DBPL'99*, volume 1949 of *LNCS*, pages 80–98. Springer-Verlag, 2000.
19. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *J. of Comp. Syst. Sci.*, 66:66–97, 2003.
20. T. Perst and H. Seidl. Macro forest transducers. To appear in *IPL*.
21. W. Plandowski. Testing equivalence of morphisms on context-free languages. In Jan van Leeuwen, editor, *Proc. Second European Symposium on Algorithms – ESA'94*, volume 855 of *LNCS*, pages 460–470. Springer-Verlag, 1994.