# The Formal Language Theory Column

by

**Giovanni Pighizzini**

Dipartimento di Informatica
Università degli Studi di Milano
20135 Milano, Italy
pighizzini@di.unimi.it

# On DAG Languages and DAG Transducers

Frank Drewes
Department of Computing Science
Umeå University (Sweden)
drewes@cs.umu.se

**Abstract**

We review recent results regarding DAG automata and regular DAG languages and point out some open problems that may be interesting to work on. Moreover, a notion of DAG transducers is suggested.

## 1 Introduction

The traditional objects studied in formal language theory are string languages, string transductions, and the devices that define or compute them. However, for many application areas more general structures than strings need to be considered, which is why an almost equally rich theory addressing the generation and transformation of trees and graphs has been developed over the years. The generalization of regular (or right-linear) string grammars and finite-state string automata to regular tree grammars and (bottom-up) finite-state tree automata is the easiest step. There are almost no surprises at all, and all the basic properties, results, and algorithms carry over, without essential losses in efficiency. Tree transductions and context-free tree grammars are interesting because they provide more challenges, yet offer interesting results. Their theory is also very well understood today.

Graph languages and graph transformations take this a step further. However, graphs are somewhat complicated objects. A string has a beginning and an end, which offers a natural sense of direction for an automaton to process it. Similarly, a tree can be processed from the root towards the leaves or vice versa. For a graph, a similar sense of direction is missing, which at the very least makes it difficult to conceive a reasonable notion of *deterministic* graph automata. This may explain why there does not seem to be a widely accepted notion of graph automata.

However, there exists a structure in between trees and graphs that does offer a similar sense of direction as the tree: the directed acyclic graph (DAG). A DAG can be seen as a tree (or forest) with sharing of common subtrees allowed. Like a

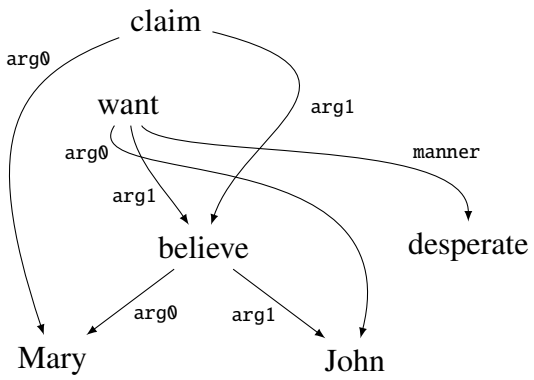



Figure 1: DAG capturing (the basic aspects of) the meaning of "John desperately wants Mary to believe him, and she claims she does". The first outgoing edge of 'want', 'believe' or 'claim' (labelled by `arg0`) represents the agent relation whereas the second (labelled by `arg1`) represents the patient relation and `manner` modifies the wanting.

tree, a DAG has roots and leaves (nodes without incoming and outgoing edges, respectively), hence it, too, offers obvious directions in which it can be processed by an automaton: either top down or bottom up. Interestingly, this has hitherto not led to more than the occasional paper formalizing and studying some notion of DAG automata. However, now interest in such automata rises, triggered by new developments in natural language processing, and especially by research on semantic parsing and the *Abstract Meaning Representation* (AMR, see [2]). Abstracting from some details, a semantic representation of a natural language sentence is a (possibly multi-rooted) DAG whose node labels are the concepts occurring in the sentence and whose edges represent their "argument" relation. A (simplified) example is shown in Figure 1. Note that, while the DAG could be "unfolded" into a tree, this would be inappropriate because it would not faithfully represent the fact that John wants Mary to believe *him* (rather than some other John). Moreover, what John wants is identical to what Mary claims she does.

DAG automata for semantic representations such as AMR were first proposed by Quernheim and Knight [19, 20], taking inspiration from early work by Kamimura and Slutzki [13]. This work was continued in [3, 4, 7]. Older work on (other types of) DAG automata can be found in [14, 5, 21, 12, 18, 6, 8, 1, 17, 10].

The remainder of this paper reviews some results of the papers mentioned above, and proposes potential avenues for future research. In particular, different options regarding the formalization of DAGs and DAG automata are discussed, a corresponding notion of regular DAG grammars is defined, and a class of DAG transducers is suggested.

## 2 DAGs and DAG Automata

Even more so than in the tree case, a theory of DAG automata may be built upon different variants of DAGs and DAG automata. To mention some of the most obvious options, one may consider labelled or unlabelled nodes and edges; DAGs may be ordered in the sense that the incoming and the outgoing edges of each node form a sequence, or unordered, in which case they simply form a set. DAGs may be singly rooted or allowed to have any number of roots. A rule of a DAG automaton may process nodes with a fixed in- and out-degree, or there may be a mechanism by which arbitrarily many incoming and outgoing edges can be processed. Compared to the world of tree automata, the latter distinction corresponds to the one between ranked and unranked tree automata.

The example illustrated in Figure 1 seems to suggest that, if motivated by semantic representation similar to AMR, DAGs should be node and edge labelled, unordered (at least as regards incoming edges), multi-rooted, and unranked. The latter quite obviously holds for incoming edges, but also for outgoing ones as there can be a potentially arbitrary number of modifiers such as `manner` attached to a concept. However, to study basic formal properties it is useful to choose a simple model, especially if it can be shown to be able to simulate more complex ones. Therefore, we shall consider a simplified setting in this paper, taken from [4]: regular DAG languages consist of node-labelled ordered DAGs of bounded rank, with no restriction on the number of roots. We shall discuss later which of these assumptions make a difference and which do not. Most do not.

Let us first compile some basic notation and terminology. For $n \in \mathbb{N}$, we let $[n] = \{1, \dots, n\}$. As usual, an *alphabet* is a finite set $\Sigma$ whose elements are called symbols. The set of finite strings over $\Sigma$ is denoted by $\Sigma^*$, the empty string by $\lambda$, and the powerset of $\Sigma$ by $2^\Sigma$. By abuse of notation the canonical extensions of a function $f\colon \Sigma \to \Delta$ to functions from $\Sigma^*$ to $\Delta^*$ and from $2^\Sigma$ to $2^\Delta$ are denoted by $f$ as well. For $w \in \Sigma^*$, the set of all symbols appearing in $w$, i.e., the smallest set $S$ such that $w \in S^*$, is denoted by $[w]$. Given a binary relation $\to \subseteq A^2$ on a set $A$, we denote its transitive closure by $\to^+$.

**Definition 2.1 (graph).** A $\Sigma$-*graph* is a tuple $G = (V_G, E_G, lab_G, in_G, out_G)$ consisting of

- finite disjoint sets $V_G$ and $E_G$ of *nodes* (or *vertices*) and *edges*, respectively,
- a *node labelling* $lab_G\colon V_G \to \Sigma$, and
- mappings $in_G, out_G\colon V \to E^*$ which assign to each node a sequence of ingoing and outgoing edges in such a way that each $e \in E_G$ occurs exactly once in all the $in_G(u)$, $u \in V_G$, and exactly once in all the $out_G(v)$, $v \in V_G$.

Thus, every edge $e$ has a unique *source* $src_G(e)$ and a unique *target* $tar_G(e)$, which are the nodes $u$ and $v$, respectively, such that $e$ occurs in $out_G(u)$ and in

$in_G(v)$. Notions such as paths, cycles, directedness of paths, the empty $\Sigma$-graph $\emptyset$, and the disjoint union $G \uplus G'$ of $\Sigma$-graphs are defined in the usual way. For future use, let us be a bit more precise regarding paths: a path from $u \in V_G$ to $v \in V_G$ is an alternating sequence $v_0 e_1 v_1 \cdots e_n v_n$ of edges such that $v_0 = u$, $v_n = v$, and $\{v_{i-1}, v_i\} = \{src_G(e_i), tar_G(e_i)\}$ for all $i \in [n]$.

A vertex $v \in V_G$ is a *leaf* if $out_G(v) = \lambda$ (the empty edge sequence). Deviating slightly from traditional mathematical terminology, we call $v \in V_G$ a *root* if $in_G(v) = \lambda$. Thus, the two notions are duals; $v$ being a root does *not* imply that every other vertex can be reached from $v$.

**Definition 2.2 (DAG and DAG automaton).** A $\Sigma$-*DAG* (or simply *DAG*) is a $\Sigma$-graph that does not contain any directed cycle. The set of all nonempty connected $\Sigma$-DAGs is denoted by by $\mathcal{D}_\Sigma$. A *DAG language* is a subset of $\mathcal{D}_\Sigma$, for some alphabet $\Sigma$.

A *DAG automaton* is a triple $A = (Q, \Sigma, R)$ consisting of a set of *states* $Q$, an alphabet $\Sigma$, and a set $R$ of *rules*, all finite. Each rule is of the form $\alpha \overset{\sigma}{\longleftarrow} \beta$ with $\sigma \in \Sigma$ and $\alpha, \beta \in Q^*$. Such a rule is also called a $\sigma$-rule, and the state sequences $\alpha$ and $\beta$ are its *head* and *tail*, respectively.

A *run* of $A$ on a $\Sigma$-DAG $D$ is a mapping $\rho\colon E_D \to Q$ such that the rule

$$\rho(in_D(v)) \overset{lab_D(v)}{\longleftarrow} \rho(out_D(v))\ ^{1}$$

is in $R$ for all $v \in V_D$. *$A$ accepts $D$* if a run on $D$ exists. The *DAG language accepted by $A$* is the set $L(A) = \{D \in \mathcal{D}_\Sigma \mid A \text{ accepts } D\}$. A DAG language that can be accepted by a DAG automaton is called a *regular DAG language*.

Note that, by the preceding definition, only DAGs consisting of a single (nonempty) connected component are in $\mathcal{D}_\Sigma$ and thus in $L(A)$. This is a convenient convention because every $\Sigma$-DAG $D$ is a disjoint union of DAGs $D_1, \ldots, D_k \in \mathcal{D}_\Sigma$ for some $k \in \mathbb{N}$, and thus $D$ admits a run if and only if each $D_i$ does. In other words, the language *all* $\Sigma$-DAGs accepted by $A$ is simply the closure $L(A)^{\uplus} = \{D_1 \uplus \cdots \uplus D_k \mid k \in \mathbb{N}, D_1, \ldots, D_k \in L(A)\}$ of $L(A)$ under disjoint union. Therefore, $L(A)$ is the more reasonable object to study, as questions such as the emptiness and finiteness problem are uninteresting for $L(A)^{\uplus}$ (which is never empty and is finite iff $L(A) = \emptyset$ iff $L(A)^{\uplus} = \{\emptyset\}$).

Clearly, trees over $\Sigma$ "are" those elements of $\mathcal{D}_\Sigma$ in which each node has in-degree at most 1 (the root has in-degree 0 and the others have in-degree 1). A DAG automaton in which all rule heads are in $\{\lambda\} \cup Q$ is thus nothing else than an ordinary finite-state tree automaton, and the tree languages accepted by DAG automata are exactly the regular tree languages. (Note that, since DAG automata explicitly distinguish rules applying to roots from those applying to

[1] Recall that $\rho$ is extended to sequences of nodes.

non-roots, no final or initial states are required. If a bottom-up tree automaton has a rule $\sigma(q_1, \ldots, q_k) \to q$, then the corresponding DAG automaton contains the rule $q \overset{\sigma}{\longleftarrow} q_1 \cdots q_k$, and if $q$ is a final state it additionally contains the rule $\lambda \overset{\sigma}{\longleftarrow} q_1 \cdots q_k$.)

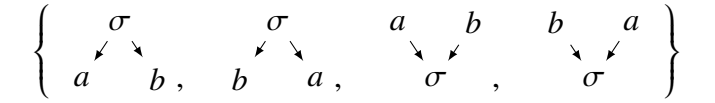
The reader may have noticed that the definitions of DAGs and DAG automata are entirely symmetric with respect to the direction of edges: the *dual* of a DAG is obtained by reversing all edges, thus turning its DAGs "upside down", and the dual of a DAG automaton $A$ is obtained by turning every rule $\alpha \overset{\sigma}{\longleftarrow} \beta$ into $\beta \overset{\sigma}{\longleftarrow} \alpha$. Obviously, the dual of $A$ accepts the dual of $L(A)$, and thus the class of regular DAG languages is invariant under taking duals.

Let us briefly discuss some of the alternatives mentioned earlier.

### Edge Labels and Unordered DAGs

Edge labels can easily be simulated by representing an $\ell$-labelled edge from $u$ to $v$ by an intermediate node $w$ with label $\ell$, and turning the original edge into two unlabelled edges from $u$ to $w$ and from $w$ to $v$. Moreover, if we regard a rule $\alpha \overset{\sigma}{\longleftarrow} \beta$ as a shorthand for all rules $\alpha' \overset{\sigma}{\longleftarrow} \beta'$ with $\alpha'$ and $\beta'$ being reorderings of $\alpha$ and $\beta$, respectively, we effectively view DAGs as unordered DAGs.

Thus, essentially no power is lost by considering ordered DAGs. Instead, something is gained, namely a meaningful notion of determinism: a DAG automaton $A$ is *top-down deterministic* if it does not contain any distinct rules $\alpha \overset{\sigma}{\longleftarrow} \beta$ and $\alpha' \overset{\sigma'}{\longleftarrow} \beta'$ with $(\alpha, \sigma) = (\alpha', \sigma')$. $A$ is *bottom-up deterministic* if its dual is top-down deterministic. Clearly, these notions coincide with the usual ones when restricted to tree automata (in their disguise as a special case of DAG automata). It is well-known that, in contrast to bottom-up deterministic tree automata, top-down deterministic ones cannot accept all regular tree languages, a counterexample being the finite tree language $\{\sigma(a, b), \sigma(b, a)\}$. Viewed as a DAG language, this means that it is not a top-down deterministic regular DAG language, and thus its dual is not bottom-up deterministic and their union



is neither. In other words, there are (finite) regular DAG languages which are neither top-down nor bottom-up deterministically regular [3].

### Unranked DAG Automata

The definition of DAG automata may be generalized to the unranked case in a similar way as for tree automata; see also [17, 7], where this is done in the un-

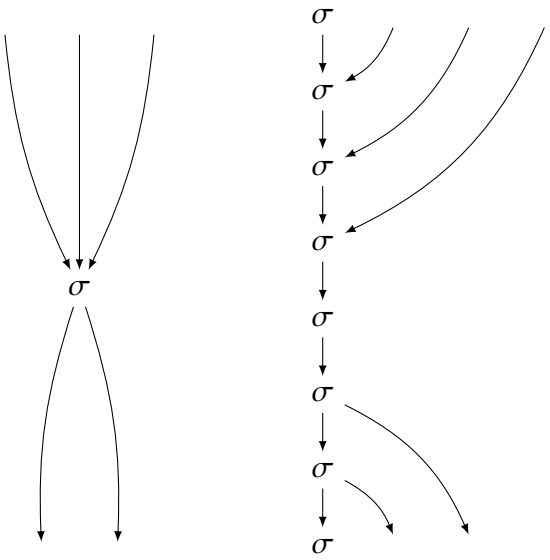



Figure 2: Binarizing a node of in-degree 3 and out-degree 2

ordered setting. One simply generalizes rules by letting their heads and tails be regular expressions over $Q$.[2] Then $\rho\colon E_D \to Q$ is a run if for all $v \in V_D$ there is a rule $\alpha \overset{lab_D(v)}{\longleftrightarrow} \beta$ in $R$ such that $\rho(in_D(v)) \in L(\alpha)$ and $\rho(out_D(v)) \in L(\beta)$.

As in the tree case, there is a close correspondence between ranked and unranked DAG automata via an encoding that replaces every node by a sub-DAG in which no node has more than two incoming or outgoing edges. A simple encoding of this type is illustrated in Figure 2. Now, an unranked DAG automaton can be binarized accordingly. To see this, consider a rule $\alpha \overset{lab_D(v)}{\longleftrightarrow} \beta$ (where $\alpha$ and $\beta$ are now regular expressions over $Q$). For $\gamma \in \{\alpha, \beta\}$, let $A_\gamma = (Q_\gamma, Q, \delta_\gamma, q_\gamma, F_\gamma)$ be a nondeterministic finite-state automaton (NFA) accepting $L(\gamma)$.[3] Then the binarized (ranked) DAG automaton contains the following rules simulating $A_\alpha$ and $A_\beta$ on the binarized sub-DAG:

1. $\lambda \overset{\sigma}{\longleftrightarrow} q_\alpha$ (start in initial state of $A_\alpha$), $qp \overset{\sigma}{\longleftrightarrow} q'$ for all $q' \in \delta_\alpha(q, p)$ (process the next incoming edge, labelled $p \in Q$), and $q \overset{\sigma}{\longleftrightarrow} q_\beta$ for all $q \in F_\alpha$ (switch from final state of $A_\alpha$ to initial state of $A_\beta$), and
2. $q \overset{\sigma}{\longleftrightarrow} q'p$ for all $q' \in \delta_\beta(q, p)$ (process the next outgoing edge, labelled $p \in Q$) and $q \overset{\sigma}{\longleftrightarrow} \lambda$ for all $q \in F_\beta$ (finish successfully).

[2] [17, 7] do this for the unordered case and require that the regular expressions used specify commutative regular languages.

[3] Thus, $A_\gamma$ consists of the set $Q_\gamma$ of states, the input alphabet $Q$, the transition function $\delta_\gamma\colon Q_\gamma \times Q \to 2^{Q_\gamma}$, the initial state $q_\gamma \in Q_\gamma$, and the set $F_\gamma \subseteq Q_\gamma$ of final states.

Let $bin(D)$ denote the binarized version of a DAG $D$, and let $A$ be the original unranked DAG automaton. Then it should be clear that the binarized DAG automaton accepts the DAG language $bin(L(A))$. As a consequence, almost all conceivable decidability and undecidability results and many other properties can be transferred from ranked to unranked DAG automata and vice versa.

In [7] an additional, more general binarization is presented, which can be useful from an efficiency point of view. Rather than replacing every node by a spine of a fixed form, it creates the sub-DAG to be used on the basis of a given (binary) tree decomposition of the DAG. This results in a more efficient membership test, provided that a good tree decomposition of the input DAG is known.

# 3 DAG Grammars

As is the case for finite-state tree automata, a corresponding grammar type can be defined in a straightforward way. By duality, regular DAG grammars can be defined from a top-down or bottom-up perspective. Here we use the former, in order to maintain the analogy with regular tree grammars.

**Definition 3.1 (regular DAG grammar).** A (top-down) *regular DAG grammar* is a triple $G = (\Xi, \Sigma, R)$ consisting of disjoint finite alphabets $\Xi$ and $\Sigma$ of *nonterminals* and *terminals*, and a finite set $R$ of rules. Each rule is of the form $\alpha \xrightarrow{\sigma} \beta$, where $\alpha, \beta \in \Xi^*$ and $\sigma \in \Sigma$.

Let $D$ be a $(\Sigma \cup \Xi)$-DAG in which all nodes with labels in $\Xi$ are leaves of in-degree 1. If $R$ contains a rule $\xi_1 \cdots \xi_k \xrightarrow{\sigma} \xi'_1 \ldots, \xi'_\ell$ and $u_1, \ldots, u_k \in V_D$ are pairwise distinct nodes with $lab_D(u_i) = \xi_i$ for all $i \in [k]$, then $D \to_R D'$ where $D'$ is obtained from $D$ by

1. merging $u_1, \ldots, u_k$ into a single node $u$ with $lab_{D'}(u) = \sigma$ and $in_{D'}(u) = in_D(u_1 \cdots u_k)$, and
2. adding fresh edges $e_1, \ldots, e_\ell$ and leaves $v_1, \ldots, v_\ell$ with $out_{D'}(u) = e_1 \cdots e_\ell$, $in_{D'}(v_i) = e_i$, and $lab_{D'}(v_i) = \xi'_i$ for $i \in [\ell]$.

The *DAG language generated by* $G$ is $L(G) = \{D \in \mathcal{D}_\Sigma \mid \emptyset \to_R^+ D\}$.

The three components of regular DAG grammars are, disregarding the slight differences in names and notations, the same as those of DAG automata. Thus, a DAG automaton $A$ may alternatively be viewed as a regular DAG grammar $G$ (consisting of the same components), and vice versa. Moreover, it is straightforward to show that $L(A) = L(G)$, because every run of $A$ on a DAG $D$ gives rise to a corresponding derivation $\emptyset \to_R^+ D$, and every such derivation gives rise to a run. Thus, we have the following:

**Observation 3.2.** Regular DAG grammars accept precisely the regular DAG languages.

Disregarding the actual DAG being generated, a regular DAG grammar may be viewed as a producer-consumer system in which each rule $\alpha \xrightarrow{\sigma} \beta$ consumes a finite number of nonterminals (those in $\alpha$) and produces some new nonterminals (those in $\beta$). A derivation starts with no nonterminals at all and terminates, after one or more steps, in the same situation. Thus we have a vector addition system or Petri net, where places are elements of $\Xi$ and the transitions consume and produce tokens representing occurrences of nonterminals.

Let us briefly recall Petri nets. An element of $\mathbb{N}^{\xi}$, i.e., a finite multiset with elements in $\Xi$ or a vector indexed by $\Xi$, is a *configuration*. The *zero configuration* is the zero vector $\mathbf{0}$. The order $\leq$ on configurations as well as their addition and subtraction is defined componentwise. Now, a *Petri net* is a finite set $N$ of *transitions* being pairs of *input* and *output* configurations. The semantics of a Petri net is the following binary relation $\rightarrow_N$ over configurations: $C \rightarrow_N C'$ if there is a transition $t = (I, O) \in N$ such that $I \leq C$ and $C' = C - I + O$.

To turn a regular DAG grammar into a Petri net, just convert every rule $\alpha \xrightarrow{\sigma} \beta$ into the transition $(\overline{\alpha}, \overline{\beta})$, where $\overline{\gamma}$ ($\gamma \in \{\alpha, \beta\}$) is obtained from $\gamma$ by forgetting about the order of its symbol occurrences. Clearly, in this Petri net $N$ we have $\mathbf{0} \rightarrow_N^+ \mathbf{0}$, a *zero cycle*, if and only if $G$ generates at least one DAG.[4] It was shown in [9] that the existence of zero cycles is decidable in polynomial time, and hence the emptiness problem for DAG automata and regular DAG grammars is [7].

Another consequence of this construction, detailed in [4], is that useless rules can efficiently be removed from regular DAG grammars, as a rule is useful (in the sense that it appears in at least one derivation of a DAG in $L(G)$) if it occurs in some zero cycle. Once useless rules have been removed, it is furthermore possible to decide in polynomial time whether $L(G)$ is finite by searching the set of rules for an (undirected) cycle of matching states. Let us illustrate the argument, which is from [3, 4].

An example of a cycle involving three rules is illustrated in Figure 3. Each rule $r_i$ ($i \in [3]$) is involved in the cycle with two distinct edges $e_i, e_i'$ in such a way that $e_i'$ and $e_{i \bmod 3+1}$ (connected by dashed arrows) point into the same direction and carry the same state. Now, choose for each of the three rules a DAG in $L(G)$ which can be generated using the rule, with their associated runs (now considering $G$ as a DAG automaton). Since there are no useless rules this is possible. Then their disjoint union is an accepted DAG consisting of three connected components. For the sake of illustration, we assume here that this DAG has only twelve nodes, i.e., all nodes except those involved in the rules of the cycle are roots and leaves.

[4]The DAG corresponding to such a transition sequence can be disconnected, but we already know that this means that each of its connected components is in $L(G)$.

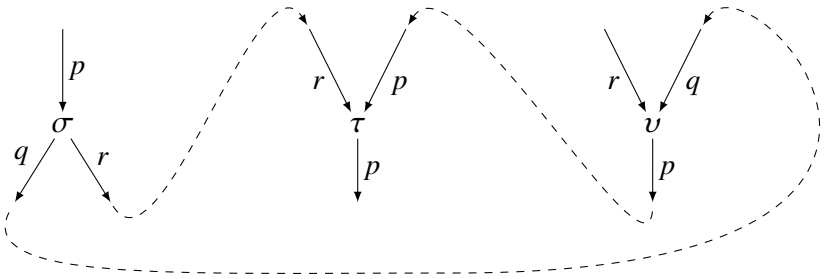



Figure 3: A rule cycle involving three rules

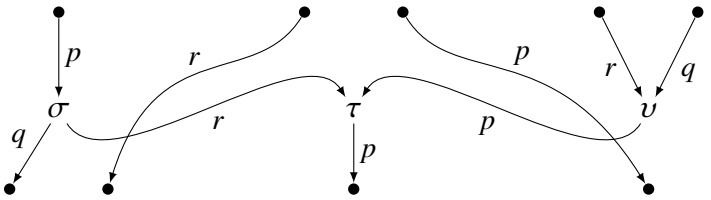



Figure 4: Connecting the three rules yields a component with an undirected path between $e_1$ and $e'_3$ (i.e., between the two edges labelled with the state $q$)

Suppose that we swap the targets $e'_i$ and $e_{i+1}$ for $i \in [2]$ (note that we do not close the cycle). This operation does not affect the validity of the run, and it results in a DAG containing an undirected path from $e_1$ to $e'_3$, as shown in Figure 4. It follows that, if there is a cycle involving $k \geq 1$ rules $r_1, \ldots, r_k$, then $L(G)^{\uplus}$ contains a DAG $D$ having an undirected simple path of length $k+1$. Thus $D$ contains a connected component which is in $L(G)$ and has at least $k+1$ edges. Further, if $r_1 \cdots r_k$ forms a rule cycle, then $r_1 \cdots r_k r_1 \cdots r_k$ does so, too, and hence $L(G)$ is infinite. Altogether, this yields the following theorem combining results of [7] and [3, 4].

**Theorem 3.3.** The emptiness problem and the finiteness problem for DAG automata and regular DAG grammars are decidable in polynomial time.

# 4 Multiple versus Single Roots

A seemingly innocent detail regarding the definition of DAG automata and regular DAG grammars is whether they provide a mechanism to control the set of roots. Those in Definition 2.2 do not. Taking into account the earlier remarks regarding unordered and unranked DAG automata, this seems to be the most essential difference between them and those studied by Priese [17]. In the latter, runs provide each root with an incoming state and each leaf with an outgoing one. The collection of states on roots (and leaves) can furthermore be checked for regular properties. To study whether this makes a difference, rather than adding such a

mechanism, it essentially suffices to consider the setting in which DAGs are required to possess a unique root, i.e., to study $L_{\mathrm{u}} = \{D \in L \mid D \text{ has a unique root}\}$ where $L$ is a regular DAG language. The fact that this provides runs (or derivations in regular DAG grammars) with a unique starting point has consequences regarding their expressive power that may be unexpected at first.

Consider, for simplicity, the case where a regular DAG grammar $G$ has a unique rule with the head $\lambda$, say $r = (\lambda \xrightarrow{\sigma} \beta)$, and construct the Petri net $N$ from $G$ as in the previous section, but excluding the transition $(\mathbf{0}, \overline{\beta})$. Since every derivation of a DAG in $L(G)_{\mathrm{u}}$ starts with $r$ and must not use it ever again, the configuration sequences corresponding to derivations of DAGs in $L(G)_{\mathrm{u}}$ are those which start with the configuration $\overline{\beta}$ and result in $\mathbf{0}$. If we want to decide emptiness of $L(G)_{\mathrm{u}}$, we thus have to ask whether $\overline{\beta} \to_N^+ \mathbf{0}$. This is a special case of *Petri net reachability*, which is decidable [11, 15], but without a known primitive recursive bound on its complexity. One may thus wonder how special the special case is.

For this, consider any Petri net $N$ and two distinct configurations $C, C'$. We want to construct a regular DAG grammar $G$ such that $L(G)_{\mathrm{u}} \neq \emptyset$ iff $C \to_N^+ C'$. Enrich $\Xi$ by a fresh symbol $\xi_0$, which is added once to the input and output of each transition in $N$ as well as to $C$ and $C'$. This makes sure that there are no transitions with empty input or output. Now, we choose a dummy symbol $\sigma$ and invert the construction of Petri nets from regular DAG grammars, turning every transition of $N$ into a transition $\alpha \xrightarrow{\sigma} \beta$ (where the order of symbols in $\alpha$ and $\beta$ does not matter). Note that $\alpha, \beta \neq \lambda$. Finally, we add rules $\lambda \xrightarrow{\sigma} \gamma$ and $\gamma' \xrightarrow{\sigma} \lambda$ where $\gamma$ and $\gamma'$ are obtained by arbitrarily ordering $C$ and $C'$, respectively. In a derivation of a DAG in $L(G)_{\mathrm{u}}$, the rule $\lambda \xrightarrow{\sigma} \gamma$ is applied exactly once, and every rule except $\gamma' \xrightarrow{\sigma} \lambda$ preserves the invariant that there is exactly one occurrence of $\xi_0$. Hence the rule $\gamma' \xrightarrow{\sigma} \lambda$ can only be applied once, too, thus terminating the derivation. It follows that $L(G)_{\mathrm{u}}$ is empty if and only if $C \to_N^+ C'$, which proves the following theorem:

**Theorem 4.1.** Deciding the emptiness of $L(G)_{\mathrm{u}}$ for regular DAG grammars $G$ is logspace equivalent to the reachability problem for Petri nets.

It seems to be unclear whether a similar result can be shown regarding the finiteness problem. Clearly, the finiteness problem is not easier than the emptiness problem: to reduce the latter to the former, simply modify the input grammar in such a way that it appends, below every leaf of a generated DAG, an arbitrarily long chain of edges.

**Open Problem 1.** Prove or disprove that the finiteness problem for $L(G)_{\mathrm{u}}$, given a regular DAG grammar $G$ as input, is decidable.

Another significant difference between regular DAG languages in the sense of Definition 2.2 and rooted ones concerns their path languages. For a DAG $D \in \mathcal{D}_\Sigma$,

let $dpaths(D)$ be the set of all *directed* paths $v_0 e_1 \cdots e_n v_n$ from a root $v_0$ to a leaf $v_n$. The *path language* of $D$ is $\pi(D) = \{lab_D(v_0 \cdots v_n) \mid v_0 e_1 \cdots e_n v_n \in dpaths(D)\}$, and the path language of $L \subseteq \mathcal{D}_\Sigma$ is $\pi(L) = \bigcup_{D \in L} dpaths(D)$.

We argue that $\pi(L)$ is regular for all regular DAG languages $L$. For this, consider a DAG automaton $A = (Q, \Sigma, R)$ such that $R$ does not contain any useless rule (where a rule is useless if it cannot be used in any run). Build the NFA $A' = (Q', \Sigma, \delta, q_0, \{q_f\})$, where $Q' = Q \cup \{q_0, q_f\}$ (for some $q_0, q_f \notin Q$) in the most straightforward imaginable way, nondeterministically travelling down a DAG without looking left or right: $\delta$ is defined to be the smallest relation such that, for all rules $(\alpha \overset{\sigma}{\longleftrightarrow} \beta) \in R$,

- if $\alpha = \lambda$ then $[\beta] \subseteq \delta(q_0, \sigma)$,
- $[\beta] \subseteq \delta(q, \sigma)$ for all $q \in [\alpha]$, and
- if $\beta = \lambda$ then $q_f \in \delta(q, \sigma)$ for all rules $q \in [\alpha]$.

We claim that $A'$ accepts $\pi(L(A))$. It should be clear that $\pi(L(A)) \subseteq L(A')$, but the converse is not equally obvious. Consider a string $u = \sigma_1 \cdots \sigma_{n+1} \in L(A')$. Then there are states $q_1, \ldots, q_n \in Q$ such that $q_i \in \delta(q_{i-1}, \sigma_i)$ for all $i \in [n]$, and $q_f \in \delta(q_n, \sigma_{i+1})$. Let us first consider the proper prefixes $\sigma_1 \cdots \sigma_i$, $i \in [n]$. We show by induction on $i$ that there exist a run $\rho$ on a DAG $D \in \mathcal{D}_\Sigma$ and a path $v_1 e_1 \ldots e_{m-1} v_m \in dpaths(D)$ with $m > i$ such that $lab_D(v_1 \cdots v_i) = \sigma_1 \cdots \sigma_i$ and $\rho(e_i) = q_i$. Suppose that the statement holds for $i < n$. The situation is illustrated in Figure 5 on the left. Now, since $q_{i+1} \in \delta(q_i, \sigma_{i+1})$, we know that there is a rule of the form $\cdots q_i \cdots \overset{\sigma_{i+1}}{\longleftrightarrow} \cdots q_{i+1} \cdots$ in $R$, and since this rule is not useless, there is another DAG $D' \in L(A)$ admitting a run as indicated in the right part of Figure 5. Swapping the targets of $e_i$ and the corresponding edge in $D'$ yields a run on the (now possibly disconnected) DAG illustrated in Figure 6, because both edges carry the same state. Clearly, the connected component containing $e_i$ has the form claimed, and by definition it is in $L(A)$. We have thus shown that there exists a run $\rho$ on a DAG $D \in \mathcal{D}_\Sigma$ and a path $v_1 e_1 \ldots e_{m-1} v_m \in dpaths(D)$ with $m > n$ such that $lab_D(v_1 \cdots v_n) = \sigma_1 \cdots \sigma_n$ and $\rho(e_n) = q_n$. Now recall that $q_f \in \delta(q_n, \sigma_{n+1})$. By the construction of $\delta$, this means that $R$ contains a rule $\cdots q_n \cdots \overset{\sigma_{n+1}}{\longleftrightarrow} \lambda$. Again, there is a DAG $D' \in L(A)$ with a run using this rule. This time, however, the corresponding node labelled with $\sigma_{n+1}$ in $D'$ is a leaf. Thus, applying the very same swapping trick once again to $D$ and $D'$ yields a DAG $D'' \in L(A)$ such that $\sigma_1 \cdots \sigma_{n+1} \in \pi(D)$.

Since every regular string language "is" also a regular DAG language, we get the following theorem.

**Theorem 4.2 ([7, 3]).** The class of all string languages $\pi(L)$ such that $L$ is a regular DAG language, is equal to the class of all regular string languages.

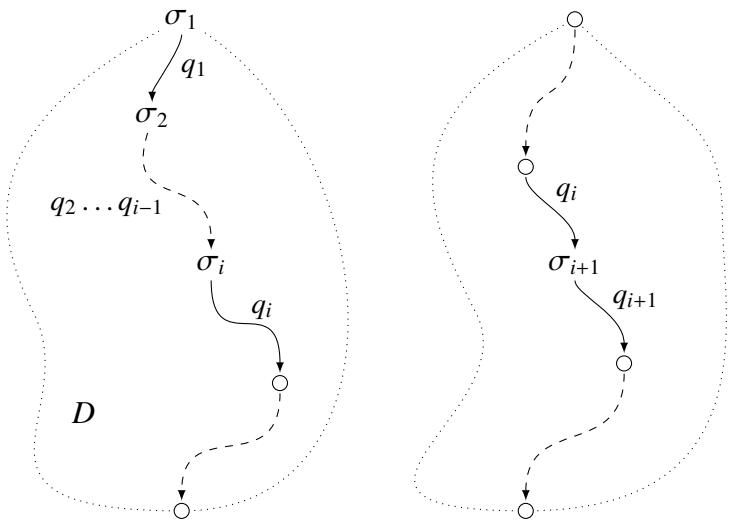



Figure 5: DAG $D$ of the induction hypothesis in path construction (left) and another DAG in $L(A)$ with a run using rule $\cdots q_i \cdots \overset{\sigma_{i+1}}{\longleftrightarrow} \cdots q_{i+1} \cdots$ (right); circles represent nodes with irrelevant labels

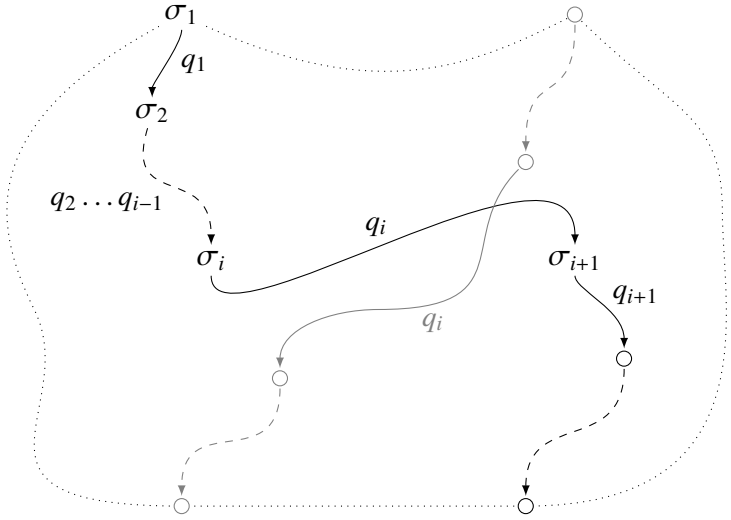



Figure 6: Swapping edges in the DAGs of Figure 5

The theorem was first shown in [7], though with a non-constructive proof. In [4] it is obtained as a corollary of another result: the "unfolding" of a regular DAG language is a regular tree language. Since unfolding preserves path languages, the theorem follows. Moreover, also shown in [4], useless rules can be detected in polynomial time, which makes the theorem effective.

Now, what about the case of rooted DAGs? As observed in [17], $\pi(L_{\mathrm{u}})$ is not

even necessarily context-free. Consider, for example, the regular DAG grammar $G$ with the following rules:

$$\begin{array}{rrrrr} \lambda \xrightarrow{a} \nu_a\xi_a & & \nu_a \xrightarrow{a} \nu_a\xi_a & \xi_a \xrightarrow{\diamond} \xi'_a & \\ \nu_a\xi'_a \xrightarrow{b} \nu_b\xi_b & & \nu_b\xi'_a \xrightarrow{b} \nu_b\xi_b & \xi_b \xrightarrow{\diamond} \xi'_b & \\ \nu_b\xi'_b \xrightarrow{c} \nu_c\xi_c & & \nu_c\xi'_b \xrightarrow{c} \nu_c\xi_c & \xi_c \xrightarrow{\diamond} \xi'_c & \nu_c\xi'_b \xrightarrow{c} \lambda\,. \end{array}$$

In a generated DAG with only one root, reading along its left spine yields strings of the form $a^k b^l c^m$. However, the generation of each $a$ creates a $\xi_a$ which can only give rise to a $\diamond$ and a $\xi'_a$. The latter can vanish only if a $b$ is generated. This ensures that $k = l$. Similarly, $l = m$. Hence, $\pi(L(G)_{\mathrm{u}}) \cap a^*b^*c^* = \{a^n b^n c^n \mid n > 0\}$, a non-context-free language. Thus, $\pi(L(G)_{\mathrm{u}})$ is not context-free either. It seems to be unknown exactly which string languages are of the form $\pi(L(G)_{\mathrm{u}})$:

**Open Problem 2.** Characterize the class of string languages of the form $\pi(L_{\mathrm{u}})$, for regular DAG languages $L$.

It does not seem to be very unlikely that there actually is no "nice" characterization of this class, in which case one may instead consider the following problem:

**Open Problem 3.** Characterize the class of string languages $\pi(L_{\mathrm{u}}) \cap R$, where $L$ is a regular DAG language and $R$ is a regular string language.

A related question is whether the Parikh images $\psi(L)$ of regular DAG languages $L$ are semilinear.[5] We conjecture that this is indeed the case, but the results above seem to indicate that it may not be an easy result to prove. This is because, if $\psi(L)$ is semilinear for all regular DAG languages $L$, then so is $\psi(L_{\mathrm{u}})$. To see this, let $\{\sigma_1, \ldots, \sigma_k\}$ be the alphabet of node labels. Modify $L$ by attaching an artificial root, labelled with a fresh symbol $\sigma_0$, above each original root. The DAG language $L'$ obtained in this way is still regular, and $\psi(L_{\mathrm{u}}) = \{(n_1, \ldots, n_k) \in \psi(L') \mid (1, n_1, \ldots, n_k) \in \psi(L')\}$, which is semilinear if $\psi(L')$ is.

**Open Problem 4.** Prove or disprove the following conjecture: *For every regular DAG language $L$, $\psi(L)$ is semilinear (and hence so is $\psi(L_{\mathrm{u}})$).*

# 5 Transforming Trees and DAGs

DAG-to-tree transductions, transductions that turn DAGs into trees, were proposed long ago by Kamimura and Slutzki [14], though in a setting quite different from the one discussed in this paper. A variant of this type of transduction has

[5] If $L$ is a DAG language over $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$, let $\psi(D) = (n_1, \ldots, n_k)$ for a DAG $D \in L$ if $D$ contains $n_i$ nodes labelled with $\sigma_i$, for all $i \in [k]$. Then $\psi(L) = \{\psi(D) \mid D \in L\}$ is the Parikh image of $L$. See [16] for Parikh's original paper on Parikh images of regular string languages.

been implemented in DAGGER [19]. However, the opposite direction, tree-to-DAG transductions, does not seem to have been pursued in the literature at all. This is surprising because such transductions could be particularly useful in natural language processing where they could be used to turn a syntactic parse of a sentence into a semantic representation. In general, one may look for ways to turn tree languages into DAG languages, either by creating a DAG automaton from a tree automaton, or by using a tree-to-DAG transduction. In this section, some preliminary ideas regarding this topic are presented.

As mentioned above, it is shown in [4] that the unfolding of a regular DAG language yields a regular tree language. Unfolding is defined in a straightforward way, as follows. Given a DAG $D$ and a root $r$ of $D$, define $tree_r(D) = (V, E, lab, in, out)$ where

- $V$ contains a node $v_p$, called a *residual of* $v$, for every $v \in V_D$ and every directed path $p$ from $r$ to $v$,
- $E$ contains an edge $e_p$ for every directed path $p$ from $r$ to some $v \in V_D \setminus \{r\}$,

and, for all $v_p \in V$,

- $lab(v_p) = lab_D(v)$,
- $in(v_p)$ is equal to $e_p$ if $p$ is nonempty and $\lambda$ otherwise,
- $out(v_p) = e_{p.e_1} \cdots e_{p.e_n}$, where $out_G(v) = e_1 \cdots e_n$.

Here, for a path $p$ from $u$ to $v$ and an edge $e$ from $v$ to $v'$, $p.e$ denotes the path obtained by appending $e\,v'$ to $p$. The unfolding of a DAG language $L$ is $tree(L) = \{tree_r(D) \mid D \in L,\ r \text{ a root of } D\}$. From a DAG automaton $A$ without useless rules one can easily obtain a DAG automaton $A'$ with $L(A') = tree(L(A))$ by keeping all rules whose head is $\lambda$ and replacing every other rule $p_1 \cdots p_k \overset{\sigma}{\longleftarrow} q_1 \cdots q_\ell$ by all rules $p_i \overset{\sigma}{\longleftarrow} q_1 \cdots q_\ell$, $i \in [k]$.

However, how about the converse? Of course, every regular tree language *is* a regular DAG language, but are there interesting nontrivial ways to turn a regular tree language into a regular DAG language? This seems to be another open problem.

**Open Problem 5.** Find interesting ways to construct a regular DAG language $L'$ from a regular tree language $L$, for example in such a way that $tree(L') = L$.

In order to illustrate one possible direction such results may take, let us have a look at an idea briefly mentioned in the concluding section of [4], working it out in slightly greater detail. It provides a way to construct $L'$ such that $tree(L') = L$, starting from a regular tree grammar $G = (\Xi, \Sigma, R)$ that accepts $L$ (i.e., $G$ is a regular DAG grammar such that all heads of rules are either empty or consist of exactly one state). To keep the presentation simple, assume that the trees in $L$ are

binary ones, i.e., the tail of each rule is either empty or of length two. The DAGs in $L'$ will in addition have nodes with two incoming edges, intuitively corresponding to two subtrees that have been folded into one sub-DAG. Imagine that, after a few steps of a derivation in $G$, there are two nonterminal nodes labelled $\xi$ and $\xi'$. If we want to combine them into a single node with two incoming edges, assuming for the moment that the sub-DAG generated from this nonterminal is just a tree, it must be a tree belonging to the intersection of the tree languages generated by $\xi$ and $\xi'$. If the same thing happens recursively at several levels, intersections signified by any subset of $\Xi$ are needed. Therefore, we shall incorporate such intersections into the regular tree grammar before turning it into a regular DAG grammar. For this, let $G' = (2^\Xi, \Sigma, R')$ use the powerset of $\Xi$ as its set of nonterminals, the intention being that each $\Xi' \subseteq \Xi$ generates the intersection of the tree languages $G$ generates from the individual nonterminals $\xi \in \Xi'$. Intuitively, every $\xi \in \Xi'$ can be viewed as a constraint. The rules in $R'$ are given as follows for every $\sigma \in \Sigma$ and $\Xi_0, \Xi_1, \Xi_2 \subseteq \Xi$:

(a) If there are $\xi_1 \in \Xi_1, \xi_2 \in \Xi_2$ such that $R$ contains the rule $\lambda \xrightarrow{\sigma} \xi_1\xi_2$ then $\lambda \xrightarrow{\sigma} \Xi_1\Xi_2$ is in $R'$. (When generating a root, we may allow for any additional constraints to be added nondeterministically.)
(b) If, for every $\xi_0 \in \Xi_0$, there are $\xi_1 \in \Xi_1, \xi_2 \in \Xi_2$ such that $R$ contains the rule $\xi_0 \xrightarrow{\sigma} \xi_1\xi_2$ then $\Xi_0 \xrightarrow{\sigma} \Xi_1\Xi_2$ is in $R'$.
(c) If, for every $\xi_0 \in \Xi_0$, the rule $\xi_0 \xrightarrow{\sigma} \lambda$ is in $R$, then $\Xi_0 \xrightarrow{\sigma} \lambda$ is in $R'$.

It should be clear that, in $G'$, every nonterminal $\Xi'$ indeed generates the intersection of the sets of trees generated by the individual nonterminals $\xi \in \Xi'$ in $G$. In particular, $L(G') = L(G)$. Thus, a tree that can be generated from both $\Xi_1$ and $\Xi_2$ can also be generated from $\Xi_1 \cup \Xi_2$. Now we can turn $G'$ into a regular DAG grammar $G'' = (2^\Xi, \Sigma, R'')$ as follows: for $\Xi_1, \Xi_2, \Xi'_1, \Xi'_2 \subseteq \Xi$

(a′) if $\lambda \xrightarrow{\sigma} \Xi'_1\Xi'_2$ is in $R'$ then it is in $R''$ as well;
(b′) if both $\Xi_1 \xrightarrow{\sigma} \Xi'_1\Xi'_2$ and $\Xi_2 \xrightarrow{\sigma} \Xi'_1\Xi'_2$ are in $R'$, then $\Xi_1\Xi_2 \xrightarrow{\sigma} \Xi'_1\Xi'_2$ is in $R''$;
(c′) if both $\Xi_1 \xrightarrow{\sigma} \lambda$ and $\Xi_2 \xrightarrow{\sigma} \lambda$ are in $R'$, then $\Xi_1\Xi_2 \xrightarrow{\sigma} \lambda$ is in $R''$.

**Proposition 5.1.** The above construction yields $tree(L(G'')) = L(G)$.

*Proof sketch.* Since $L(G) = L(G')$ we only need to prove $tree(L(G'')) = L(G')$.

'$\subseteq$' We show the following by induction on the length of derivations: if $\emptyset \to^+_{R''} D$ and $T = tree_r(D)$ for a root $r$ of $D$, then $\emptyset \to^+_{R'} T$. For derivations of length 1 this is trivial by (a′). Now, assume that $\emptyset \to^+_{R''} D_0 \to_{R''} D$ and $T \in tree_r(D)$, and let $v_1, v_2 \in V_{D_0}$ be the two nonterminal nodes in $D_0$ which the rule in the last step, say $\Xi_1\Xi_2 \xrightarrow{\sigma} \Xi'_1\Xi'_2$, is applied to. Let $T_0 = tree_r(D_0)$. Then $T$ is obtained from $T_0$

by replacing all residuals of $v_1$ and $v_2$ by nodes labelled $\sigma$, which children labelled $\Xi'_1$ and $\Xi'_2$. By (b′) the rules $\Xi_i \xrightarrow{\sigma} \Xi'_1\Xi'_2$ are in $R'$ for $i = 1, 2$ and thus, together with the induction hypothesis, $\emptyset \to^+_{R'} T_0 \to^+_{R'} T$. The case where the rule applied to $v_1$ and $v_2$ is a terminating rule $\Xi_1\Xi_2 \xrightarrow{\sigma} \lambda$ is similar.

'$\supseteq$' Consider $A = (2^{\Xi}, \Sigma, R' \cup R'')$, which we may view as a DAG automaton rather than a regular DAG grammar. Let $D_0 \in L(G')$ and consider a run $\rho_0$ of $A$ on $D_0$. Since $D_0$ is a tree, $\rho_0$ uses only rules in $R'$. If $d$ is the largest distance of any node of $D_0$ from the root, let the *level* of a node $v \in V_{D_0}$ be $d - d'$, where $d'$ is the distance of $v$ itself from the root. In other words, the leaves deepest down in the tree are at level 0, the nodes one step nearer toward the root are at level 1, and so on. Now, for $\ell = 1, \ldots, d$, construct $D_\ell$ from $D_{\ell-1}$ by

- copying all nodes at levels $\geq \ell$ together with their incident edges (i.e., the incoming and outgoing edges of the copy of a node $v$ are the copies of the incoming and outgoing edges of $v$) and
- defining $in_{D_\ell}(v) = ee'$ for every node $v$ at level $\ell - 1$ with $in_{D_{\ell-1}}(v) = e$, where $e'$ is the copy of $e$.

Moreover, $\rho_{\ell-1}$ is extended to a run $\rho_\ell$ on $D_\ell$ by assigning each copied edge the same state as the original edge. By (b′) and (c′) $\rho_\ell$ is indeed a run if $\rho_{\ell-1}$ is. Moreover, by construction we have $tree_r(D_\ell) = tree_r(D_{\ell-1})$.

Thus, we eventually obtain a run $\rho_d$ of $A$ on $D_d$ with $tree_r(D_d) = tree_r(D_0)$. However, $D_d$ does not contain any node with a single incoming edge, which means that $\rho_d$ is actually a run of (the DAG automaton corresponding to) $G''$, showing that $L(G') \subseteq tree(L(G''))$. □

Another open problem is to come up with natural notions of transducers.

**Open Problem 6.** Devise useful notions of tree-to-DAG and DAG-to-DAG transducers with good properties.

In the following, a class of DAG transducers is proposed that may turn out to be of interest in this regard.

Let us say that a $\Sigma$-*DAG section* $S$ is defined in the same way as a $\Sigma$-DAG (see Section 2) except that the labelling $lab_S$ is defined only on $V_S \setminus (top(S) \cup bot(S))$, where $top(S) \subseteq V_S$ contains only nodes $v$ with $[in_S(v)] = \emptyset \neq [out_S(v)]$ and $bot(S) \subseteq V_S$ contains only nodes $v$ with $[in_S(v)] \neq \emptyset = [out_S(v)]$. In other words, a node may be in $top(S)$ if it is a root but not a leaf, and it may be in $bot(S)$ if it is a leaf but not a root. Thus $top(S)$ and $bot(S)$ are disjoint sets that are uniquely determined by the domain $V_S \setminus (top(S) \cup bot(S))$ of $lab_S$. In the following, we denote $V_S \setminus (top(S) \cup bot(S))$ by $inner(S)$. Moreover, $E_S^\top = \{e \in E_S \mid src_S(e) \in top(S)\}$ and $E_S^\bot = \{e \in E_S \mid tar_S(e) \in bot(S)\}$ are the *top* and *bottom edges* of

$S$, respectively. Note that, as DAGs are DAG sections with $\mathit{inner}(S) = V_S$, all definitions regarding DAG sections apply to DAGs as well.

For a DAG section $S$ and a DAG $D$, we call a mapping $m\colon (V_S \cup E_S) \to (V_D \cup E_D)$ that injectively maps nodes to nodes and edges to edges a *bottom-up morphism*[6] and denote it as $m\colon S \to D$ if

(a) $\mathit{lab}_D(m(u)) = \mathit{lab}_S(u)$ for all $u \in \mathit{inner}(S)$ and
(b) for all $v \in V_S$ the canonical extension of $m$ to sequences satisfies

$$\begin{cases} m(\mathit{out}_S(v)) \text{ is a subsequence}^{7} \text{ of } \mathit{out}_D(m(v)) & \text{if } v \in \mathit{top}(S) \\ m(\mathit{out}_S(v)) = \mathit{out}_D(m(v)) & \text{if } v \in \mathit{inner}(S) \\ m(\mathit{in}_S(v)) = \mathit{in}_D(m(v)) & \text{if } v \in V_S \setminus \mathit{top}(S). \end{cases}$$

With these conditions, $m|_{V_S}$ uniquely determines $m|_{E_S}$, i.e., $m$ only needs to be specified on nodes. Note the asymmetry of the definition. For nodes $v \in \mathit{bot}(S)$ it requires that the entire sequence of incoming edges of $m(v)$ in $D$ is "matched" by the sequence of incoming edges of $v$ in $S$. For nodes $v \in \mathit{inner}(S)$ this is required to hold for both incoming and outgoing edges. However, for $v \in \mathit{top}(S)$ only some of outgoing edges of $m(v)$ actually need to be matched by edges in $S$. We denote the image of $S$ under $m$, which is a sub-DAG of $D$, by $m(S)$.

Given a set $Q$ of states understood from the context, an *$E$-labelled DAG section* $S\langle q\rangle$ consists of a DAG section $S$ with $E \subseteq E_S$ and a mapping $q\colon E \to Q$ that assigns states to the edges in $E$. A $\emptyset$-labelled DAG section $S\langle q\rangle$ is identified with $S$ itself.

**Definition 5.1 (bottom-up DAG transducer).** A *bottom-up DAG transducer* is a system $\tau = (Q, \Sigma, \Delta, R)$ consisting of a finite set $Q$ of states, alphabets $\Sigma$ and $\Delta$ of *input* and *output labels*, and a finite set $R$ of *bottom-up rules*. Each bottom-up rule has the form $S\langle b\rangle \to S'\langle t\rangle$ where $S\langle b\rangle$ is a $\mathit{bot}(S)$-labelled $\Sigma$-DAG section, $S'\langle t\rangle$ is a $\mathit{top}(S')$-labelled $\Delta$-DAG section, $\mathit{top}(S) = \mathit{top}(S')$, $E_S^\top = E_{S'}^\top$, and $\mathit{bot}(S) = \mathit{bot}(S')$.

Consider an $E$-labelled $(\Sigma \cup \Delta)$-DAG $D\langle q\rangle$, a bottom-up rule $S\langle b\rangle \to S'\langle t\rangle$ in $R$, and a bottom-up morphism $m\colon S \to D$ such that $q(m(e)) = b(e)$ for all $e \in \mathit{bot}(S)$.[8] Then $D\langle q\rangle \to_R D'\langle q'\rangle$ where

- $D'$ is obtained from $D$ by replacing $m(S)$ by $m(S')$, and
- $q' = q|_{E \setminus m(E_S^+)} \cup \{e \mapsto t(e) \mid e \in E_{S'}^\top\}$.

The *bottom-up DAG transduction* computed by $\tau$ is the set of all pairs $(D, D')$ such that $D$ is a $\Sigma$-DAG, $D'$ is a $\Delta$-DAG, and $D \to_R^+ D'$.

[6] A *top-down morphism*, which we do not define explicitly here, can be defined in the dual way.
[7] A *subsequence* of $e_1 \cdots e_k$ is any sequence $e_{i_1} \cdots e_{i_\ell}$ such that $1 \le i_1 < \cdots < i_\ell \le k$.
[8] In particular, $m(e) \in E$ for all $e \in \mathit{bot}(S)$.

A dual notion of top-down DAG transducers can easily be obtained by reversing rules and using the dual notion of top-down morphisms.

Note that we do not restrict DAG transductions to $\mathcal{D}_\Sigma \times \mathcal{D}_\Delta$ (i.e., to connected DAGs), because the transduction can turn connected DAGs into disconnected ones and vice versa. It may, however, also be meaningful to study alternatives, such as the "forgetful" transduction consisting of all $(D, D') \in \mathcal{D}_\Sigma \times \mathcal{D}_\Delta$ such that $D \to_R^+ D' \uplus D''$ for a $\Delta$-DAG $D''$.

It follows by a straightforward induction on the length of derivations that the definition above is consistent in the sense that, after any number of steps, the structure $D\langle q \rangle$ derived from a $\Sigma$-DAG $D_0$ is indeed an $E$-labelled $(\Sigma \cup \Delta)$-DAG. More precisely, $E$ forms a cut set cutting $D$ into a $\Delta$-DAG $D^\perp$ and a $\Sigma$-DAG $D^\top$ such that, for every $e \in E$, $src_D(e)$ belongs to $D^\top$ and $tar_D(e)$ is a root of $D^\perp$. Further, let us say that a derivation step as in the definition *processes* the nodes in $m(inner(S))$. Then $D^\top$ is the sub-DAG of $D_0$ consisting of all still unprocessed nodes and their incident edges save those in $E$.

By a standard search procedure, it is decidable for DAGs $D, D'$ whether $D \to_R^+ D'$, but the complexity of the problem remains to be studied.

Clearly, tree-to-DAG transductions and DAG-to-tree transductions[9] are obtained as special cases of DAG transductions. Let us consider an example of a bottom-up tree-to-DAG transduction that involves some folding.

**Example 5.2 (bottom-up tree-to-DAG transduction).** We consider input trees whose leaves are labelled by $b$ and $g$, representing 'boys' and 'girls'. Binary nodes are labelled by $c$ and represent groups of 'children'. In addition, there can be leaves labelled by $r$. Such a leaf is a 'reference' that can refer to any group provided that all members of the group share the same gender. The transduction shall allow to fold pairs of equal subtrees into one, turn $c$ into $u$ at the roots of subtrees representing groups of one unique gender, and redirect any references to appropriate groups. We use states `b`, `g`, and `m` to remember whether the subtree consists of boys or girls only, or of children of mixed genders. Some typical rules of this transducer are shown in Figure 7, but several more would be needed to cover all major cases. Figure 8 shows an example of a computation.

# 6 Conclusion

Various ideas and properties of DAG languages have been discussed, most of them known from the literature and especially from [17, 7, 3, 3]. However, equally many interesting results had to be left out. In particular, the complexity of the

[9] Perhaps these should rather be called forest-to-DAG transductions and DAG-to-forest transductions because they actually process (finite) forests.

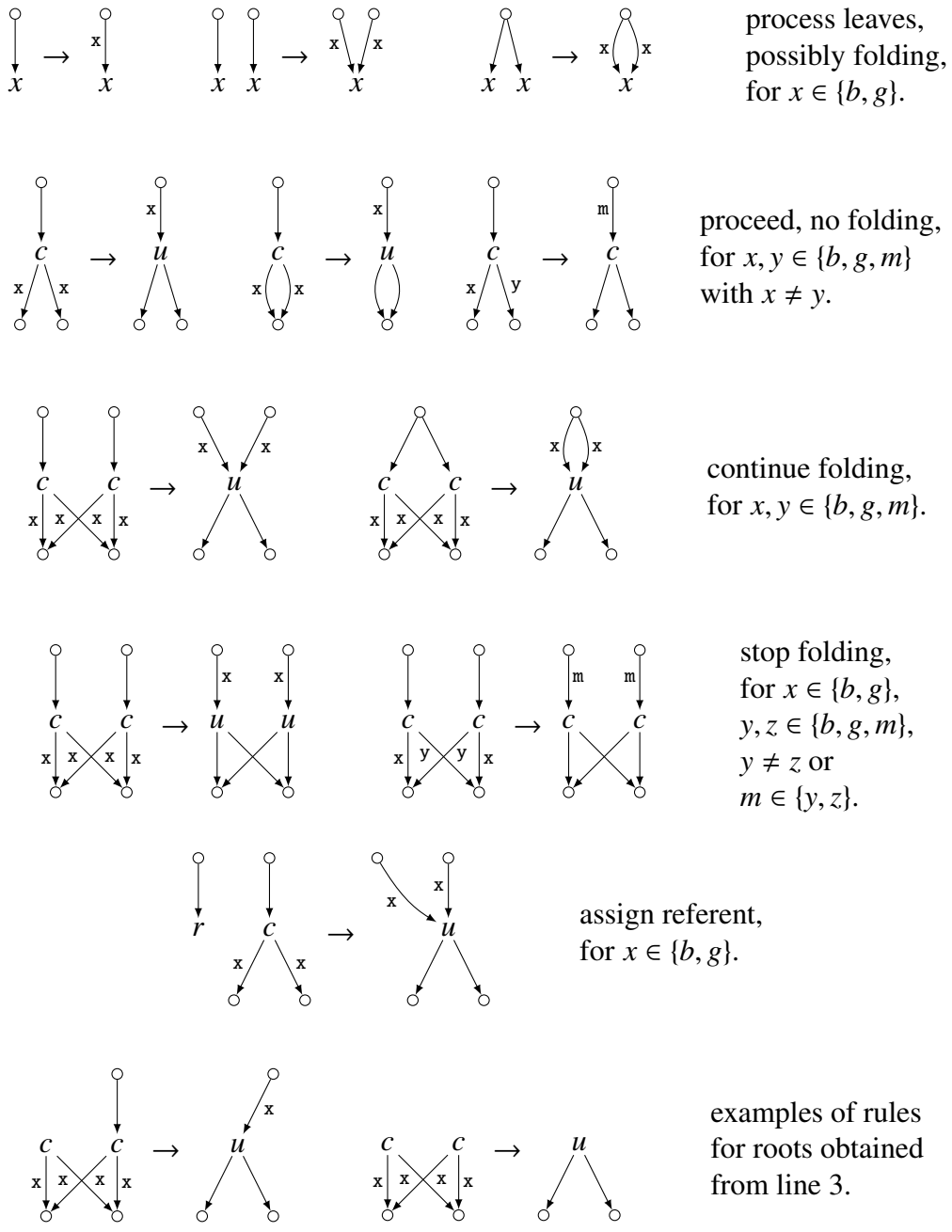



Figure 7: Rules of a bottom-up tree-to-DAG transducer. Various rules, especially many of those applying to roots, are omitted. The last row, obtained from the leftmost rule in the third row by deleting one or both of the nodes in $top(S) = top(S')$, shows an example of how they can be obtained.

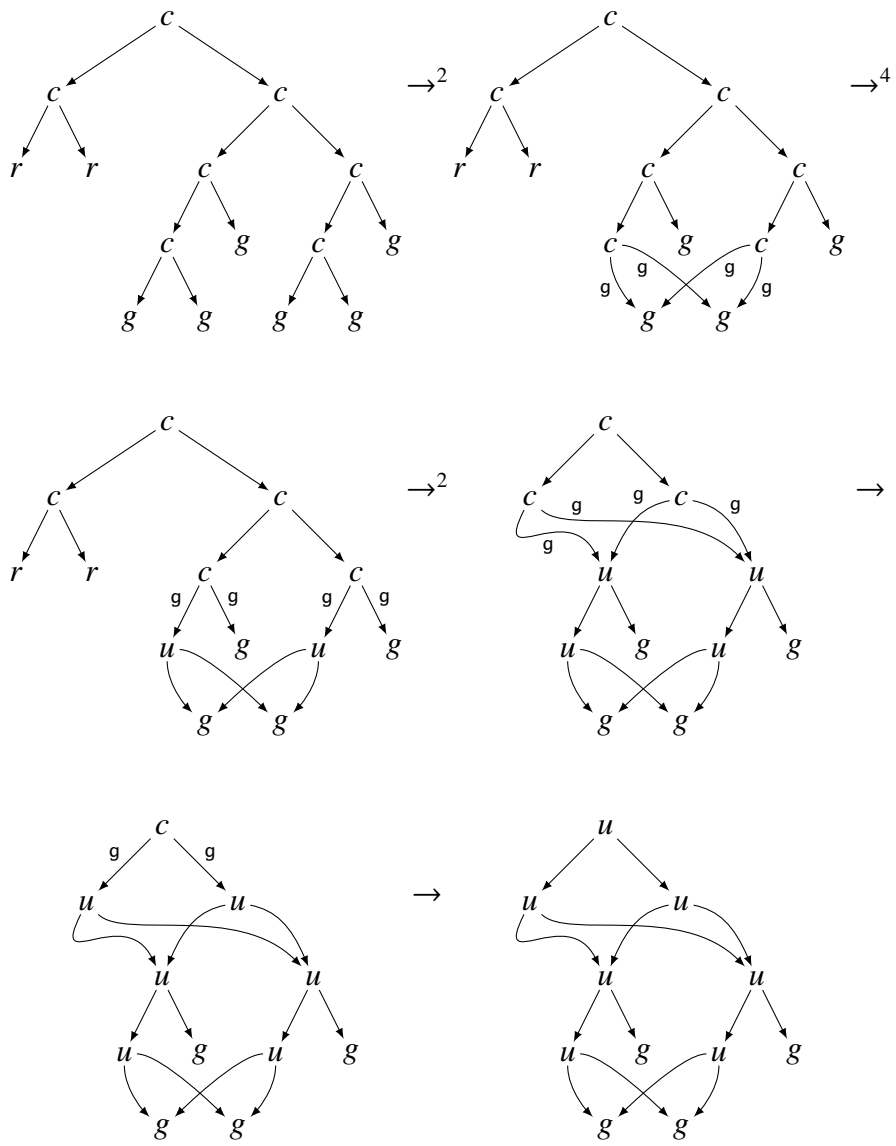



Figure 8: A computation of the tree-to-DAG transducer in Example 5.2

membership problem for regular DAG languages was not discussed. It is shown in [7] that even the non-uniform membership problem is, in fact, NP-complete: there are fixed NP-complete regular DAG languages. The problem becomes easier but not necessarily practical if restricted to DAGs of bounded treewidth and, of course, much easier for deterministic DAG automata. Finding further restrictions that result in a manageable membership problem remains another open problem. Weighted DAG automata, as defined in [7], are another interesting topic that should be explored further.

Moreover, as indicated by the discussion in the previous section, establishing further relations between regular tree languages and regular DAG languages, e.g. by folding and unfolding, and studying DAG transducers of various kinds seem to be potentially rewarding tasks not only from the theoretical perspective, but also in view of their possible applications in natural language processing and other areas. The reader is invited to contribute!

## Acknowledgment

I thank Johannes Blum, David Chiang, Daniel Gildea, Adam Lopez, and Giorgio Satta for their contributions to the work discussed here, and for many hours of inspiring discussions.

# References

[1] Siva Anantharaman, Paliath Narendran, and Michaël Rusinowitch. Closure properties and decision problems of dag automata. *Information Processing Letters*, 94:231–240, 2005.

[2] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. Abstract meaning representation for sembanking. In *Proc. 7th Linguistic Annotation Workshop, ACL 2013 Workshop*, 2013.

[3] Johannes Blum and Frank Drewes. properties of regular DAG languages. In A.H. Dediu, J. Janoušek, C. Martín-Vide, and B. Truthe, editors, *Proc. 10th Intl. Conf. on Language and Automata Theory and Applications*, volume 9618 of *Lecture Notes in Computer Science*, pages 427–438, 2016.

[4] Johannes Blum and Frank Drewes. Language theoretic properties of regular DAG languages. Unpublished manuscript, 2016.

[5] Francis Bossut, Max Dauchet, and Bruno Warin. Automata and rational expressions on planar graphs. In *Mathematical Foundations of Computer Science 1988, MFCS'88, Carlsbad, Czechoslovakia, August 29 - September 2, 1988, Proceedings*, pages 190–200, 1988.

[6] Francis Bossut, Max Dauchet, and Bruno Warin. A kleene theorem for a class of planar acyclic graphs. volume 117, pages 251–265, 1995.

[7] David Chiang, Frank Drewes, Daniel Gildea, Adam Lopez, and Giorgio Satta. Weighted dag automata for semantic graphs. Unpublished manuscript, 2016.

[8] Witold Charatonik. Automata on dag representations of finite trees. Research Report MPI-I-1999-2-001, Max-Planck-Institut für Informatik, Saarbrücken, 1999.

[9] Frank Drewes and Jérôme Leroux. Structurally cyclic petri nets. *Logical Methods in Computer Science*, 11(4:15), 2015.

[10] Akio Fujiyoshi. Recognition of directed acyclic graphs by spanning tree automata. *Theoretical Computer Science*, 411:3493–3506, 2010.

[11] S. Rao Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 267–281, 1982.

[12] Michael Kaminski and Shlomit S. Pinter. Finite automata on directed graphs. *Journal of Computer and System Sciences*, (44):425–446, 1992.

[13] Tsutomu Kamimura and Giora Slutzki. Parallel and two-way automata on directed ordered acyclic graphs. *Information and Control*, 49:10–51, 1981.

[14] Tsutomu Kamimura and Giora Slutzki. Transductions of dags and trees. *Mathematical Systems Theory*, 15:225–249, 1982.

[15] E. W. Mayr. An algorithm for the general petri net reachability problem. *SIAM J. Comput.*, 13:441–460, 1984.

[16] Rohit J. Parikh. Language generating devices. Quarterly Progress Report 60, Research Laboratory of Electronics, M.I.T., 1961.

[17] Lutz Priese. Finite automata on unranked and unordered DAGs. In T. Harju, J. Karhumäki, and A. Lepistö, editors, *Proc. 11th Intl. Conf. on Developments in Language Theory (DLT 2007)*, volume 4588 of Lecture Notes in Computer Science, pages 346–360. 2007.

[18] Andreas Potthoff, Sebastian Seibert, and Wolfgang Thomas. Nondeterminism versus determinism of finite automata over directed acyclic graphs. *Bulletin of the Belgian Mathematical Society Simon Stevin*, 1:285–298, 1994.

[19] Daniel Quernheim and Kevin Knight. DAGGER: A toolkit for automata on directed acyclic graphs. In *Proc. 10th Intl. Workshop on Finite State Methods and Natural Language Processing*, pages 40–44. Association for Computational Linguistics, 2012.

[20] Daniel Quernheim and Kevin Knight. Towards probabilistic acceptors and transducers for feature structures. In *Proc. 6th Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 76–85. Association for Computational Linguistics, 2012.

[21] Wolfgang Thomas. On logics, tilings, and automata. In *Proc. 18th Intl. Coll. on Automata, Languages and Programming*, pages 441–454, 1991.